KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — 3001 Leuven

Dynamic Software Reconfiguration in Programmable Networks

Promotoren :
Prof. Dr. ir. P. VERBAETEN
Prof. Dr. T. HOLVOET

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Nico JANSSENS

December 2006

KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — 3001 Leuven

Dynamic Software Reconfiguration in Programmable Networks

Jury :
Prof. Dr. ir. G. De Roeck, voorzitter
Prof. Dr. ir. P. Verbaeten, promotor
Prof. Dr. T. Holvoet, promotor
Prof. Dr. ir. E. Steegmans
Prof. Dr. ir. E. Van Lil
Prof. Dr. ir. W. Joosen
Prof. Dr. G. Coulson
(Lancaster University, United Kingdom)

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Nico JANSSENS

U.D.C. 681.3∗C2

December 2006

D/2006/7515/95
ISBN 978-90-5682-761-8

Abstract

Programmable networks allow third parties to reprogram networking devices. By opening up the execution environment of routers, firewalls, gateways, etc., users and service providers can adapt the behavior of these devices to meet their own specific needs. Programmable networks are therefore an interesting technology to build adaptive networks and to support the increasing evolution of networking software.

At the same time, it can be perceived that many distributed applications impose stringent availability and performance requirements on the employed network infrastructure, among others to meet increased user expectations. Interrupting network communication to update or to customize the network software on programmable network devices hence may have extensive consequences. This dissertation, therefore, is targeted at supporting the reconfiguration of network software dynamically – that is, without temporarily shutting down (parts of) the network.

Whether or not such dynamic reconfigurations are beneficial depends very much on the effectiveness and efficiency of the reconfiguration process. Besides, implementing a correct reconfiguration that causes limited overhead can be very complex and error-prone (hence compromising the benefit of a dynamic reconfiguration). We argue that specific reconfiguration support is needed, therefore, which (1) conducts the effective and efficient reconfiguration of network software, and (2) conceals the complexity of these reconfigurations from users or service provides who initiate the actual reconfigurations.

This dissertation proposes the NeCoMan (*Ne*twork re*Co*nfiguration *Man*agement) middleware as reconfiguration support for programmable networks. In short, this middleware coordinates the runtime addition, replacement, and removal of both local and distributed network services among out-of-band active nodes. The novelty of this middleware is in its ability to tailor the reconfiguration process. To accomplish this, the NeCoMan middleware includes various reconfiguration algorithms as well as an extensive set of customizations to these algorithms. This enables NeCoMan to customize the reconfiguration process starting from (1) a declarative description of the recomposition that must be executed and (2) a specification of the network service characteristics and the reconfiguration semantics.

To conclude, we summarize the key contributions of this dissertation. Besides proposing a middleware to reconfigure out-of-band active nodes and validating this middleware by a number of reconfigurations, we present an extensive analysis on how to coordinate local and distributed out-of-band compositional adaptations. In addition, this dissertation proposes to make change management support customizable. In contrast to existing change management support (which typically conforms to the black-box philosophy by encapsulating a single and fixed reconfiguration algorithm) NeCoMan tailors the employed reconfiguration algorithm to exploit the network service characteristics and the reconfiguration semantics.

Voorwoord – Preface

Hoewel vaak onzichtbaar zijn computernetwerken alomtegenwoordig in ons dagelijkse leven. Het WWW, digitale TV en IP-telefonie, zijn slechts enkele toepassingen waarvoor een netwerkinfrastructuur onontbeerlijk is. Bovendien leggen vele van deze toepassingen strenge beschikbaarheids- en performantievereisten op aan deze netwerkinfrastructuur, onder andere als gevolg van toenemende gebruikersverwachtingen. Het onderbreken van de netwerkcommunicatie om de software van de gebruikte netwerkinfrastructuur bij te werken of aan te passen kan bijgevolg verstrekkende gevolgen hebben. Dit proefschrift onderzoekt daarom de dynamische herconfiguratie van netwerk-software – dat wil zeggen, de uitvoering van herconfiguraties zonder tijdelijk de werking van (een deel van) de netwerkinfrastructuur te onderbreken.

Bij het begin van dit proefschrift wil ik iedereen bedanken die (zowel op professioneel als op persoonlijk vlak) bijgedragen heeft tot de realisatie van dit werk. In de eerste plaats bedankt ik mijn promotoren, Prof. Pierre Verbaeten en Prof. Tom Holvoet, om mij de kans te geven dit onderzoek tot een goed einde te brengen. Hun kritische geest heeft me steeds gestimuleerd om te trachten het beste van mezelf te geven. Daarnaast wil ik hen en de andere leden van de begeleidingscommissie, Prof. Eric Steegmans en Prof. Emmanuel Van Lil, van harte bedanken voor het kritisch nalezen van deze tekst. Ook de voorzitter en de leden van de doctoraatsjury, Prof. Guido De Roeck, Prof. Wouter Joosen en Prof. Geoff Coulson wil ik langs deze weg bedanken. Prof. Geoff Coulson, thank you very much for kindly accepting to join the jury.

Onderzoek doe je zelden alleen. Ik wil dan ook de collega's van de DistriNet onderzoeksgroep bedanken voor de fijne samenwerking. In het bijzonder bedank ik Sam Michiels, Lieven Desmet, Alexander Helleboogh en Eddy Truyen van harte voor de vele vruchtbare discussies. Ook mijn (vroegere en huidige) collega's in de netwerkinggroep mag ik hierbij zeker niet vergeten. Frank, Tom, Thomas, Bart, Nelson, Klaas en Wouter, bedankt voor de aangename en leerrijke werkomgeving.

Werken op het departement computerwetenschappen (en binnen DistriNet in het bijzonder) is sowieso een uitzonderlijke ervaring. Alle mensen opsommen die mee voor deze unieke sfeer zorgden, is onbegonnen werk. Toch wil ik (op het risico om iemand te vergeten) een aantal mensen expliciet bedanken. Yves "mijn auto

heeft weer problemen" Younan, Frans "wanneer spelen we nog eens een wolvenspel" Sanen, Thomas "ik ben een beetje asociaal" Delaet, Jan "FY" Smans, Kurt "triestig gezichtje" Schelfthout, Alexander "witte vw-power" Helleboogh, Bart "da's toch simpel" Van Eylen, Peter "heeeeeuh" Rigole, Marko "alma" van Dooren, Elke "pannenkoekenkoffie" Steegmans, Yves "magic" Vandewoude, Bart "triatlon" De Win, Bart "de office cd's liggen *onderaan* de kast" Swennen en Jean "pc met of zonder besturingssysteem" Huens: bedankt allemaal voor die onvergetelijke jaren.

Verder wil ik ook mijn vrienden buiten cw bedanken. Bedankt voor de ontspannende surf- en skivakanties, klimmomenten, de avonden (en nachten) aan de toog. En vooral, bedankt voor jullie geduld, zeker het laatste jaar. Ik beloof dat ik nu terug meer tijd ga hebben voor jullie. Hou de agenda's al maar klaar!

Tot slot wil ik de mensen die het dichtst bij mij staan nog even expliciet in de kijker zetten. Bedankt, Raf en Lutgarde, voor al jullie aanmoedigingen. Bedankt, ma en pa, voor jullie onvoorwaardelijke steun. Bedankt, bomma en peter, voor de prachtige momenten die Steven en ik bij jullie hebben beleefd. Bedankt, Steven, voor de geweldige broer die je bent. Bedankt, Ellen, voor alles en zoveel meer ...

Nico Janssens, december 2006.

Contents

List of Figures

Listings

List of Tables

Chapter 1

Introduction

Although often invisible, computer networks have become ubiquitous in our daily life. The world wide web, on-line gaming, instant messaging, IP telephony and digital TV are only a few applications that highly depend on networked computer systems. At the same time, many distributed applications nowadays impose stringent availability and performance requirements on the employed network infrastructure, among others to meet increased user expectations. Interrupting network communication to update or to customize the network software on PC's, routers, gateways or other networking devices hence may have extensive consequences. This dissertation, therefore, is targeted at supporting the reconfiguration of network software dynamically – that is, without temporarily shutting down (parts of) the network.

To familiarize the reader with this research challenge, we start this introduction by exploring the benefits of dynamic software reconfiguration in computer networks. Subsequently, we discuss the pre-conditions that must be fulfilled for such dynamic reconfigurations to be beneficial. Starting from this problem statement, we explain the goal of this dissertation. Finally, we give a brief overview of the next chapters.

Note that we deliberately do not define all terminology used within this introduction. Most of the terms will be treated explicitly in Chapter 2.

1.1 Dynamic software reconfiguration in computer networks

Dynamic software reconfiguration has been subject of research since the mid 1980's. The interest in this topic was derived from an increasing need to support change in long-lived and highly-available distributed systems – such as banking applications, database servers, and telecommunication switches. These systems often require updates in their software over time to fix bugs, to add new features, or to optimize performance. Because long downtimes typically violate the stringent performance

and availability requirements these systems must satisfy, they cannot be taken off-line to accommodate change. The only option is to reconfigure them dynamically – that is, while they continue to operate.

Since the mid 1990's, part of the networking community is exploring the benefit that can be gained through dynamic software reconfiguration as well. This evolution coincides with two ongoing trends in network research and development. On the one hand, one can observe a shift towards *higher network programmability*. At the same time, an increasing interest emerges – both by academia and network industry – for computer networks to conceal *dynamic and heterogeneous network characteristics*, and to support *increasing network software evolution*. Let us explain this in more detail.

Network programmability

Traditionally, the key purpose of computer networks, such as today's Internet, is to deliver messages from one endpoint to another. Because distributed systems are only as effective as the network they run on, the main interest of network developers is to scale up the network's availability and performance. For a long time, this concern has been satisfied by keeping networks as simple as possible: the Internet's network functions, for instance, have been limited largely to routing, congestion control and simple QoS support [83, 112, 25]. Additionally, these network functions are typically abstracted away from end-users and applications. Case in point are the intermediate nodes (such as routers and switches) in present day networks, which are (mostly) closed vertically integrated systems whose functions are tightly programmed by the vendor into the embedded software and hardware.

On the one hand, this design principle is extremely powerful because it separates distributed applications from much of the complexity of the underlying communication system. At the same time, however, it constrains some of these applications because the detailed knowledge of the underlying network cannot be exploited to enhance performance [12]. For that reason, considerable interest has raised over the past decade to push extra functionality inside the network in an effort to provide better services in a cost-effective way. Examples include the transport-level support for wireless links of snoop-TCP [16], a network implementation of reliable multicast with congestion control [119], and network support for preserving the quality of MPEG video in the face of congestion [18]. Other more familiar examples of network services that require computations within the network include RSVP [24], Mobile IP [107], Web caches [52] and firewalls [116].

Since the mid 1990's, various initiatives seek to support this paradigm shift towards higher levels of network-internal computation by opening up the network infrastructure and increasing its programmability [28]. By providing support to customize the network software of both intermediate nodes and endpoints, the network as a whole becomes a fully programmable environment. In this manner, the entire network can be customized to improve its efficiency [83]. Similar to distributed

systems, at least two classes of programmable computer networks can benefit from accomplishing these software customizations dynamically. These include (1) *adaptive networks* that aim to conceal dynamic and heterogeneous network characteristics, and (2) networks that need to support *increasing network software evolution* without temporarily being taken off-line.

Adaptive networks

An increasing demand for networked consumer systems and devices (among other reasons) is currently transforming computer networks into dynamic and heterogeneous environments. End-users desire transparent networking of their mobile and embedded devices – such as sensors, mobile phones, PDAs and Pocket/Handheld PCs, digital TV set-top boxes, laptops and game consoles – to provide entertainment, information, and communication [45]. These devices inherently differ in available memory, processing and energy resources. Additionally, they employ various transmission technologies (such as IrDA, Bluetooth, 802.11 Wi-Fi, Wireless 3G, and wired 10/100 Ethernet), each with different and dynamic bandwidth, latency, and error characteristics.

The heterogeneous and dynamic nature of these networks is an important hurdle that must be overcome to seamlessly interconnect these devices – that is, both among themselves and with the required (internet) servers. For example, interconnecting end-user devices typically involves diverse transmission technologies that are characterized with different and dynamic bandwidth characteristics (e.g. 2.4 kbps - 16 Mbps for IrDA, 1 Mbps - 2 Mbps for Bluetooth, 11 Mbps - 54 Mbps for 802.11b/g Wi-Fi, etc. [1]). These differences, combined with the fact that the nodes along a communication path can also possess very different and dynamic capabilities (most true for the end-devices) can produce unsatisfactory performance for network-oblivious applications [49].

In addition to these network-oblivious applications, many existing protocols do not perform well either in such heterogeneous and dynamic networks. These protocols trade some loss in efficiency for the ability to deal with increased heterogeneity [88]. Case in point is the transmission control protocol (TCP), which is developed for wired networks that experience little transmission errors [111]. When employed in error-prone wireless networks, TCP exhibits poor performance because it interprets all perceived packet losses as caused by congestion. TCP thus incorrectly reduces its congestion window in case of packet loss due to wireless transmission errors. This causes a substantial degradation of performance in terms of throughput, even though sufficient bandwidth might be available [60].

A substantial amount of research effort targets to conceal dynamic and heterogeneous network characteristics by developing *adaptive* network support, which dynamically adapts the network software based on perceived changes inside the network [48, 49, 42, 88, 98, 99, 30, 133, 18, 96, 63]. Fu et al., for instance, present support to reduce performance degradation of network-oblivious applications by dy-

namic deployment of compression components in the face of low-bandwidth network conditions [48, 49]. Marcus and Feldmeier, for their part, have developed a number of "protocol boosters" to hide heterogeneous and dynamic network characteristics from existing protocols such as TCP [42, 88]. For both approaches, the employed network customizations are only effective under specific network conditions, and are therefore only applied when these conditions are fulfilled.

Besides, network industry as well is exploring the benefit of adaptive networks. Cisco Systems'® vision on "intelligent networking", for instance, targets to enable a network to adapt on its own to its environment, with minimal operator intervention [4, 17]. Besides, both Cisco Systems® and Check Point Software Technologies® are exploring adaptive network support to cope with new security threats [53, 5, 6].

Evolutionary changes in network software

In addition to dynamic and heterogeneous network characteristics, computer networks are becoming liable to evolutionary changes as well. The reason for this is twofold.

First, the pace of evolution in communication protocol specifications is increasing noticeably. The Internet Engineering Task Force (IETF), for example, has proposed and standardized more new protocols over the past five years than in the previous fifteen [125]. Additionally, some existing protocols have undergone various revisions over time to include extra functionality. Case in point are a number of RFCs[1] that discuss several extensions to TCP for improving the protocol's efficiency when operating over networks with specific characteristics [46, 64, 89]. This growing evolution of network communication protocols, in general, derives from the increasing rate in which network technologies and application requirements are currently changing. As long as existing protocols do not perform well in new circumstances, the pace of evolution in communication protocol specifications most likely will not slow down.

Second, increasing the level of network programmability inherently makes network software more liable to evolution. When pushing application specific functionality into the network software, for instance, the latter indirectly becomes exposed to changes that apply to the application software. In addition, because programmable networks allow third party actors to participate in the development of future network services, they enable a free-market approach to protocol and network software design. Consequently, protocol and network software designs can compete economically in the marketplace, rather than politically in a standards committee [42]. As for any other business, third party vendors therefore must be able to rapidly address changing stakeholder requirements, new market demands, etc. in order to be competitive in this new market.

This increasing demand to support evolutionary changes, however, conflicts with the stringent performance and availability requirements that most networks must

[1] The Request for Comments (RFC) document series is a set of technical and organizational notes about the Internet [2].

satisfy. Because computer networks are at the heart of all distributed systems, taking network nodes off-line to carry out (evolutionary) changes is difficult to coordinate and may have extensive consequences. Similar to distributed systems, computer networks may benefit from performing such adaptations while they continue to operate, so as to mitigate the costs and risks associated with (increasing) network software evolution.

> *By increasing network programmability, network developers are facing new challenges. These include the ability to accommodate changing circumstances that can be initiated by both network internal events (such as dynamic network characteristics) and network external events (such as rapid network software evolution). Similar to distributed systems, dynamic software reconfiguration is an enabling technology for programmable networks to address both types of dynamism without temporarily taking part of the network off-line.*

1.2 Problem statement

Whether or not dynamic reconfiguration of network software is beneficial depends very much on the effectiveness and efficiency of the reconfiguration process. In general, dynamic software reconfiguration (obviously) is only beneficial when the costs and risks it introduces do not outweigh those associated with shutting down and restarting the affected system. This pre-condition applies in particular to computer networks, since a dynamic reconfiguration of network software may cause disruptions, failures or inconsistencies that can be more harmful to the network than accomplishing the reconfiguration off-line.

Besides, implementing a correct reconfiguration that causes limited overhead can be very complex and error-prone (hence compromising the benefit of such a dynamic reconfiguration). This is especially true when the network administrator (who initiates an actual reconfiguration) must coordinate the reconfiguration himself/herself. We argue that specific reconfiguration support is needed, therefore, which (1) conducts the effective and efficient reconfiguration of network software, and (2) conceals the complexity of these reconfigurations from the network administrator.

1.3 Goal

Starting from this problem statement, we present the NeCoMan (*Ne*twork re*Co*nfiguration *Man*agement) middleware [65, 66, 70]. This middleware operates on top of programmable network nodes, and coordinates the runtime addition, replacement, and removal of both local and distributed network services. As we further explain in more detail in the next chapter, this middleware must fulfill the following requirements:

- **Correct reconfigurations.** To meet the requirement for effectiveness, NeCoMan must protect the network from failing because of the reconfiguration process. Stated differently, NeCoMan must conduct correct reconfigurations.
- **Limited reconfiguration overhead.** To meet the requirement for efficiency, NeCoMan must be able to conduct optimized reconfiguration scenarios. This involves taking into account context specific information such as the characteristics of the network services that will be reconfigured and the reconfiguration semantics.
- **Limited openness.** To conceal the complexity of dynamic network software reconfiguration, NeCoMan's reconfiguration API must be limited. A network administrator should only be able to specify *what* reconfiguration NeCoMan must execute, instead of also defining *how* this reconfiguration must be coordinated.
- **Reusability.** The dependencies between the NeCoMan middleware and the nodes that it reconfigures must be minimal. This enables to deploy NeCoMan on top of different node architectures[2].

1.4 Overview

The remainder of this text is structured as follows. Chapter 2 clearly delimits the scope of this dissertation in the context of programmable networks, dynamic software reconfiguration, and change management research. Next, Chapters 3 and 4 explain how the NeCoMan middleware executes local reconfigurations. Chapters 5 and 6 then elaborate on distributed reconfigurations. After that, Chapter 7 presents the design of the NeCoMan middleware and evaluates the reconfiguration overhead that NeCoMan brings about. Next, Chapter 8 situates our approach with regards to related research in the field of programmable networking. Finally, Chapter 9 summarizes the main achievements presented in this dissertation, and identifies future research tracks that spin off from this research.

[2]As we further explain in Chapters 2 and 3, these node architectures must (1) must support compositional adaptation, and (2) provide a predefined set of reconfiguration operations for NeCoMan to invoke

Chapter 2

Background and scope

The objective of this chapter is to clearly delimit the scope of this dissertation. As illustrated in Figure 2.1, this dissertation brings together the following research topics: programmable networks, dynamic software reconfiguration and dynamic change management. Over the past decade, interest in these topics has increased significantly, resulting in numerous projects that each target various concerns. Sections 2.1, 2.2, and 2.3 therefore position this dissertation in each of these research areas, respectively. Next, Section 2.4 elaborates on the characteristics of the network services that we target for dynamic reconfiguration. Section 2.5 then distinguishes between service-internal and service-external communication ports. After that, Section 2.6 motivates in more detail the four requirements that NeCoMan must fulfill. Finally, Section 2.7 presents a detailed overview of the following chapters.

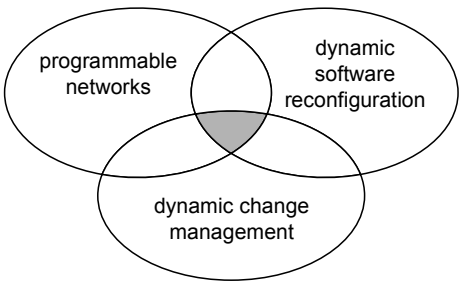


Figure 2.1: Research context

2.1 Programmable networks

Traditional data networks provide a transport mechanism to transfer bits from one end system to another. To scale up their performance and availability, processing within these networks was limited largely to routing, congestion control and simple quality of service (QoS) support [83, 112, 25]. As already mentioned in the introduction[1], considerable interest has raised over time to push more and more functionality *inside* the network, so as to provide better network services to users in a cost-effective way. Unfortunately, the closed and inflexible nature of today's networks is an obstructive factor in this evolution[2]. The current process to change network protocols and services is both lengthy and difficult because it requires standardization, adaptation of vertically integrated network devices, and manual, backwardly-compatible deployment [141]. For example, there was a span of five or more years from the time the Resource Reservation Protocol (RSVP) was conceptualized to the time it was deployed, even in a very limited manner [55].

Programmable networks seek to address the increasing demand for network software programming by *opening up the network*, and by *simplifying and accelerating its programmability* in a secure and controlled manner. As a result of building programmability into the network infrastructure, programmable networks enable to create, deploy, and manage novel network services without going through a lengthy and difficult standardization and deployment process [28]. Programmable networks thus aim to transform the employed network infrastructure into a flexible computing environment where the behavior of routers, firewalls, base stations, etc. can be changed rapidly in response to new requirements.

Over the past decade, the programmable networks research community has investigated many aspects of network programmability, resulting in a number of toolkits [141, 14], network programming models [36, 144, 133, 99, 143, 25], special-purpose network programming languages [55, 102], node operating systems [95, 108], node programming architectures [18, 12, 75, 119, 71], and protocols [142, 10]. Because the programmable networks umbrella covers such a wide variety of research projects that each target various concerns, it is important to clearly delimit the scope of this dissertation in the field of programmable networking. This scope encompasses *dynamic software reconfiguration* in *out-of-band active networks*.

In the remainder of this section, we first compare and contrast the characteristics of such *out-of-band active networks* with other programmable networking initiatives[3]. To do so, Subsection 2.1.1 outlines a reference architecture Coulson et al. have proposed in [36] to study the design space of programmable networking. Next, Subsection 2.1.2 distinguishes between in-band and out-of-band network

[1] more specifically, in Section 1.1 (page 4)

[2] except for vendor-specific network function customizations that are transparent to end-user systems

[3] presenting a complete survey of programmable networks research is beyond the scope of this dissertation; for an overview of a broad spectrum of programmable networks projects we refer to [126, 28, 112]

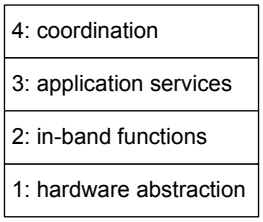


Figure 2.2: A reference architecture for programmable networking, presented by Coulson et al. in [36].

programming. Subsection 2.1.3 then discusses three approaches that have emerged over time to build programmability into the network infrastructure, including the out-of-band active approach. Finally, Subsection 2.1.4 reflects on *dynamic software reconfiguration* in programmable networks, aiming at out-of-band active networks in particular.

2.1.1 A design space of programmable networking

To classify programmable network research, Coulson et al. have proposed an interesting reference architecture to represent the design space of programmable networking [36]. As illustrated in Figure 2.2, this (highly abstract) architecture contains four strata: a *hardware abstraction* stratum, an *in-band functions* stratum, an *application services* stratum, and a *coordination* stratum. Because this reference architecture is an interesting means to clearly delimit the programmable networking functionality that this dissertation targets, we describe each of these strata in the remainder of this subsection.

Hardware abstraction stratum

The hardware abstraction stratum contains the operating system functionality of programmable nodes. This functionality serves to enable higher-level programmability, among others by providing abstractions to access node resources – including computing cycles, storage, and transmission bandwidth – in a controlled and secure manner. Additionally, functionality in this stratum shields the underlying hardware heterogeneity, so as to offer a uniform API to the node programming functionality of the upper strata.

Examples of programmable networking initiatives that focus on this stratum include the Active Network Encapsulation Protocol (ANEP) [10] and ActiveIP [142], which both specify a mechanism to encapsulate programmed frames for transmission over today's Internet infrastructure. Other projects that address stratum 1 concerns include node operating systems such as Bowman [95], a node os serving the CANEs network programming functionality [119], and NodeOS [108], which tar-

gets portability of node programming functionality across different types of physical nodes by exposing a common, standard interface.

In-band functions stratum

The in-band functions stratum comprises low-level packet processing functions that affect most or all packets. Examples of such functionality include, among others, packet filtering, classification, scheduling, header processing, and queueing. Because of their very nature, these low-level in-band functions are performance critical; their implementation efficiency has a direct bearing on the network performance.

Examples of programmable networking projects that are targeted at this stratum include a number of software architectures for building flexible and configurable routers, such as the Click modular router [75], VERA [71], Washington University's pluggable router framework [38], the NetBind toolkit [26], and the PromethOS NP framework [117]. In addition, also part of the Mobiware toolkit functionality [14] – which aims to improve QoS in mobile system by supporting the introduction of new adaptive mobile services at the transport, network and datalink layer – focuses on the in-band functions stratum.

Application services stratum

The application services stratum, as its name already suggests, covers higher-level application-related network services. In contrast to stratum 2 functionality, this stratum comprises coarser grained network services that apply to pre-selected packet flows in application specific ways. These services therefore are less performance critical.

Examples of programmable networking initiatives that (partially) focus on this stratum include many "active networks" projects [141, 55, 56, 102, 9, 120, 131, 39, 144, 10, 142]. As we further discuss in the next section, these networks enable applications to insert application-specific control in the data path, thus customizing the way packets in transit are processed. In addition, also the Active Services framework Amir et al. presented in [12] addresses stratum 3 concerns. The authors of this framework deliberately restricted the network's programmability to the application layer, among others to preserve all the routing and forwarding semantics of the current Internet architecture. This way, the Active Services framework can be incrementally deployed in today's Internet. A last example of programmable network support that is targeted at this stratum is Nakao's architecture to establish network paths for playing media objects (such as MPEG video, MP3 audio, JPEG images, etc.) [103]. Depending on the resources available at all nodes that a media object must traverse to reach its destination, Nakao's architecture distributes portions of the media processing functionality (such as, for instance, transcoding behavior) among the intermediate nodes that connect the source to the destination node.

Coordination stratum

Finally, the coordination stratum includes or supports out-of-band signaling protocols that perform distributed coordination (e.g. configuration, reconfiguration) of the lower strata [36]. Examples of programmable networking projects that are partially targeted at this stratum include Genesis [27], Mobiware [14], and Darwin [29]. Part of these architectures' functionality is responsible for coordinating the distributed execution of their customized router services. Other initiatives that address stratum 4 concerns include Ensemble [133] and Cactus [30]. These frameworks for constructing adaptive protocol stacks provide support to coordinate dynamic reconfiguration of distributed services and communication protocols.

2.1.2 In-band versus out-of-band network programming

Besides the networking functionality they focus on, programmable network projects differ as well by their ability to program nodes in an out-of-band or an in-band fashion [23]. In case of out-of-band network programming, node programs[4] and payload data are distributed by logically and/or physically distinct communication channels. Consequently, as Figure 2.3 illustrates, network programs and payload data are carried discretely – that is, within separate packets –, and are exchanged by a separate control and data plane. Data packets thus cannot individually adapt the behavior of network nodes.

In case of in-band network programming, network programs and payload data are carried in an integrated fashion (as illustrated in Figure 2.4). Network packets in this case may contain both data and network programs. Consequently, the data and control planes of such in-band programmable nodes are combined (in what is often referred to as the node execution environment). When a packet arrives at a programmable node, this node executes the packet's program (if allowed), optionally using the payload data as input[5].

2.1.3 Network programming paradigms

As stated above, this dissertation focuses on *out-of-band active networks*. To clearly define this type of programmable networks, we first elaborate on two other paradigmatic approaches (discussed by Calvert et al. in [25]) that have emerged over time to build programmability into the network infrastructure: *open signaling* and *active networks*. In addition to both approaches, Coulson et al. have identified *out-of-band active networks* as a third paradigmatic approach that positions between open signaling and active networks [36]. As we further discuss in the remainder of this

[4] These node programs may contain, at one extreme, a scalar argument to select a pre-defined computation at the network nodes, or at the other extreme, mobile code written in a Turing-complete language that must be interpreted and executed by the network nodes [25].

[5] Note that the execution of these in-band programs may require out-of-band *deployment* of processing routines that are unavailable at the moment of execution.

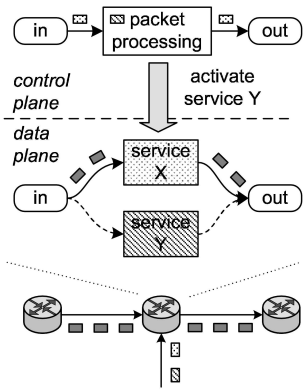


Figure 2.3: Out-of-band network programming

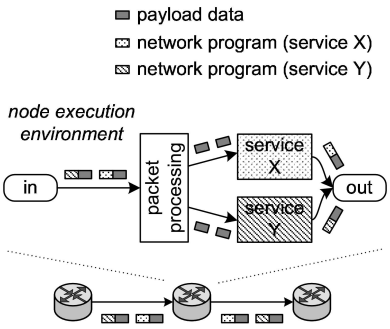


Figure 2.4: In-band network programming

section, all three approaches differ, among others, by the strata of Coulson's reference architecture they target, as well as by their ability to support in-band or out-of-band programming (see Table 2.1 for an overview).

A first approach to increase network programmability is advocated by the *open signaling* (Opensig) community. This community mainly focuses on extending routing and switching hardware with well-defined programming interfaces to open up these devices. Such interfaces allow service providers to (re)configure network devices remotely in an out-of-band manner, among others by using middleware toolkits or specific signaling protocols. Besides, open signaling takes a telecommunications approach to make the network programmable by clearly separating network control and management functionality (located at the control plane) from data transport (located at the data plane) [28]. Furthermore, this approach primarily targets network devices that provide some level of QoS support, such as IP routers [29], ATM switches [132], and nodes of mobile networks [14]. Research projects that adopt the open signaling approach include Mobiware [14], the Tempest framework [132], Genesis [27], and Darwin [29]. These projects tend to focus mostly on strata 2 and 4 of Coulson's programmable networking reference architecture.

The *active networks* (AN) approach [126, 112] is an order of magnitude more flexible: it gives applications control over the network services and allows them to tailor these services to application needs. Active network technology achieves its flexibility by allowing (untrusted) applications to insert application-specific control in the data path. Typically, imperative programs written in a specialized language are carried in network packets (which are referred to as active packets). Such languages can be seen as glue languages that compose node extensions together to form a customized service. These node extensions are custom pieces of software that are dynamically installed at the various nodes and form the basic building blocks

approach	*adaptation*	*strata*	*flexibility*	*safety & security vulnerability*	*deploy-ability*
active networks	in-band	1 & 3	+++	higher	–
out-of-band active networks	out-of-band	2 & 3	++	medium	+
open signaling	out-of-band	2 & 4	+	lower	+

Table 2.1: Comparison of active networks, out-of-band active networks and open signaling networks

for more elaborate services. In contrast to the open signaling approach, active networks thus support in-band customizations of network services at packet transport granularity, rather than out-of-band reconfigurations through a dedicated open control interface. This originates from a history of AN projects that have focused primarily on IP networks, where the control and data paths are combined. Such research projects include ANTS [141], the Active Service Framework [12], PLAN [55], PLANet [56], SNAP [102], SwitchWare [9], smart packets [120], M0 [131], Dan [39], NetScript [144], ANEP [10], and ActiveIP [142]. These projects tend to focus mainly on strata 1 and 3 of Coulson's programmable networking reference architecture.

In addition to these two schools of thought on how to make networks programmable, Coulson et al. have identified a third paradigmatic approach which they refer to as *out-of-band active networks* [36]. This paradigm promotes a restricted active networking approach. Similar to regular active networks, out-of-band active networks allow to dynamically install downloadable node extensions onto IP routers. The behavior of these network nodes, however, is not customized by (untrusted) packets passing by, but instead become reconfigured in an out-of-band manner by external (expert) actors such as network administrators, network engineers, adaptive network management software, etc. Once node extensions are deployed and brought into use, they are employed just as any other service is employed in today's closed networks. Research initiatives that adopt the out-of-band active approach include VERA [71], Click [75], Ensemble [133], Cactus [143], DiPS+ [97, 99], router plugins [38], and the dynamic-reconfigurable protocol stack that Lee et al. presented in [82]. These projects tend to focus mostly on strata 2 and 3 of Coulson's programmable networking reference architecture.

Furthermore, the out-of-band active networking approach positions between the two classic approaches in terms of *flexibility*, *security and safety vulnerability*, and *deployability* (see Table 2.1 for an overview). We illustrate this in the remainder of this subsection.

Flexibility

The ability for applications to tailor network services according to their specific needs potentially makes *active networks* the most flexible of all three approaches. We deliberately say 'potentially', because the flexibility of an active network depends among others on the degree of network programmability. At one extreme, packets (sometimes referred to as "capsules" [141]) carry mobile code that will be executed at all nodes in transit. At the other end of this spectrum, packets may contain a scalar argument to select a pre-defined computation at the network node [25]. Other initiatives such as the PLAN [55] and SNAP [102] languages position between both extremes. These special-purpose languages restrict the expressiveness of programs carried by active packets, so as to increase among others the safety, security, and performance at which new services can be introduced in the infrastructure.

In addition, the flexibility of active networks relates to the supported "granularity of control" [25] as well, which indicates the scope of node behavior that can be modified by a received packet. As Calvert et al. describe in [25], at one end of this spectrum, a single packet can modify the node behavior seen by all packets arriving at a node. At the other extreme, a single packet modifies the behavior seen only by that one packet. Between these extremes, modifications may apply to a packet flow, which can be defined as a set of packets sharing some common characteristics.

Out-of-band active networks, in contrast, are less flexible than regular active networks. Recall that the behavior of these network nodes is not customized by packets passing by, but instead become reconfigured in an out-of-band manner by external (expert) actors. Similar to regular active networks, out-of-band active networks are concerned with supporting the deployment of an extensive set of new IP services. The flexibility of out-of-band active networks thus also depends on the degree of network programmability. Because network customizations become initiated in an out-of-band manner, however, the granularity of control is more coarse grained than for regular active networks: a modification may apply to a packet flow in case of a stratum 3 service, or to all packets arriving at the node in case of a stratum 2 function.

Finally, many *open signaling* architectures are less concerned with attaining maximum flexibility. Their main concern is to provide controlled QoS support for a given context, such as mobile networks, rather than supporting the introduction of a virtually unlimited set of novel services.

Security and safety vulnerability

While network programmability increases flexibility, it also raises safety and security concerns. According to the terminology used in [55], safety refers to reducing the risk of mistakes or unintended behavior, while by security we mean the usual concept of protecting privacy, integrity, and availability in the face of malicious attacks.

Since flexibility has a direct bearing on security and safety vulnerabilities, the regular *active networking* approach is perceived as more prone to these threats than

the out-of-band active networking and open signaling approaches. Because many active network platforms allow (untrusted) applications to attach arbitrary code to data packets, solid security mechanisms must prevent the execution of malicious code in a shared network infrastructure. These demanding security precautions – in the worst case the code of every packet must be controlled – consequently also affect the performance of programmable networks. In addition, network code supplied by application programmers will be more error-prone. This results, among others, from the problem that application programmers typically are not network specialists. They know the application domain very well, but in general they do not know how to develop a network service. We call this problem domain mismatch: application programmers, who wish to customize the way their applications are executed, are suddenly confronted with a domain (in this case, active networks) that is completely different from the familiar application domain.

Out-of-band active networks, in contrast to regular active networks, can control and secure network software customizations more easily. As already mentioned above, these customizations are not triggered by (untrusted) packets passing by, but instead become initiated in an out-of-band manner by external (expert) actors. This has a direct bearing on safety and security vulnerability. First, because node extensions are developed by network experts instead of application developers, these extensions are considered as less error-prone. Second, the out-of-band installation of new network extensions enables to employ a set of existing security mechanisms including encryption, program verification, and authentication. Third, recall that out-of-band customizations typically do not operate at packet-level granularity, but instead apply to a packet flow or to all packets arriving at the customized nodes. Security checks thus are executed at a lower frequency as for most regular active networks, which reduces their impact on network performance.

Finally, because *open signaling* networks often provide a more restricted level of network programmability, they can be perceived as the least vulnerable to security and safety threats of all three approaches to increase network programmability.

Deployability

A final characteristic to compare these approaches is the ability to be deployed in legacy networks. Recall that regular active networks support in-band customizations of network services at packet transport granularity. The packets that the nodes of these networks exchange may carry network programs as well as payload data. When the format of these packets does not conform to the protocol specifications adopted by legacy network nodes (such as for instance the packet format defined by the IP standard), these packets will not be interpreted correctly by legacy nodes and thus may become discarded. Additional support is needed to enable the integration of such active network nodes in a legacy network environment. Protocols like ActiveIP [142] and ANEP [10] seek to achieve this by tunnelling active packets over legacy network nodes.

Both out-of-band network programming approaches, in contrast, can be deployed more easily. Because these networks become programmed in an out-of-band manner, node programs and payload data are distributed by logically and/or physically distinct communication channels. As this involves the use of legacy networks, no tunnelling support is needed to enable the large-scale deployment of out-of-band active networks and open signaling networks. Note, however, that this requires for the programmable nodes to preserve all the routing and forwarding semantics of the "hosting" network architecture.

2.1.4 Dynamic reconfiguration in out-of-band active networks

The majority of programmable network architectures do not support dynamic software reconfiguration[6]. These network architectures enable the initial deployment of specific services, but do not support subsequent reconfigurations of services that are already in use, among others because they encapsulate state [58]. However, this imposes two restrictions on programmable networks.

First, it limits the ability of programmable networks to accommodate *evolutionary changes*. In PLANet, for instance, some components are always in use; the packet queue may always contain packets, and therefore may never be upgraded [58]. Second, it restricts the ability to provide *adaptive network support* when memory resources on network devices limit the number of service components that can be stored simultaneously. If this is the case, dynamic software reconfiguration enables to implement adaptive support by switching service components in and out of lightweight programmable nodes at runtime on an as-needed basis.

This dissertation, therefore, investigates dynamic software reconfiguration in programmable networks. More specifically, and as already mentioned a few times before, we target *dynamic software reconfiguration in out-of-band active networks*. Although we do not claim that this programmable networking approach will become dominant in use, we believe it has significant potential to enable network programmability in a controlled and secure way.

2.2 Dynamic software reconfiguration

Over the past two decades, researchers and developers have investigated various aspects of dynamic software reconfiguration. As Subsection 2.2.1 illustrates, this has resulted in a wide variety of methods, tools, and techniques to enable structural software reconfiguration without temporarily shutting down (part of) the system. Many of these approaches, however, do not assure that system consistency will be preserved after reconfiguration. In the context of programmable networks, this may

[6]Exceptions to this include Click [75], Ensemble [133], Cactus [30], Netkit [36], and Lee's dynamically reconfigurable protocol stack [82]

potentially break the stringent availability requirements that computer networks must satisfy. Subsection 2.2.2 therefore elaborates on how to leave a system in a consistent state after reconfiguration such that it can continue functioning normally.

The next subsections aim to clearly delimit the scope of this dissertation in the context of dynamic software reconfiguration, which encompasses *dynamic compositional adaptation* in *pipe-and-filter (network) software architectures*. Subsection 2.2.3 first introduces compositional adaptation, and then discusses three key enabling technologies: separation of concerns, component-based design, and computational reflection. Next, Subsection 2.2.4 elaborates on pipe-and-filter based (network) software architectures, and discusses how these architectures promote the development of flexible (network) software. Finally, Subsection 2.2.5 briefly presents the characteristics of DiPS+, a pipe-and-filter network architecture developed at DistriNet which has served as network programming environment to validate the NeCoMan middleware.

2.2.1 A brief overview of dynamic software reconfiguration approaches

The increasing pace of research in the area of dynamic software reconfiguration has resulted in a wide variety of methods, tools, and techniques to support structural software reconfiguration without temporarily shutting down (part of) the system. To familiarize the reader with the diversity of research in this area, this section presents a brief overview[7] of research that targets dynamic software reconfiguration. We classify this research by the supported *unit of adaptation*, as well as by the employed *structural reconfiguration mechanism*.

Unit of adaptation

An important aspect of dynamic software reconfiguration is the supported unit of adaptation, as this impacts the granularity at which structural adaptations can be performed. We define this unit of adaptation as follows:

> *A unit of adaptation represents the most fine-grained software abstraction that is subject for structural change.*

Evidently, a reconfigurable system's unit of adaptation highly depends on the adopted programming language model. Fabry's work on dynamic software reconfiguration, for instance, enables to dynamically upgrade the implementation of *abstract data types* (ADTs) [41]. Other research efforts focus on dynamic software adaptation of *procedure-oriented systems*, thus operating at the granularity of procedures (also called functions, routines, subroutines, or methods, depending on the programming language). These research projects include PODUS [121], and the work of McNamee [93], Hicks [57], and Lyu [85].

[7] This is by no means an exhaustive overview. For a more detailed overview of research efforts to support dynamic software reconfiguration, we refer to a number of interesting surveys [47, 8, 94, 91].

A different research thread in the area of dynamic software reconfiguration targets *object-oriented systems*. In short, object-oriented programming (OOP) is a programming language model that presents a software system as a collaboration of modules ("objects") that each encapsulate their own internal state and execute a common task by invoking each other's interface. Additionally, each object is a specific instance of a "class", which specifies a template definition of the methods and variables for a particular set of objects. In this context, projects such as JDrums [114], Kava [140], and the work of Malabarba [87] have investigated how to customize class structures, variable types, object collaborations, etc. at run-time.

Additionally, a vast majority of research on dynamic software reconfiguration targets *component-oriented systems*. Component-oriented programming (COP), in contrast to object-oriented programming, aims to build software systems through the composition of predefined software modules ("components"). As we further discuss in Section 2.2.3, the explicit notion and reification of software composition makes component-oriented systems very suitable for dynamic software reconfiguration – often referred to as dynamic software (re)composition. Many research projects that focus on dynamic software reconfiguration, therefore, have investigated how to dynamically change the behavior of component-oriented systems. These include Simplex [122, 115], SOFA/DCUP [109], Hercules [32], Netkit [36], DiPS/CuPS [68], Draco [134], ACT [118], OpenORB [20], 2K [76], the CIAO/QuO middleware [138], and the work of Feng et al. [43].

Another research thread on dynamic software reconfiguration targets *aspect-oriented systems*. In short, aspect-oriented programming (AOP) seeks to enable the modularization of system concerns that inherently crosscut multiple software modules – such as security, logging, and persistence, to only name a few. Object-oriented techniques for implementing such crosscutting concerns (called "aspects") most often result in systems that are invasive to implement, hard to understand, and difficult to maintain and adapt [73, 105]. The AOP community has recognized this problem and investigates how to separate these crosscutting concerns from the system's functional logic. In this context, research projects such as Lasagne [128, 129, 130], PROSE [110], and Handi-Wrap [15] have explored how to dynamically reconfigure these crosscutting concerns.

Finally, a number of research projects implement dynamic reconfiguration by replacing the node processes collaborating in a distributed system. Kramer and Magee justify these coarse grained adaptations by arguing that software reconfiguration at the granularity of fine grained programming elements is at too low a level, being both too detailed and impractical due to the tight coupling with other program elements [79]. Research projects that focus on such coarse grained reconfigurations include Conic [79], Polylith [62, 113], and the work of Kindberg [74], and Goudarzi [101, 100].

Structural reconfiguration mechanism

In addition to the unit of adaptation, most dynamic reconfiguration projects differ as well by the employed mechanism to enable structural change. Table 2.2 lists a number of these mechanisms, along with some references to systems that implement them. These mechanisms can roughly be classified into three categories: *indirection mechanisms*, *relinking mechanisms*, and *mechanisms based on design patterns*.

Indirection mechanisms exploit a level of indirection in the interactions between software entities. This enables to reconfigure a system's software structure by intercepting and redirecting interactions among different software entities. These indirections, however, introduce a performance cost when software entities invoke each other during regular execution.

Relinking mechanisms, in contrast, do not exploit indirections but carry out a reconfiguration by relinking all references from old to new software entities. To change a system's software structure, the reconfiguration system[8] must trace all references that are involved and fix them. The main advantage of this mechanism is that it reduces the overhead caused by indirections. A drawback is that the reconfiguration system must keep track of all software entities to make sure that every broken link becomes repaired.

Finally, *mechanisms based on design patterns* seek to achieve structural software reconfiguration in a language-neutral manner by focusing on the system's software design. In [50], Gamma et al. define design patterns as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context". As illustrated in Table 2.2, a number of these design patterns aim to increase software flexibility, which makes them an interesting means to develop software systems that support structural reconfiguration.

2.2.2 Consistency preservation

As illustrated in the previous section, much research in the area of dynamic software reconfiguration has focused primarily on providing support to enable dynamic reconfiguration of software systems. Most of these enabling technologies do not assure that system consistency will be preserved after reconfiguration – that is, they do not assure that system parts interacting with entities under reconfiguration do not fail because of the reconfiguration. For dynamic software reconfiguration to be effective, however, this is an essential pre-condition: after completing a reconfiguration, the system must be left in a "correct" state such that it can keep on functioning properly. After all, the effort in reconfiguring a system would be in vain if a faulty reconfiguration process compromises the system's correct functioning. This applies in particular to programmable networks, which have to satisfy stringent availability requirements.

[8] which conducts the actual reconfiguration

indirection mechanism	**description**	**refs**
function pointer indirection	Enables to dynamically adapt a program's execution path by altering the function pointer.	[41, 93]
meta-object protocol	Allows to modify a program behavior's by supporting introspection and intercession (for more details, see Section 2.2.3).	[20, 140, 78, 130]
debugging support	Enables to intercept and redirect method invocations while executing the software system in debugging mode.	[110]
relinking mechanism	**description**	**refs**
code relinking	All links to the old programming entities become redirected to refer to the new ones	[57, 82]
architectural connectors	These architectural abstractions mediate communication between software modules (both local and distributed). A reconfiguration thus involves replacing these connectors.	[104, 68, 80, 101]
design pattern	**description**	**refs**
proxy pattern	Provides a surrogate or placeholder for another object to control access to it.	[118, 74, 43]
strategy pattern	Encapsulates a family of algorithms and makes them dynamically interchangeable	[78]
decorator pattern (wrapper)	Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality	[130]

Table 2.2: An overview of structural reconfiguration mechanisms

As Goudarzi describes in [100], a dynamic software reconfiguration yields a correct system if after completing the reconfiguration process:

1. the system satisfies its *structural integrity* requirements,

2. the entities in the system are in *mutually consistent states*, and

3. the *application state invariants* hold.

We elaborate on these pre-conditions for safe dynamic software reconfiguration in the remainder of this section.

2.2.2.1 Structural integrity

A first pre-condition for safe dynamic software reconfiguration relates to the system's software structure: after completing a reconfiguration, the system must still satisfy its structural integrity requirements. As Almeida correctly states in [11], these structural integrity requirements constrain the structure of a system in terms of the relationships between collaborating software entities[9] and the ways in which these entities must be put together. A system's structural integrity requirements thus define how its software entities must be inter-connected such that the system can operate correctly. Preserving structural integrity after reconfiguration therefore involves maintaining *referential integrity*, *interface compatibility*, and *(distributed) dependencies*.

Referential integrity. Referential integrity may be broken when *replacing* software entities. Once an entity is replaced, all its client processes must be redirected consistently to invoke the new software entity instead of the old version. In some cases, however, the entities to be replaced may not be aware of their client processes. Feng et al. refer to this as the "referential transparency problem" [43]. Additionally, in many cases these client references must be redirected atomically, so as to prevent inconsistencies caused by some client processes that still invoke the old software entities while others already invoke the new versions.

Various research projects have investigated how to preserve a system's referential integrity. Feng et al., for instance, solve the referential transparency problem by employing proxy-objects [43]. These proxy objects provide placeholders for replaceable objects. Because client processes invoke these proxies instead of the associated replaceable objects, the latter become decoupled from their client processes. Consequently, replacing such an object involves only redirecting its (single) proxy object, instead of redirecting each one of its client processes. Gilgul adopts the same principles but implements them in a different way to support transparent upgrades of Java objects [33, 34]. A reference to an object in Gilgul's model is realized as an object oriented pointer, which points to an entry in an object table that holds the actual memory address of the object. This enables to replace objects at runtime transparently for their client processes by changing the associated memory addresses. Almeida et al., for their part, target referential integrity in the context of dynamic software reconfiguration for CORBA middleware [11]. In order to re-establish broken bindings after reconfiguration, they provide a central point of contact for clients to find the objects with invalidated object references. Finally, JDrums employs a modified Java Virtual Machine (JVM) to preserve referential integrity when upgrading Java classes [114]. After loading a new class into the modified JVM, the objects of the old class are converted to the new version when they are invoked (lazy upgrades). Once a new object is initialized, JDrums replaces this object's old

[9] A software entity may be a procedure, object, component, aspect, etc., depending on the unit of adaptation supported by the employed programming model.

reference in the JVM with a reference to the new object, such that only the latter will be invoked afterwards.

Interface compatibility. Besides preserving referential integrity, interface compatibility must be dealt with as well to preserve a system's structural integrity. When replacing a software entity with a new version, the latter must satisfy the interface definition of the original version for the associated client process to continue operating normally. Systems like SOFA/DCUP [109], JDrums [114], and Almeida's extension of CORBA [11] meet this requirement by adopting Liskov's substitution principle [84]. These systems support the upgrade of an old software entity when the new version implements the original interface or when it implements an interface that is derived from the original one. This way, a software entity can be replaced transparently without compromising the functioning of its client processes.

Dependency preservation. Finally, preserving a system's structural integrity requires to maintain all dependencies between the system's collaborating software entities as well. When software entities need to collaborate for the system to perform correctly, breaking these dependencies during a reconfiguration compromises the correct functioning of that system. To illustrate this in the context of programmable networks, suppose two neighboring network nodes are equipped with a compression and a decompression entity, respectively. The node hosting the compression entity thus expects from its neighbor node to restore compressed data packets into their original state. To prevent breaking this dependency, structural changes that affect both entities (such as exchanging the employed compression algorithm) must be carried out atomically. If this is not the case, for example, packets processed by the old compression entity may not be decompressed correctly when the neighbor node is already upgraded to use the new algorithm[10]. Breaking the dependency between both entities thus compromises the correct functioning of both the compression service and the network.

Various research projects have focused on preserving distributed dependencies during dynamic reconfiguration. The earliest work on atomic distributed reconfigurations appears to be Bloom's research on upgrades in Argus [21, 22], where one dedicated node coordinates distributed reconfigurations among the other ones. Ensemble [133] and Cactus [30], in contrast, conduct atomic distributed reconfigurations by employing a distributed coordination protocol. Lasagne adopts a different approach to preserve distributed dependencies. This system provides system-wide structural consistency by having customizations propagated together with the message flow of an entire collaboration [128, 129, 130]. Finally, Ensink et al. preserve distributed dependencies by scheduling the local reconfiguration operations [40].

[10] presupposing that this new decompression entity is not backward-compatible with the old version

2.2.2.2 Mutually consistent execution states

In addition to preserving structural integrity, a second pre-condition for safe dynamic software reconfiguration relates to the execution state of the software entities that will be reconfigured: after completing a reconfiguration these states must be mutually consistent, such that a system can continue processing normally rather than progressing towards an error state. To illustrate this, consider the replacement of a database-service while being in the middle of processing a query. In that case, the client process may continue waiting for a reply that will never arrive since the new database-service is unaware of previously ongoing requests. Consequently, if mutually consistent execution states are not preserved after completing a dynamic reconfiguration, the costs and risks introduced by failures and inconsistencies may outweigh those introduced by conducting the reconfiguration off-line.

Many different approaches have been proposed to preserve mutually consistent execution states in the context of dynamic reconfigurations. Kramer and Magee, for instance, have stated that software modules must be both *consistent* and *frozen* [80, 81] before being reconfigured. When software modules are consistent, they do not contain results of partially completed services (or transactions). By freezing software modules, new transactions are prevented from executing and thus cannot cause state changes. Kramer and Magee define this consistent and frozen state as the *quiescence* of a software module. They propose a mechanism to impose such a quiescent state by means of a configuration manager, which recognizes and finishes (or deactivate) the relevant transaction initiators [80]. Besides Kramer and Magee's implementation, various other projects provide support as well for their software modules to reach a quiescent state. These include Cactus [30], Ensemble [133], and the work of Goudarzi [101, 100] and Almeida [11].

As an alternative, Hofmeister has proposed to freeze a software module immediately instead of waiting for it to reach a desirable state [61]. However, since in this case there is no guarantee about the termination of a transaction in progress at the moment the actual reconfiguration is conducted, reconfiguration can endanger the system's consistency. By consequence, mutually consistent execution states can only be preserved by means of additional consistency recovery support, which requires software modules to capture and reinstate module specific state at runtime. Inconsistencies are thus allowed during reconfiguration, as long as consistency returns when the reconfiguration is complete. Other systems that implement state transfer as well include the work of Gupta [54], Vandewoude [136, 135], and Kasten [72].

The problem of preserving mutually consistent execution states has also been recognized in the area of dependable real-time systems. The Hercules framework [32] and the Simplex architecture [122, 115] propose a different strategy from the previous ones to preserve consistency. Both frameworks allow for the old and new version of a module to coexist during a reconfiguration. Once the output of the new software module converges with that of the old one according to user provided criteria, mutually consistent execution states are reached. The output of the old module is then turned off and the new module is brought in use. However, testing

for convergence of the output generated by two coexisting components can be very difficult. Besides, there might be no guarantee concerning convergence anyway. We believe that this approach therefore is limited in its ability to be employed in programmable networks.

Finally, we briefly present two other approaches to preserve mutually consistent execution states which are not applicable either in the context of this dissertation. Bloom's dynamic reconfiguration support [21], for instance, makes use of the transactional support that Argus provides – to be precise, Argus guarantees that actions are atomic and that stable state survives crashes. Bloom uses this support to initialize a new software module in the correct execution state by instructing Argus to restore the last stable state of the old software module. Ajmani et al., for their part, provide a different way to preserve consistent execution states of objects from one version to the next [7]. Their approach involves the use of "simulation objects", which implement the behavior of different versions of a specific object by calling methods of the latter. This way, all new versions of a specific object are automatically initialized in the correct execution state.

2.2.2.3 Application state-invariants

A last pre-condition for safe dynamic software reconfiguration relates to the application state-invariants[11]. These invariants define the predicates for a reconfiguration to be legal, each expressed over the state of (a subset of) the entities in the system [100]. A typical example to illustrate this involves the replacement of a module that generates unique ID's. For this replacement to be legal, the new generator should not repeat ID's that were already produced by the old version. To preserve this invariant, the new generator must be initialized in a state which prevents it from producing ID's that were already generated by the old module. Because preserving application state-invariants is beyond the scope of this dissertation, we do not further elaborate this topic.

2.2.3 Dynamic compositional adaptation

The two previous subsections have presented a wide variety of research efforts in the context of dynamic software reconfiguration that each target various concerns. To gain focus, this and subsequent sections aim to clearly position this dissertation in such a wide and active research area. To be precise, in the context of dynamic software reconfiguration this research concentrates on *compositional adaptation* of *pipe-and-filter architectures*.

We start by elaborating on compositional adaptation. In [92], McKinley et al. define compositional adaptation as follows:

[11] Bloom refers to this invariant as a system's "continuation abstraction" [21].

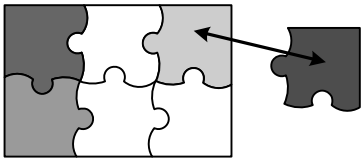


Figure 2.5: Modularized concerns

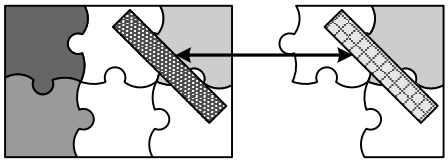


Figure 2.6: Crosscutting concerns

> *Compositional adaptation exchanges algorithmic or structural system components with others that improve a program's fit to its current environment. With compositional adaptation, an application can adopt new algorithms for addressing concerns that were unforeseen during development. This flexibility supports more than simple tuning of program variables or strategy selection. It enables dynamic recomposition of the software during execution – for example, to switch program components in and out of a memory-limited device or to add new behavior to deployed systems.*

This makes compositional adaptation an interesting approach to implement dynamic software reconfiguration of programmable networks[12]. More in detail, compositional adaptation enables programmable networks to deal with both shortcomings listed in Section 2.1.4. First, it permits programmable nodes to dynamically add, replace, or remove network service components to accommodate evolutionary changes. Second, when memory resources on network devices restrict the number of service components that can be deployed simultaneously, compositional adaptation enables to develop adaptive network support that customizes light-weight programmable nodes at runtime on an as-needed basis.

As McKinley et al. stated in [92], compositional adaptation has been enabled by the confluence of three key technologies: *separation of concerns*, *component-based design*, and *computational reflection*. We elaborate on these key technologies in the remainder of this section.

Separation of concerns

As Parnas already stated in the 70's, separation-of-concerns (SoC)[13] is an essential principle for improving a system's flexibility and comprehensibility [106]. In short, SoC promotes to identify and separate different interests in a program. These concerns are the primary criteria for decomposing software into smaller, more manageable, comprehensible, and coherent building blocks [105]. Besides, such decomposition enables to clearly target the software modules representing a specific concern.

[12]Note that compositional adaptation is often referred to as component hot-swapping or dynamic recomposition.

[13]To be correct, Parnas did not use the term "separation-of-concerns", but referred to this principle as "modularization".

So, developers can manipulate these concerns without the need for detailed knowledge of other concerns, and without facing accidental complexities that these other concerns may introduce.

To illustrate the importance of SoC in the context of compositional adaptation, we refer to Figures 2.5 and 2.6. When a software system's functional logic has been decomposed into modular, coherent building blocks that each represent a specific functional concern, changing one of these concerns can be isolated to a single location in the software (as illustrated in Figure 2.5). In addition to their functional logic, however, software systems often incorporate various additional concerns such as concurrency, distribution, real-time constraints, persistence, quality-of-service, testability, logging, and failure recovery. These extra-functional concerns inherently crosscut the system's functional logic, and thus break its modular decomposition. Logging functionality, for instance, is often tangled and scattered with the system's functional logic. This makes it difficult to recompose the functional logic without taking into account the logging support or vice versa (as illustrated in Figure 2.6). The Aspect Oriented Programming (AOP) community has recognized this problem and seeks to modularize these crosscutting concerns and decouple them from the functional logic.

Component-based design

Component-based design is the second key technology supporting compositional adaptation. In short[14], component-based design promotes constructing software systems through the composition of self-contained building blocks (components) which may be developed independently, for instance by third parties. This brings us to Szyperski's definition of a component [124]:

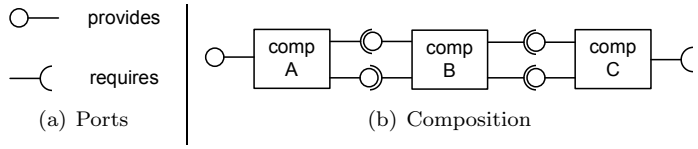
> *A component is a unit of composition with contractually defined interfaces and explicit context dependencies only. Components can be deployed independently, and are subject to composition by third parties.*

According to this definition, components are often represented as coherent software modules[15] that expose a set of contractually defined "ports". These ports are named interfaces and define the operations and events that a component *provides* or that it *requires* from its environment to work properly[16]. This enables third parties to compose software systems from a set of predefined components by (directly or indirectly) connecting the ports of compatible components to each other. To illustrate this, Figure 2.7 depicts a composition of three components that are connected to each other directly. When such a component-based system supports dynamic

[14] For an excellent analysis of component technology, we refer to [127], section 3.1.2

[15] Note that a component may package many objects

[16] Szyperski defines these required operations and events as a component's context dependencies [127].



(a) Ports (b) Composition

Figure 2.7: A composition of three components.

class loading and late binding[17], its composition can be dynamically adapted in a systematic (as opposed to ad hoc) manner.

Constructing programmable network services through the composition of predefined components thus facilitates customizing these services at runtime. Besides its value in the technological realization of compositional adaptation, we argue that component-based development is useful as well to support third party network software development. As already mentioned in the introduction, allowing third party actors to develop future network services enables a free-market for – both open or proprietary – development of new network services and protocols. For being competitive in this new market, network service developers will have to meet new software quality attributes like reusability, maintainability, flexibility, portability, etc. – that is, additional to the traditional concern for scaling up the network's availability and performance. Because component-based software engineering seeks to support developers to meet these quality attributes, it provides a valuable methodology for constructing third party network services whether or not these services are reconfigured dynamically.

Computational reflection

A third enabling technology for compositional adaptation is computational reflection. In [86], Maes defines compositional reflection as follows:

> *Computation reflection is the behavior exhibited by a reflective system, where a reflective system is a computational system which is about itself in a causally connected way.*

According to this definition, computational reflection (further on called reflection) refers to a program's ability to reason about and manipulate itself during the course of its execution, in the same way as it does in relation to its application domain [77]. A reflective system therefore exposes its implementation details at a level of abstraction which hides unnecessary details, but still exposes enough details to allow for *introspection* and *intercession*, – that is, to enable the system to observe its internal behavior and structure, and if needed, to act on these observations by adapting itself. For that reason, reflection has been applied in many research areas, including middleware [78, 20, 77], operating systems [145] and programmable

[17] which enables to connect its components at runtime

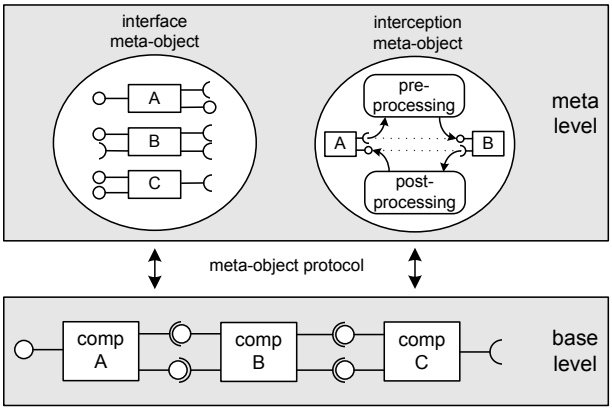


Figure 2.8: Computational reflection.

networks [137, 36], as a useful technique to open up black-box systems and increase their flexibility in a systematic manner.

According to [77], a reflective system is structured in terms of a *base-level* that deals with application specific concerns, and a *meta-level* that controls the reflective behavior (as illustrated in Figure 2.8). At the base-level, the implementation details that are relevant to the reflective behavior are *reified* – that is, they are exposed as programming entities that can be manipulated at runtime. At the meta-level, the reflective behavior is represented by *meta-objects*. To illustrate this, Figure 2.8 depicts two different meta-objects that are also provided by the OpenORB middleware [20]: the interface and the interception meta-object. The interface meta-object enables to inspect the external representation of the software components located at the base-level by exploring the interfaces they provide and require. The interception meta-object allows to manipulate the behavior of the base-level system by performing pre and post-processing of the interactions intercepted at the component interfaces. The interaction protocol these meta-objects employ is referred to as the *meta-object protocol* (MOP). A MOP thus enables a meta-object to observe and adapt base-level components, and therefore serves as the glue between a system's meta and base-level. Note that the base-level and the meta-level are *causally connected*. Changes to either one will be reflected in the other [92].

Finally, reflective behavior can be classified in two different categories: *structural* and *behavioral reflection*. Structural reflection refers to the ability of a system to inspect or change the (reified) structural aspects of the software that is currently operating at the base-level. Structural reflection thus addresses issues related to component types, component interfaces, compositions, object states, data types, etc. An example of structural reflection includes the interface meta-object presented above. Behavioral reflection, in contrast, focuses on the (reified) semantics of the base-level software [92]. An example of this type of reflection includes the

interception meta-object, which enables to change the execution of the underlying program by performing pre and post-processing of intercepted interactions.

To summarize, in the context of dynamic software reconfiguration this dissertation investigates how to coordinate *compositional adaptation* in out-of-band active networks. A first precondition that must be fulfilled, therefore, includes that the affected nodes support compositional adaptation. Besides, this dissertation targets compositional adaptation for a specific type of software architectures, namely *pipe-and-filter (network) software architectures*. This will be explained in more detail in the following subsections.

2.2.4 Pipe-and-filter based (network) software architectures

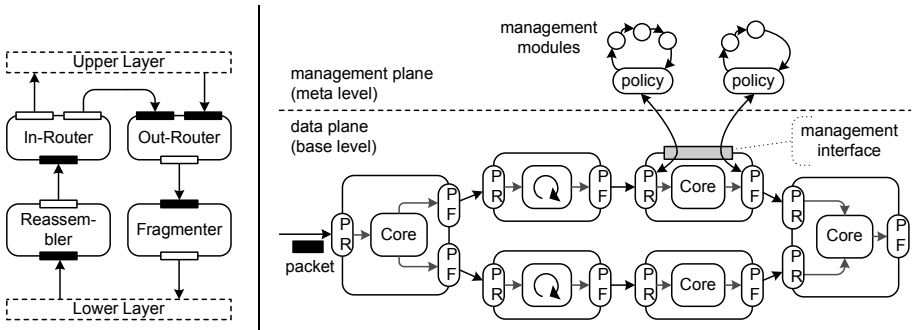
Shaw and Garlan define a pipe-and-filter architecture as follows [123]:

> *In a pipe-and-filter based architecture each component reads streams of data on its inputs and produces streams of data on its outputs, usually while applying a transformation to the input streams and processing them incrementally so that output begins before the input is completely consumed.*

A pipe-and-filter software architecture thus forces a programmer to develop self-contained components (filters) that process incoming data, which are then plugged one after another by means of connectors (pipes) to create a functional system. This style naturally maps to networking software. A protocol stack, for example, comprises various functions (such as fragmentation, routing, header parsing and construction, etc.) that are executed successively on the incoming data packets (both along the up-going and the down-going path).

Shaw and Garlan describe some interesting properties of this pipe-and-filter architectural style [123]. Two of them are highly relevant in the context of this dissertation. First, the pipe-and-filter style supports *reuse*: any two filters can be hooked together, provided that they agree on the data that is transmitted between them. This enables third party network service developers to reuse their service components in different programmable networks. Second, pipe-and-filter systems can be *easily maintained* and *enhanced*: new filters can be added to existing systems, and old filters can be replaced by improved ones. In the context of programmable networks, this enables to build flexible network software architectures.

Because of these properties, various network software architectures have adopted this pipe-and-filter style. These architectures include NetScript [144], Click [75], the protocol stack framework that Lee and Chang presented in [82], VERA [71], and DiPS+ [97, 99], to only name a few. Because the current prototype of the NeCoMan middleware has been developed to operate on top of DiPS+, we briefly present this architecture's characteristics in the next subsection.



(a) DiPS+ router

(b) DiPS+ composition

Figure 2.9: The DistriNet Protocol Stack (DiPS+).

2.2.5 DiPS+ protocol stacks

DiPS+ has been developed at the DistriNet labs to increase flexibility in protocol stack software. This has resulted in a software architecture tailored to build highly reconfigurable protocol stacks, as well as a component framework that enforces this architecture [99]. In the remainder of this section, we briefly discuss how DiPS+ has been prepared for compositional adaptation by implementing the three key enabling technologies listed in Section 2.2.3, which included separation of concerns, component-based design, and computational reflection.

Separation of concerns

Separation of concerns has been one of the key technologies underpinning DiPS+. In short, DiPS+ stacks have been decomposed into self-contained components that each encapsulate a single protocol function, such as packet fragmentation, compression, routing, header parsing and construction, etc. To illustrate this modular decomposition, Figure 2.9(a) depicts the basic functionality of a simple DiPS+ router (as Matthijs presented in [90]). Each of these components has a specific responsibility. The `Out-Router` component, for instance, will determine to which interface outgoing packets should be forwarded. The `In-Router` component, in contrast, decides whether an incoming packet is for local delivery or whether it should be forwarded. Additionally, the `Fragmenter` and the `Reassembler` components encapsulate fragmentation support. This modular router decomposition enables a developer to change fragmentation support without affecting the employed routing functionality, and vice versa.

DiPS+ component model

Conform to the philosophy of component based development, DiPS+ promotes constructing protocol stacks through the composition of fine-grained, self-contained components. In addition, because DiPS+ adopts the pipe-and-filter architectural style, such a composition results from plugging components one after another (as illustrated in Figure 2.9(b)). Once a protocol stack is composed that way, it can process packet flows passing by.

To support this pipe-and-filter based composition, DiPS+ components are developed as a *core* (which encapsulates the component's specific behavior) surrounded by *communication ports*. These ports are implemented as "packet-receivers" (PR) and "packet-forwarders" (PF), which represent a component's entry and exit ports, respectively (see Figure 2.9(b)). Packet-receivers thus accept packets and deliver them to the component's core for being processed. Packet-forwarders, in contrast, deliver packets to the packet-receivers of other components in the flow. As a result, composing multiple components boils down to connecting the involved packet-forwarders and packet-receivers to each other (as illustrated in Figure 2.9(b)).

In addition, all packet-receivers share a *fixed interface*: `incomingPacket(Packet p)`. As Fielding states in [44], restricting the interfaces of components in a pipe-and-filter architecture allows independently developed components to be arranged at will to form new applications[18]. This is a major advantage concerning flexibility, because it allows compositions to be adapted without the need for checking compatibility when connecting components to each other. The router composition depicted in Figure 2.9(a), for instance, can easily be rearranged such that fragmentation will be bypassed, or compression support is added[19].

Finally, when a DiPS+ component encapsulates different processes, it provides separate communication ports for each of these processes. The component in Figure 2.10, for instance, exposes separate communication ports for the compression and decompression process that it encapsulates. This enables to distinguish between packet-flows that are directed to specific component processes (such as packets that are to be compressed and packets that must become decompressed). Note that separate ports for each process that a component encapsulates was not provided by the original DiPS+ architecture (presented in [97, 99]). We extended the DiPS+

[18] Fielding defines such a pipe-and-filter architecture style with the constraint that all filters must have the same interface as a *uniform* pipe-and-filter style [44]

[19] The use of a fixed communication interface, however, limits the interaction between components. Communication between DiPS+ components, for instance, is restricted to forwarding packets towards the next component in the pipeline. As a work-around, DiPS+ components can also communicate with each other indirectly through *blackboard interaction*. In general, the blackboard interaction style is characterized by an indirect way of passing messages from one component to another, using an in-between data source (blackboard) [99]. DiPS+ implements this indirect interaction by enabling components to annotate packets that flow though the pipeline with meta-information. Hence, all "downstream" components (meaning all components these packets will flow through) can retrieve this meta-information independently. The main advantage of this approach is that it preserves the independence of DiPS+ components: components that consume specific meta-information do not have to know the producer of these data (and vice versa) [99].

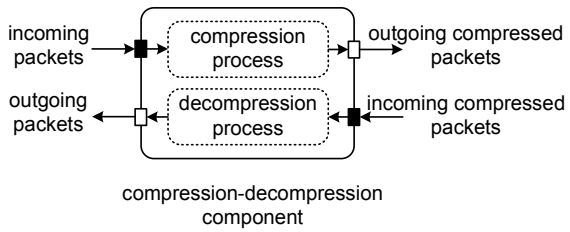


Figure 2.10: Separate communication ports for all processes that a component encapsulates.

component model afterwards to support dynamic reconfiguration more easily.

Reflection

Finally, DiPS+ supports reflection as well besides separation of concerns and component-based design (as illustrated in Figure 2.9(b)). DiPS+ clearly separates base-level functionality from meta-level objects. This has resulted in a "data plane", which contains the regular protocol stack functionality, and a "management plane", which controls among others adaptive load management support [98]. In addition, at the base-level, a component's packet-receivers and packet-forwarders are separated from its specific behavior, and thus offer indirections that can be used to manipulate interactions between components. This enables "policy" objects to intercept packets that enter or leave a component. These intercepted packets are then delegated to a number of pipelined "management modules", which are employed by meta-objects (such as the adaptive load management support) to monitor and adapt the packets flowing through the protocol stack. These policy-objects thus define the MOP that connects the data plane (base-level) and the management plane (meta-level) to each other.

2.3 Dynamic change management

The previous sections have defined how this dissertation relates to programmable networks and dynamic software reconfiguration research. To complete this positioning, this section focuses on dynamic change management, and delimits the scope of our research in this context. As will be clarified in the remainder of this section, this scope encompasses *customizable* change management support.

As already mentioned in the introduction, dynamic software reconfiguration is only beneficial when the reconfiguration process is implemented in an *effective* and *efficient* way, such that the costs and risks it introduces do not outweigh those associated with shutting down and restarting the system. This requires for the reconfiguration process to preserve the correct functioning of the system (effective-

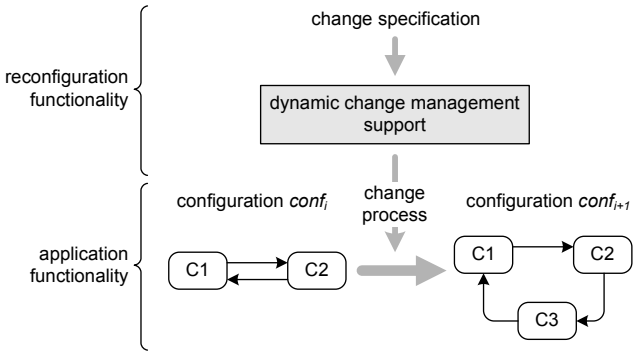


Figure 2.11: Dynamic change management support

ness), and to minimize the overhead it causes (efficiency). When implementing dynamic reconfiguration support, however, developers face several issues that have a direct bearing on both concerns, such as choosing the most appropriate reconfiguration mechanism or the most efficient approach to preserve consistency. This makes the ad hoc implementation of a dynamic reconfiguration often a complex challenge.

Dynamic change management support seeks to reduce the complexity of ad hoc dynamic software reconfiguration in a generic and application independent manner [79, 139, 104, 19, 11, 59]. As illustrated in Figure 2.11, dynamic change management support encapsulates predefined reconfiguration functionality, and clearly separates it from the application that will be reconfigured. This enables to reuse reconfiguration functionality for various applications, on condition that the latter meet some predefined architectural requirements imposed by the reconfiguration functionality.

To achieve its objectives, dynamic change management support must comply with a number of requirements. These requirements (which are derived from the requirements that Kramer and Magee have listed in [79]) relate to the effectiveness, efficiency, and complexity of dynamic software reconfiguration.

- *Change specification should be declarative.* To conceal the complexity inherent in dynamic software reconfiguration, dynamic change management support should only require from its users to specify *what* reconfiguration it must execute, instead of also defining *how* a specific reconfiguration must be executed. As illustrated in Figure 2.11, change management support thus only requires to specify a transfer from the current configuration (conf $_i$) to a resulting configuration (conf $_{i+1}$). Using declarative specifications to define this transition leaves control about the execution of the reconfiguration to the employed change management support [79].

- *Changes should be specified in terms of the software structure.* Dynamic change management support should help developers to identify what must be changed. Because the description of a software structure provides a clear means for both software comprehension and construction, it seems appropriate that software changes should be specified in terms of its structural primitives [79]. A reconfiguration of a component based system, for instance, should be expressed in terms of component creation/deletion and connection/disconnection, since these are the primitives to define a composition.

- *Change specifications should be independent of the algorithms, protocols and states of the application.* To provide generic change management support, the employed reconfiguration process should be independent of the application that will be reconfigured. This separation of concerns enables to reuse change management support for various applications. When the reconfiguration process depends on application specific aspects, for example to check whether or not the application has reached a consistent execution state, these aspects must be abstracted into generic application properties.

- *Changes should leave the system in a consistent state.* To perform reconfigurations effectively, change management support controls the reconfiguration process to prevent it from compromising the correct functioning of the system. After a reconfiguration the system must be able to continue processing normally, rather than progressing towards an error state [79].

- *Changes should be executed efficiently.* Finally, change management support controls the efficiency of the reconfiguration process. As Warren and Sommerville state in [139], reconfiguration efficiency can be decomposed into three distinct parts. First, the time between reconfiguration request and execution should be minimal. Second, the time taken to perform the requested reconfiguration should be minimal. Third, application disturbance with respect to the number of components affected by reconfiguration should be minimal.

Except for the framework of Hillman and Warren [59], existing dynamic change management systems [80, 139, 101, 104, 19, 11, 31] typically conform to the black-box philosophy. In particular, these systems encapsulate a single and fixed reconfiguration algorithm. These systems therefore lack the ability to customize the reconfiguration process such that it may perform better in a specific context, for example by exploiting some service properties or reconfiguration semantics.

We argue that in the context of programmable networks – which by nature have to satisfy stringent performance requirements – this black-box philosophy constrains the efficiency of dynamic software reconfiguration. To limit the overhead that a reconfiguration causes, one must be able to optimize the reconfiguration process whenever possible. To illustrate this, consider the dynamic *replacement* of a compression service that covers two neighboring programmable nodes with a new

version. To preserve the correct functioning of the network during this reconfiguration, the local replacement operations on both nodes must be coordinated[20]. If this is not the case, packets processed by the old compression component may not be decompressed correctly when the neighboring node is already upgraded to use the new decompression component.

When an existing compression service becomes replaced by a new version that is *compatible with the old one*, in contrast, the reconfiguration of both nodes does not have to be coordinated anymore to preserve consistency. To be precise, when both the old and new decompression component can process packets that originate from the old as well as from the new compression component, consistency is always implicitly preserved during the reconfiguration. Both nodes therefore can safely be upgraded independent from each other, thus reducing the overhead a coordinated reconfiguration causes.

This optimization, however, cannot be carried out when the employed change management support conforms to the black-box philosophy. We therefore argue that in the context of programmable networks, change management support must be *customizable* such that the employed reconfiguration algorithm can be tailored to exploit service properties and reconfiguration semantics. From this perspective, we present the NeCoMan middleware as customizable change management support for out-of-band active networks.

2.4 Network service characteristics

The previous sections have presented the scope of this dissertation in the area of programmable networks, dynamic software reconfiguration, and dynamic change management research, respectively. This section elaborates on the characteristics of the network services that we target for dynamic reconfiguration.

Programmable networks have served to develop a wide variety of network services, including application services [141, 12], resource management [29], and network management [144, 120, 43]. These services differ, among others, by the architectural domain they focus on, such as QoS control, management, transport, etc. This dissertation concentrates on network services that target data transport: such services operate on data packets flowing from a source to a sink node. Recall that the services we focus on are targeted at strata 2 and 3 of Coulson's reference architecture – that is, they can define in-band functions or application services[21]. In addition, we support dynamic reconfiguration of both isolated and distributed services. The remainder of this section elaborates on the characteristics of these network services.

[20] that is, to preserve consistency as discussed in Section 2.2.2
[21] See Section 2.1.4

2.4.1 Isolated network services

Isolated network services operate independently on data packets. These services are encapsulated by *self-contained* components, which do not require cooperation with other components. An example of such a self-contained protocol stack component is a filter component to relieve a congested node. An additional example includes a packet-scheduling component that accepts packets through different inports, and stores them in associated buffers. The scheduling process then iterates over these buffers, picks out packets according to the employed scheduling algorithm, and delivers them to the component's single outport.

Besides being self-contained, these isolated service components are also *reactive*. They only react in response to packets that must be processed, and therefore will never autonomously initiate their service execution. Note, however, that these reactive components may still contain active objects to execute their services[22]. The packet-scheduling component, for instance, executes its scheduling behavior in a new thread.

2.4.2 Distributed network services

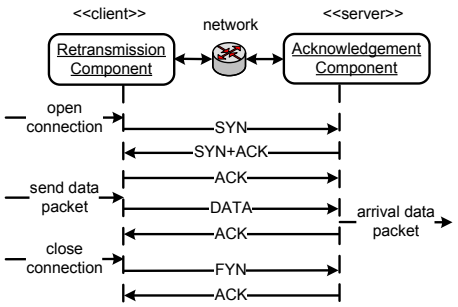
In addition to isolated network services, we also target dynamic reconfiguration of distributed network services. Examples of distributed services that are typically provided by protocol stacks include encoding, compression, fragmentation, reliability, and encryption. These services have the following characteristics in common.

Distributed dependencies

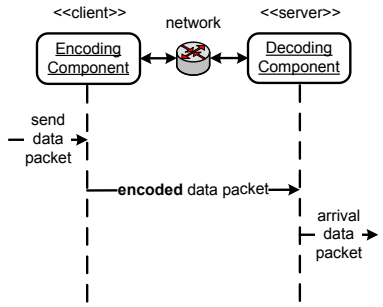
These services are represented by a pair of *tightly-coupled* distributed components (as illustrated in Figure 2.12). Both service components need to collaborate for the service to perform correctly. The distributed dependencies between both collaborating components are formalized by the employed communication protocol, which specifies when a service component invokes its counterpart. To illustrate this, the protocol of a TCP-like reliability service[23] denotes that both service components mutually depend on each other to establish a connection, to transmit data packets reliably and finally to terminate the connection (see Figure 2.12(a)). The protocol that an MPEG software encoding service uses, in contrast, specifies that the encoding component expects from the decoding component to restore encoded data packets into their original state (see Figure 2.12(b)).

[22] An active object owns its own thread to perform certain tasks asynchronously.

[23] Note that this TCP-like service is used only by way of illustration. Its protocol description is not fully RFC-compliant.



(a) Protocol of a TCP-like service



(b) Protocol of an MPEG software encoding service

Figure 2.12: Distributed dependencies formalized by the communication protocol.

Client-server based collaboration

The distributed network services we focus on apply a client-server collaboration model. Corresponding to [13], we define client-server collaboration as follows:

> *The client-server model defines the relationship between two collaborating processes. A client process makes a service request that triggers a reaction from the server process. Stated differently, a client process initiates service activity, while a server process waits for requests to be made and then reacts to them.*

The client process thus initiates the execution of the employed communication protocol, while the server process reacts to client requests according to this protocol. This collaboration model applies to many distributed network services, such as for instance the TCP-like service and the MPEG software encoding service (see Figure 2.12). The retransmission and encoding components of these services (hosting the client processes) invoke the acknowledgement and decoding components (providing the server processes), respectively, to initiate service activity. Note that both collaborating service components can accommodate client and server processes simultaneously. For instance, when an MPEG software encoding service covers a bidirectional communication link, both service components employ a server process to decode encoded data packets arriving from the network, as well as a client process to encode data packets that will be transmitted to the decoding node.

Reactive behavior

Similar to isolated service components, all distributed service components are reactive – that is, they only react in response to external service requests. Since a server process waits for requests to be made and then reacts to them, this process is inherently reactive. In addition, the client processes of the services we focus

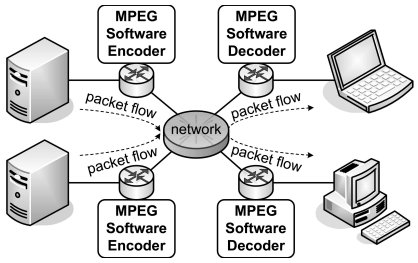


Figure 2.13: Many-to-many deployment model for distributed network services.

on are reactive as well. They will never autonomously initiate the execution of a communication protocol for a purpose other than processing external requests (for example to transmit encoded data packets). Note that although all service components are reactive, they may still contain active objects. As an example, we refer to a timer object that initiates packet resends in response to external requests for reliable transmission.

Asynchronous buffered communication

Both collaborating service components communicate by asynchronous buffered message passing. These components invoke each other by sending packets over the network asynchronously – that is, without delaying their activity until the packets they transmitted have arrived. The network capacity hence conceptually represents a shared buffer that is used by both collaborating service components to communicate.

Many-to-many service composition

Although we deal with distributed services that conceptually consist of two collaborating components, multiple instances of both service components might have to be deployed to use this service correctly in a concrete network setup. For instance, suppose an MPEG software encoding service is employed to accommodate a slow wireless sub-net in a programmable home network, as illustrated in Figure 2.13. To maintain correct network operation, all packets that enter and leave this sub-net must be encoded and decoded, respectively. This involves deploying multiple encoding and decoding components, all of them providing part of the same service. When the service is used to cover a unidirectional communication link between two neighboring nodes, in contrast, one single encoding and decoding component are sufficient.

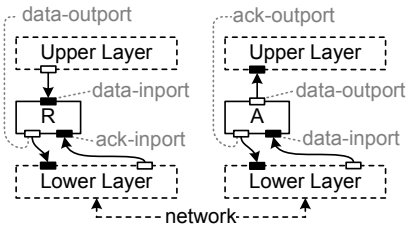


Figure 2.14: Protocol stack composition of two nodes hosting a reliability service. The black filled rectangles graphically symbolize the components' inports. The small empty rectangles, in contrast, correspond to the components' outport.

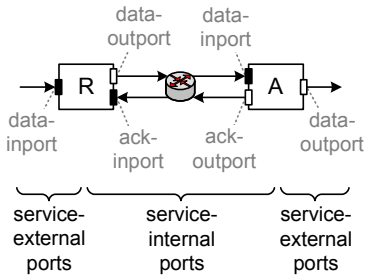


Figure 2.15: Service-internal and service-external ports of both reliability components.

2.5 Service-external and service-internal communication ports

When distributed network services are involved, we refine the employed component model by differentiating between service-external and service-internal communication ports[24]. To explain this in more detail, we first present the composition of a (distributed) reliability service. Note that this service will also be used through the remainder of this text to illustrate various aspects of dynamic software reconfiguration.

As depicted in Figure 2.14, this reliability service consists of a retransmission component R that collaborates with an acknowledgement component A. When a data-packet must be transmitted reliably, it becomes delivered to the data-inport of R. Component R then stores a copy of this data-packet in its retransmission queue, attaches a sequence number to the packet and starts counting down to retransmission. Next, R delivers this packet (and all subsequent resends) to the lower layer of the protocol stack via its data-outport. When this packet arrives at the destination node, the lower layer of that node dispatches it to the data-inport of A. If the packet arrived correctly and is not a duplicate resent packet, A delivers it to the upper layer via its data-outport and always returns an acknowledgement packet to the sending node. This ack-packet reaches the lower layer via A's ack-outport. Once the ack-packet reaches the sending node, it is delivered to the ack-inport of R. In response to this, R removes the acknowledged data-packet from its retransmission queue.

Now let us focus on the communication ports. According to our component model, client and server processes use *service-internal* ports to exchange packets

[24] Again this was not provided by the original DiPS+ architecture.

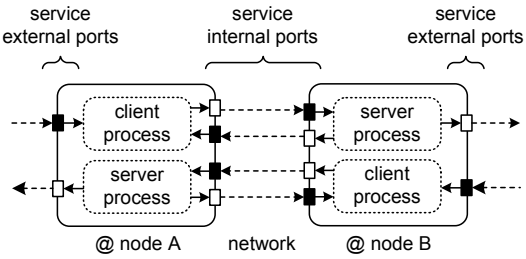


Figure 2.16: Two collaborating components, each encapsulating a client and a server process.

during the execution of the employed communication protocol. Examples of such server-internal ports include the data-outport and ack-inport of R, and the data-inport and ack-outport that component A provides. These ports are connected indirectly (that is, over the network as illustrated in Figure 2.15) and are used by the retransmission and acknowledgement process to exchange data and ack-packets. *Service-external* ports, in contrast, connect the associated processes with other protocol stack components that do not participate in the execution of communication protocols. Examples of such ports include R's data-inport and A's data-outport.

Note that a client process does not use service-external outports, and a server process does not provide service-external inports (as illustrated in Figure 2.16). This is because (according to the applied collaboration model) a client process invokes only a collaborating server process (so as to execute a communication protocol). Since client processes do not invoke other processes, they do not expose service-external outports. In addition, a server process reacts only to invocations that are sent by its client process and not to requests initiated by other components. Hence, server processes do not employ service-external inports. Besides, note also that client and server processes on the same node will never collaborate with each other, but only with processes located on other nodes.

Finally, note that the employed component model does not distinguish between service-external and service-internal ports for components that encapsulate isolated network services. Because these components are self-contained, they contain only processes that do not collaborate with other ones to complete a service. Hence, the distinction between client and server processes, as well as between service-external and service-internal communication ports becomes irrelevant.

2.6 Requirements and motivation

To conclude this chapter, we bring together the research topics that have been discussed in the previous sections, and deduce the main objective that has driven the research presented in this dissertation. This objective encompasses the development

of:

customizable change management support
for
dynamic compositional adaptation
of
local and distributed network services
in
pipe-and-filter based out-of-band active networks

This objective has resulted in the development of the NeCoMan (*Ne*twork re*Co*nfiguration *Man*agement) middleware, which operates on top of pipe-and-filter based out-of-band active nodes[25], and conducts local and distributed recompositions of these nodes' software architecture. The novelty of this middleware is in its ability to customize the reconfiguration process, taking into account both the characteristics of the network services that are involved and the reconfiguration semantics. By packaging this reconfiguration complexity in a middleware, we aim to make efficient and effective dynamic reconfiguration of programmable networks less complex and error prone. In the remainder of this subsection, we list four requirements (relating to the effectiveness, efficiency, and complexity of dynamic software reconfiguration) that NeCoMan must fulfill to meet its objective. As we clarify later on, these requirements have an effect on NeCoMan's reconfiguration algorithm as well as on its design, and therefore will run as a thread through the next chapters.

2.6.1 Correct reconfigurations

First and most important, the NeCoMan middleware must prevent a reconfiguration from compromising the correct functioning of the network and its services. A dynamic reconfiguration that causes failures or inconsistencies can be more harmful to the network than accomplishing the reconfiguration off-line. NeCoMan therefore has to ensure the network's consistency in the course of a reconfiguration. As discussed in Section 2.2.2, this involves (1) reaching *mutually consistent execution states* among the service components and the rest of network, such that after a reconfiguration the network can continue operating normally rather than progressing towards an error state, and (2) providing a correct service composition at each moment in time during the reconfiguration process to preserve *structural integrity*.

2.6.2 Limited reconfiguration overhead

The end-users of programmable networks are mainly concerned with the performance and the availability of the network, especially when using multi-media or

[25] Although the current prototype has been validated on top of DiPS+, NeCoMan seeks to enable dynamic reconfiguration of various pipe-and-filter node architectures, including the ones listed in Section 2.2.4

(soft) real-time applications. These applications impose strict quality requirements on the communication network that they employ, so as to guarantee a certain well-defined level of performance. To meet these quality requirements, the overhead caused when recomposing programmable nodes must be kept to a minimum. Consequently, NeCoMan must enable to execute dynamic reconfigurations as transparently as possible. At best, the end-users should not be able to detect that a reconfiguration has occurred inside the network – that is, they should not experience unexpected delays or service disruptions.

2.6.3 Limited openness

As already stated in Section 2.3, we argue that (in the context of programmable networks) dynamic change management support must be customizable such that each reconfiguration can be tailored to exploit service properties and reconfiguration semantics. This requires to open up change management support such that the reconfiguration algorithm can be adapted. Because implementing a safe reconfiguration algorithm that causes minimal reconfiguration overhead can be very complex and error prone, however, we strongly believe that this "openness" should be limited. Besides the specification of the structural changes that must be executed, a reconfiguration description should only contain a declarative specification of the service characteristics and reconfiguration semantics. Based on these specifications, the NeCoMan middleware should be able to carry out a tailored reconfiguration.

2.6.4 Reusability

Finally, because change management support seeks to promote reusability, NeCoMan should be decoupled from the execution environment of the programmable nodes that will be reconfigured. As we further discuss when comparing related research in Chapter 8, other frameworks and architectures that conduct dynamic software reconfiguration in programmable networks are tightly coupled to a specific node execution environment. The architecture that Chen and his colleagues presented in [30], for instance, is tightly coupled to Cactus [143], a framework for constructing adaptive protocol stacks. This makes it far from trivial to employ this architecture on top of other protocol stack frameworks, such as Click [75], Netkit [36], DiPS+ [97, 99], and the framework of Lee and Chang [82],

2.7 Detailed overview

As illustrated in Figure 2.17, the remainder of this dissertation is structured as follows. Chapters 3 and 4 target local reconfigurations, which involve the addition, replacement, and removal of service components that reside on one node. These components may belong to isolated as well as to distributed services. Chapters 5

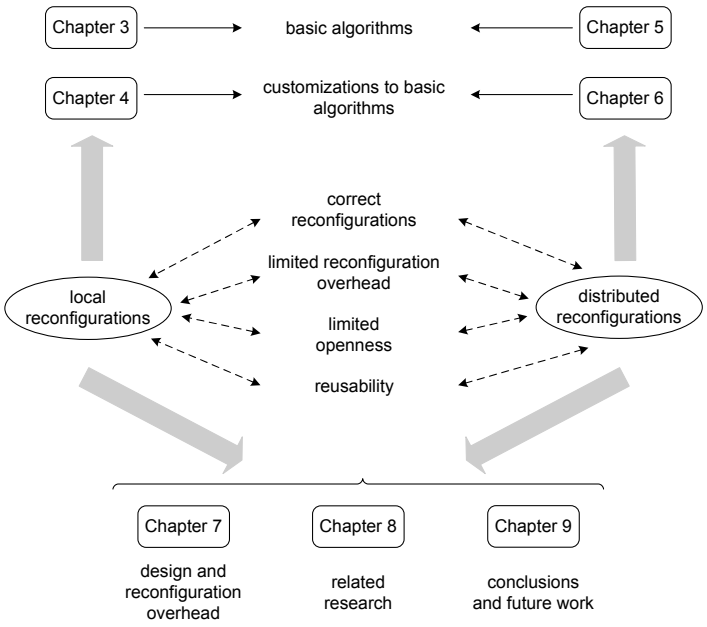


Figure 2.17: Overview of the next chapters.

and 6 then focus on reconfigurations where the unit of adaptation is a distributed network service rather than a local protocol stack adaptation.

More in detail, Chapters 3 and 5 each present two basic reconfiguration algorithms that NeCoMan uses for conducting local and distributed reconfigurations, respectively. These basic reconfiguration algorithms seek to reconfigure an extensive set of services, and thus serve the same purpose as the single algorithm that a black-box reconfiguration system typically employs. Besides, NeCoMan also incorporates a set of customizations to these local and distributed reconfiguration algorithms, which are discussed in Chapters 4 and 6, respectively. These customizations optimize and tailor the employed reconfiguration algorithm by switching the order in which some reconfiguration actions are executed, and by discarding actions that are redundant for particular reconfigurations.

Chapter 7 then presents the design of the NeCoMan middleware and evaluates the reconfiguration overhead that NeCoMan brings about. Next, Chapter 8 situates our approach to related research in the area of programmable networking. Finally, Chapter 9 summarizes the main achievements presented in this dissertation, and identifies future research tracks that spin off from this research.

Chapter 3

Local reconfigurations

In this chapter we elaborate on local recompositions in pipe-and-filter node architectures. Depending on the affected service, these recompositions may involve components that encapsulate isolated network services, as well as components that belong to a distributed network service. The remainder of this chapter explains how NeCoMan adds, replaces, or removes these different network service components, taking into account the four requirements that it must fulfill to achieve its objectives. As we explained in detail in Section 2.6.1, these requirements include consistency preservation, limited reconfiguration overhead, limited openness, and reusability.

This chapter is subdivided into two parts. The first part seeks to prepare the reader to clearly understand NeCoMan's reconfiguration algorithms. Because consistency preservation is a very important requirement for NeCoMan to satisfy, Sections 3.1 and 3.2 specify how to preserve structural integrity and mutually consistent execution states when performing local recompositions in pipe-and-filter node architectures. Next, Section 3.3 describes the functionality that a node architecture must provide to assist NeCoMan in carrying out a reconfiguration. Section 3.4 then presents a pseudo-formal notation to denote all "reconfiguration conditions" that a recomposition middleware like NeCoMan must fulfill to perform a correct reconfiguration. These reconfiguration conditions define the (partial) order in which reconfiguration actions must be executed.

The second part of this chapter focuses on NeCoMan's two local reconfiguration algorithms. The first one of these algorithms conducts recompositions that involve components of a *distributed* network service. Section 3.5 presents this algorithm and indicates that it fulfills all required reconfiguration conditions. After that, Section 3.6 explains NeCoMan's second local reconfiguration algorithm. In contrast to the first algorithm, this second algorithm conducts the addition, replacement, and removal of *isolated* network service components. Section 3.6 also indicates that this second algorithm meets all required reconfiguration conditions as well.

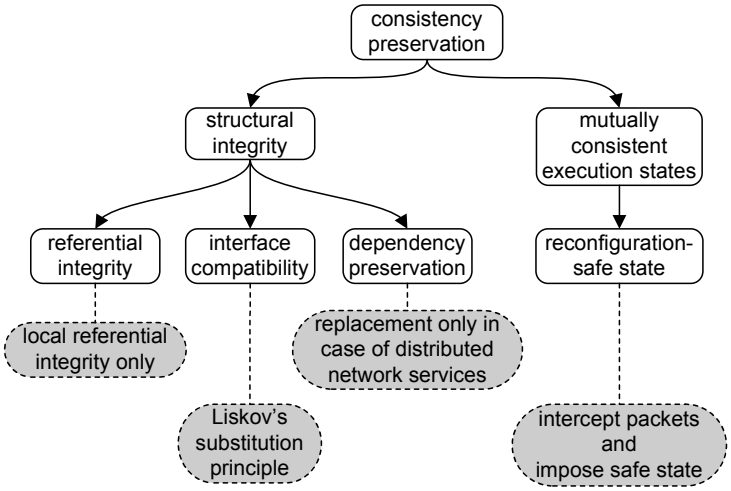


Figure 3.1: Consistency preservation when executing local recompositions in (uniform) pipe-and-filter based network architectures

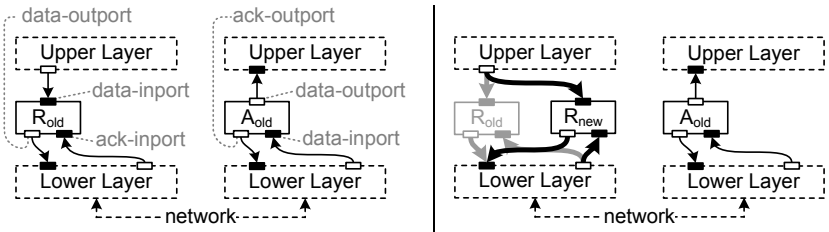
Finally, Section 3.7 concludes this chapter by checking both algorithms against the four requirements that NeCoMan must fulfill to achieve its objectives.

3.1 Structural integrity

For a dynamic reconfiguration of programmable nodes to be effective, consistency must be preserved such that after a reconfiguration the network can continue operating normally rather than progressing towards an error state. As discussed in the previous chapter, this involves preserving both *structural integrity* and *mutually consistent execution states* (see Figure 3.1). This section elaborates on the first of both requirements, and discusses how to maintain *referential integrity*, *interface compatibility*, and *dependency preservation* when recomposing pipe-and-filter based node architectures. We illustrate all this with the replacement of the old retransmission component R_{old} by a new version R_{new}. Figure 3.2 sketches the protocol stack composition of two nodes hosting the reliability service both before and after carrying out this reconfiguration. Finally, note that Table 3.1 summarizes this section as backing.

Referential integrity

A first requirement to preserve structural integrity includes maintaining referential integrity (as depicted in Figure 3.1). After recomposing a programmable node, all affected references must be redirected consistently such that (1) no references are



(a) Original composition (b) Replacement completed

Figure 3.2: Protocol stack composition of two nodes hosting a reliability service, both before and after replacing the retransmission component.

broken and (2) only the new composition will be invoked to process service requests. Conducting a correct compositional adaptation thus requires to unbind and bind all affected components in an atomic manner.

To illustrate this, replacing R_{old} with a new version R_{new} involves reconnecting all affected ports such that packets which were originally directed to R_{old}'s inports now become delivered to the inports of R_{new}. Besides, the outports of both components must be (re)connected as well to prevent breaking a correct composition. Figure 3.2(b) depicts these linking and unlinking operations by black bold and gray bold connections, respectively. To preserve structural integrity, these reconnections must be executed atomically.

Note that only local references must be controlled explicitly when recomposing pipe-and-filter node architectures. This is because the client and server processes of a distributed network service do not reference each other directly. Instead, packets processed by a client process (such as the retransmission process) reach the component that encapsulates the collaborating server process (the acknowledgement process) indirectly through the routing support that the network and the node architectures provide. A local recomposition that involves components of a distributed service therefore only has to preserve local referential integrity. Maintaining distributed referential integrity must be covered at all times by the employed routing support.

Interface compatibility

Besides referential integrity, interface compatibility must be dealt with as well to preserve a network's structural integrity. According to Liskov's substitution principle, when replacing a component with a new version the interface definition of the latter must satisfy that of the original component. Replacing R_{old} with a new version that exposes an incompatible data-inport, for instance, obviously makes it impossible to correctly recompose the node hosting this component. Hence, it is the network administrator's responsibility to make sure that interface compatibility can be preserved.

	isolated service components	**distributed service components**
referential integrity	rebind in and outports in an atomic manner	
interface compatibility	new components must offer (at least) all inports that the old ones provide	
distributed dependencies	N/A	1. component replacement only 2. new component must implement same communication protocol

Table 3.1: An overview on how to maintain referential integrity, interface compatibility, and distributed dependencies when recomposing uniform pipe-and-filter based node architectures.

Note that this requirement also applies to recompositions that involve adding or removing components from a pipeline (instead of replacing a component by a new version). In that case, the interface of the old and new neighboring components must be compatible.

Dependency preservation

Additional to maintaining interface compatibility and referential integrity, all distributed dependencies between collaborating service components must be preserved as well. These dependencies are formalized by the employed communication protocol, and exercise restraints on the recomposition operation that can be carried out. To be precise, a local recomposition that involves components of a distributed service is restricted to component *replacement* only. Besides, the new component must implement the *same communication protocol* to protect a recomposition from jeopardizing the cooperation with other service components that have not changed and thus still expect the old component to process their invocations. Removing A_{old} or replacing R_{old} with a new version that implements a different communication protocol, for instance, obviously breaks the distributed dependencies between both tightly-coupled components.

Restricting the recomposition operation to component replacement only is not required when isolated network services are involved, such as a packet filter. Because these services are self-contained, evidently no distributed dependencies have to be preserved. Components that encapsulate such isolated services therefore can be added, replaced, or removed from programmable nodes without taking into account distributed cooperation with other components.

3.2 Mutually consistent execution states

A second requirement for safe dynamic reconfiguration of programmable nodes relates to the execution state of the software components that will be recomposed. After completing a recomposition, all network service components must be left in a state that allows them to operate normally as if no reconfiguration had occurred. This requires all affected components to reach a consistent execution state before the actual recomposition is executed, and to preserve this state until the recomposition has completed. If this is not the case, a reconfiguration can break the invariant of the affected network service[1]. This, in turn, compromises the correct functioning of the applications that (indirectly) rely on this network service.

To illustrate this, consider replacing R_{old} without taking into account its execution state. This clearly jeopardizes the correct functioning of the reliability service. If R_{old} becomes replaced with R_{new} while its retransmission queue still contains packets, these packets get lost although there is no guarantee that they all have reached their destination correctly. Hence, the invariant of the reliability service (which defines that all transmitted packets certainly reach their destination) becomes broken. This compromises the correct functioning of client applications which assume that every packet will be transmitted correctly.

Besides, when replacing R_{old} without taking into account its execution state, inconsistencies may also arise with respect to the sequence numbers that R_{old} attaches to packets before sending them to A_{old}. To filter out duplicate resends, the acknowledgement component discards packets when their sequence number is below the expected value[2]. The packets that R_{new} sends, therefore, must have the same sequence number as if they were transmitted by R_{old}. If R_{new} starts counting again from 0, for instance, A_{old} will interpret all newly arrived packets as duplicate resends and discard them.

In general, to prevent a reconfiguration from leaving a software system in an inconsistent execution state, most reconfiguration systems prescribe that a recomposition can only be carried out when the software system is in a *reconfiguration-safe state* (or shortly, a safe state) [11]. In the literature various approaches have been presented to reach such a safe state. The remainder of this subsection describes three of these approaches, which we adopted to drive network service components to a reconfiguration-safe state.

3.2.1 Processing all accepted requests

In [11], Almeida et al. define that a software system is in a reconfiguration-safe state when each affected component

1. is currently not involved in servicing accepted requests, and

[1] This invariant defines the conditions that always have to be fulfilled for the service to function correctly.

[2] to be precise, when *seqnr* < (*expected seqnb*) mod (*size of retransmission queue*)

2. will not be involved in servicing new requests until after completing the reconfiguration.

Reaching a safe state thus involves (1) intercepting all requests that are directed towards the affected components and (2) monitoring these components until they have processed all accepted requests.

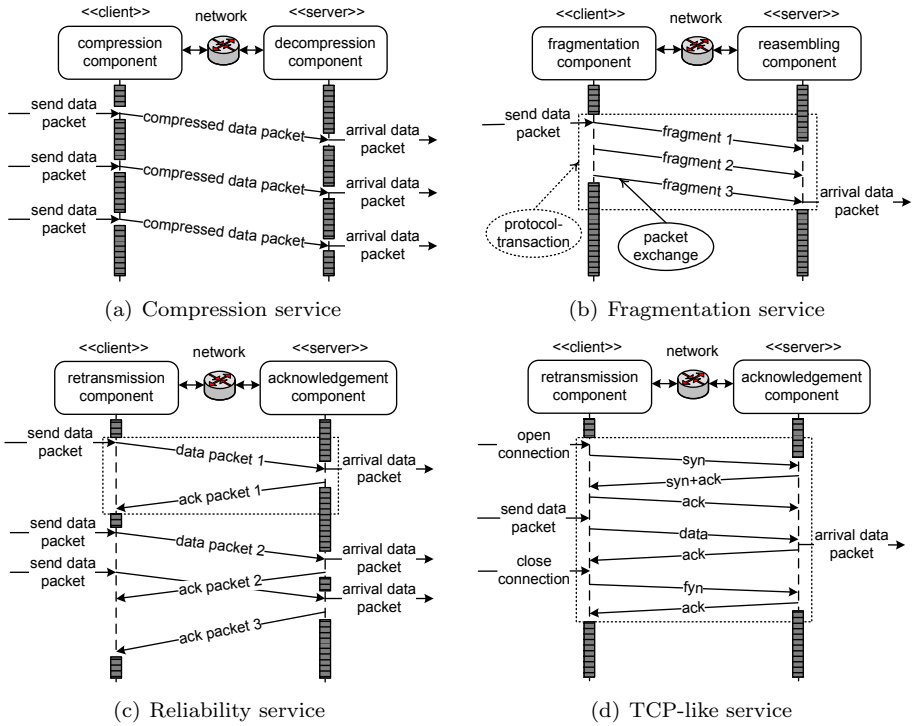
This definition only applies to components which are always in a reconfiguration-safe state once they are not processing requests. In the context of network software, this is the case for components that encapsulate isolated network services. Because these components are self-contained, they do not include results of a partially completed service after processing a single packet. A packet-scheduling component, for instance, reaches a safe state when it has scheduled all buffered packets and is prevented from receiving new packets during the reconfiguration. After this occurs, the affected scheduling component can be replaced or removed without causing information or packet loss, thus leaving the rest of the network software in a consistent execution state.

Besides isolated network components, also a number of components that implement distributed network services reach a safe state once they have processed all accepted packets. As an example we refer to both components of a compression service, illustrated in Figure 3.3(a). These components do not include results of a partially completed compression or decompression activity after processing a single packet. Consequently, they can be replaced safely once they have processed all accepted packets and new packets are prevented from invoking them.

3.2.2 Completing all accepted transactions

Network service components that need to exchange *various requests* to complete a specific service are not necessarily in a safe state while not processing requests. These components typically communicate according to a specific communication protocol that formalizes a sequence of packet exchanges. During the execution of a protocol-transaction[3], every collaborating service component includes results of a partially completed service. These intermediate results will get lost, however, when such a stateful component becomes replaced while it is still participating in ongoing protocol-transactions. Simply preventing these components from being invoked and monitoring them until they have processed all accepted packets therefore does not suffice to reach a safe state. To illustrate this, replacing a retransmission component when it has not yet received an acknowledgement for each packet in its retransmission queue breaks its service-invariant. As an additional example, replacing a defragmentation component when it has not yet received all fragments (and therefore has not yet reassembled the original packet) will cause packet loss.

[3]As its name suggests, a "protocol-transaction" represents the execution of a communication protocol. A protocol-transaction thus includes a predefined sequence of asynchronous packet exchanges between collaborating service components. This is illustrated in Figures 3.3(b), 3.3(c) and 3.3(d).



(a) Compression service

(b) Fragmentation service

(c) Reliability service

(d) TCP-like service

Figure 3.3: These examples illustrate when collaborating network service components reach a reconfiguration-safe state. The latter is depicted by gray shaded blocks.

To prevent the replacement of such stateful components from jeopardizing the correct functioning of the complete system, Goudarzi and Kramer have stated in [101] that a reconfiguration-safe state is reached when the affected entities

1. are currently not involved in a transaction, and

2. will not participate in any transaction until after the reconfiguration actions have terminated.

According to this definition, both the retransmission and the defragmentation component reach a reconfiguration-safe state once they are not participating anymore in ongoing protocol-transactions. For the defragmentation component this implies that each fragment of the packet that is currently being reassembled has arrived, and that the restored packet is delivered to the next component in the pipeline (as illustrated in Figure 3.3(b)). The retransmission component, in contrast, is not participating in ongoing protocol-transactions when each transmitted packet has been

acknowledged (see Figure 3.3(c)). Consequently, monitoring support is needed to check the execution state of these stateful components.

3.2.3 State transfer

Driving a stateful component to a reconfiguration-safe state by waiting until all accepted packets are processed (as Almeida proposes) or by waiting until the affected component does not participate anymore in ongoing protocol-transactions (as proposed by Goudarzi and Kramer) can be very time consuming. If a packet-scheduling component contains large buffers that are filled with packets, then waiting for all these packets to get scheduled significantly delays the reconfiguration. The same holds for replacing a retransmission component. If R_{old} contains a large retransmission queue that is filled with unacknowledged packets, then waiting for the arrival of every acknowledgement may have a major impact on reconfiguration efficiency.

As an alternative for monitoring until a consistent execution state comes about, Hofmeister has proposed to deactivate a software component immediately [61]. Because this implies that information may get lost during the recomposition, additional consistency recovery support is needed. This involves transferring the old component's execution state towards the new version. Consequently, a reconfiguration-safe state is then reached once

1. the affected component is deactivated, and
2. the affected component will not be involved in servicing new requests until after completing the reconfiguration, and
3. the old component's execution state is captured and reinstated in the new component.

For the packet-scheduling service this includes transferring all packets from the old component's buffers to the buffers of the new version. When replacing R_{old}, consistency is restored when all unacknowledged packets as well as the sequence number that is last used have been reinstated in R_{new}.

3.3 Reconfiguration support for DiPS+

How to drive a network service component to a reconfiguration-safe state depends on the semantics of that component. NeCoMan therefore does not enforce a safe state by itself but instead expects the affected node to accomplish this.

In general, to support reuse we believe that a reconfiguration middleware must be decoupled from all functionality that is specific to the network service or node architecture. To be precise, a reconfiguration middleware should not contain node-specific support to change a composition or to control the execution state of the affected components. Instead, a reconfiguration middleware should only coordinate

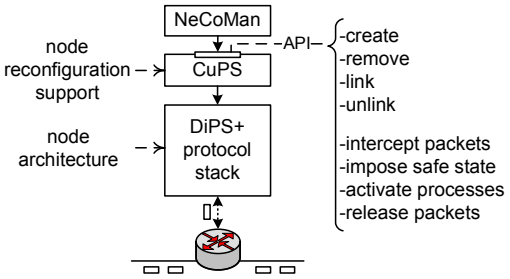


Figure 3.4: CuPS: node reconfiguration support for DiPS+

the execution of a reconfiguration, with this instructing the underlying node to carry out these node-specific operations. Each node therefore has to provide dedicated "reconfiguration support" to assist a reconfiguration middleware in carrying out a reconfiguration.

The DiPS+ protocol stack initially did not have such reconfiguration support at its disposal. We therefore developed the CuPS (Customizable Protocol Stacks) platform to assist NeCoMan in dynamically recomposing DiPS+ protocol stacks (as illustrated in Figure 3.4). CuPS has been published in [68, 69, 67, 99]. To clearly understand the reconfiguration algorithms that NeCoMan employs, we briefly elaborate on the most relevant aspects of CuPS in the remainder of this section. To be precise, Subsection 3.3.1 first presents an overview of the operations that CuPS provides. Next, Subsections 3.3.2, 3.3.3, and 3.3.4 elaborate on two of these operations which serve to obtain a reconfiguration-safe state.

3.3.1 An overview of all reconfiguration operations

To promote reuse, the dependencies between NeCoMan and the underlying node-specific reconfiguration support must be minimal. The latter therefore should only provide a limited API. At the same time, however, this API must offer sufficient operations to assist a reconfiguration middleware in (1) changing a composition and (2) controlling a component's execution state. As illustrated in Figure 3.4, CuPS provides eight operations to accomplish this.

Four operations to change a composition

Four of these operations serve to change a DiPS+ protocol stack composition. As Kramer and Magee proposed in [80], changes to a composition should be expressed (declaratively) in terms of its structure. CuPS therefore offers a *create*, *remove*, *link*, and *unlink* operation to assist a reconfiguration middleware in recomposing a DiPS+ stack.

- **Create.** When a reconfiguration middleware invokes this operation, CuPS

dynamically loads the specified component into the node's protocol stack composition – that is, without binding its communication ports. As part of this operation, CuPS also registers this new service component so as to control its execution state later on during the reconfiguration.

- **Remove.** This operation accomplishes the opposite of what the create operation achieves. It deletes a disconnected component from the node's protocol stack composition, and deregisters this component from support that controls its execution state.

- **Link.** This operation is responsible for binding a specific outport of component A to a specified inport of another component B. When this operation is called, CuPS registers the affected packet-receiver of component B to the appropriate packet-forwarder of component A. This way both communication ports become connected.

- **Unlink.** This operation disconnects a specified outport of component A from an inport of another component B to which it is currently connected. This involves removing the reference stored in component A's packet-forwarder.

Four operations to control a component's execution state

Besides assisting a reconfiguration middleware in changing a composition, a node's reconfiguration support must also enable to control a component's execution state. CuPS therefore offers the following four additional operations: *intercept packets*, *impose safe state*, *activate process*, and *release packets*.

- **Intercept packets.** Each method presented in the previous section to reach a safe state requires for intercepting packets – that is, to prevent these packets from invoking the affected component. CuPS provides this functionality by the "intercept packets"-operation.

- **Impose safe state.** Besides intercepting packets, additional support is needed to drive a component to a safe state. This includes, among others, monitoring the affected component until a safe state comes about and/or transferring this component's state-information towards its new counterpart. This support is offered by the "impose safe state"-operation.

- **Activate processes.** To activate a new component, all its active objects (if any) must be started. The "activate processes"-operation assists a reconfiguration middleware in accomplishing this.

- **Release packets.** Finally, once a recomposition has completed, all intercepted packets must be resumed. This is covered by the "release packets"-operation.

	intercept packets	**impose safe state**
processing all accepted packets	intercept packets directed to all inports	1. monitor until all packets are processed (2. stop active objects) (3. transfer remaining state)
completing accepted protocol-transactions	if *client process*: intercept packets directed to service-external inports if *server process*: intercept packets directed to all inports	1. resume intercepted packets and monitor processes until not engaged in ongoing protocol-transactions (2. stop active objects) (3. transfer remaining state)
state transfer	intercept packets directed to all inports	(1. stop active objects) 2. monitor until idle 3. transfer execution state (4. transfer remaining state)

Table 3.2: An overview of both operations that CuPS provides to intercept packets and to impose a safe state. Depending on the adopted approach to reach a safe state, the implementation of both operations differs.

Depending on the adopted method to reach a safe state, CuPS implements both operations to *intercept packets* and *impose a safe state* in a different way. To clearly understand the algorithms that NeCoMan employs, we elaborate on these two operations in the remainder of this section. Subsection 3.3.2 first discusses the implementation of both operations when a component reaches a reconfiguration-safe state once it is not processing packets and is prevented from accepting new ones. Subsection 3.3.3 then describes both operations when the affected component must be monitored until it is not participating anymore in ongoing protocol-transactions. Finally, Subsection 3.3.4 elaborates on the behavior of both operations when CuPS deactivates the affected component immediately and recovers consistency by transferring its execution state. Table 3.2 presents an overview of these different implementations as backing.

3.3.2 Processing all accepted packets

In the most simple case, a component reaches a reconfiguration-safe state once it is not processing packets and is prevented from accepting new packets. To bring about this state, all packets that are directed towards the affected component must be intercepted first. As discussed above, this is covered by the *intercept packets*-operation that CuPS provides. Once packets are intercepted, a reconfiguration middleware can instruct CuPS to *impose a safe state*, which involves

1. monitoring this component until all accepted packets are processed,
2. stopping the activity of active objects (such as timers) once all accepted packets are processed, and
3. transferring remaining state-information from the old towards the new component.

After that, the affected component can safely be removed without leaving the network in an inconsistent state. We explain in the remainder of this subsection how CuPS implements this scenario. As backing, Figure 3.5 models this scenario.

Intercept packets

When instructing CuPS to intercept packets, its "packet flow controller" holds up packets at every packet-forwarder (outport) that is connected to a packet-receiver (inport) of the affected component. To illustrate this, we refer to Figure 3.5. As depicted in this model, CuPS blocks both all outports of component A and one outport of component B to intercept packets directed to the packet-scheduling component. This approach for intercepting packets has two important advantages.

First, disturbance related to component deactivation during a reconfiguration can be limited. Instead of deactivating all components that directly or indirectly invoke the affected component (as proposed in [79]), CuPS only holds up packets at the outports of neighboring components. To be precise, only packet-flows that are directed towards the affected component become intercepted. This way, packet flows that neighboring components send out through other outports are not (unnecessarily) blocked. As illustrated in Figure 3.5, packets sent directly from component B to component C are not intercepted.

Second, holding up packets at a component's communication ports promotes separation of concerns. That is, a component's functional behavior is decoupled from support to intercept packets. Changing the way of holding up packets at the outports does not interfere with existing component functionality and vice versa. To illustrate this, CuPS supports to intercept packets by blocking the execution thread in which an outgoing packet is executed. As illustrated in Figure 3.5, this functionality is provided by CuPS' "thread blocking support". As an alternative, CuPS' "packet queueing support" intercepts packets by queueing all outgoing packets without interrupting their execution thread. Switching between both strategies only involves adapting the policy objects of the affected outports.

Impose safe state: monitor until all packets are processed

Once all packets directed towards the affected component are intercepted, CuPS can be instructed to impose a safe state. As a first step to accomplish this, CuPS's "activity monitor" observes the affected component until it has processed all accepted packets (see Figure 3.5).

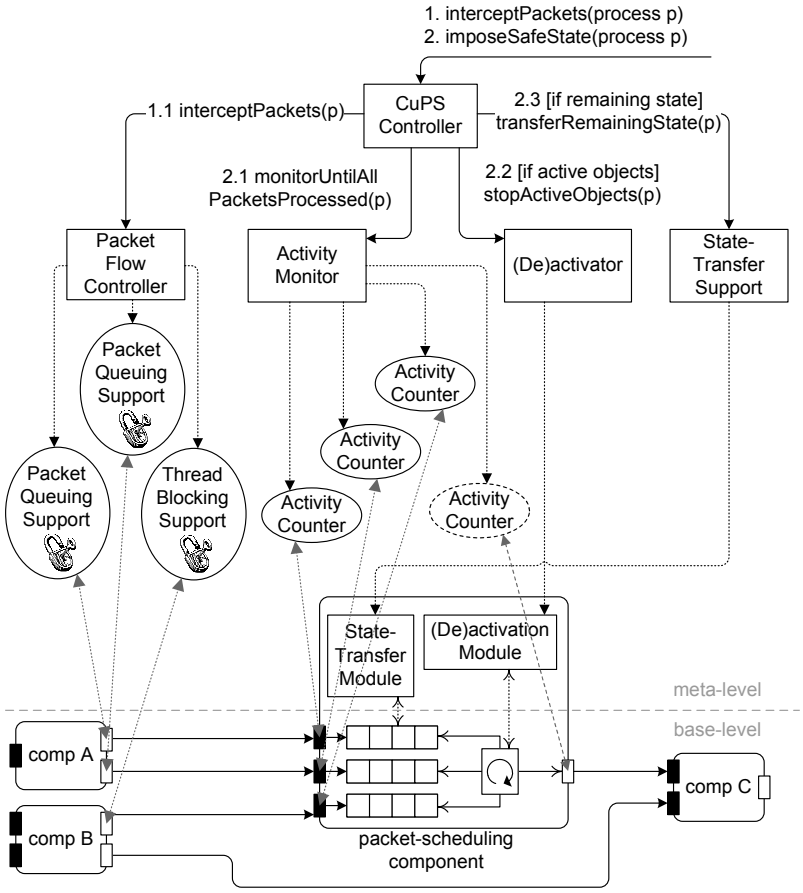


Figure 3.5: This model illustrates how CuPS imposes a safe state over a packet-scheduling component by monitoring until all accepted packets are processed

If a hot-swappable component does not contain active objects (such as a timer), CuPS locates a simple counter at each of its inports when it becomes created. These activity-counters monitor whether or not the affected component has processed all accepted packets. They are incremented for each packet that reaches an inport, and become decremented when the associated threads return. Consequently, when a component's activity-counters are zero, it is currently not processing packets.

When the affected component contains active objects, however, monitoring its inports does not suffice to control whether or not this component is currently processing packets. These active objects process packets in new execution threads. The thread that delivers a packet to a component's inport, therefore, will return and decrement the associated activity-counter as soon as a new execution thread

controls that packet. Activity-counters located at a component's inports, therefore, cannot monitor new threads which are initiated inside that component. To illustrate this, suppose a safe state must be imposed over the packet-scheduling component depicted in Figure 3.5. In that case, an activity-counter is correctly increased when a packet enters the component. However, once that packet is stored in one of the component's buffers, the associated thread returns and decreases the activity-counter. Consequently, there is no guarantee that the packet-scheduling component has processed all packets when its activity-counters are zero. CuPS solves this by monitoring both the packets that enter and those that leave the affected component if the latter employs active objects (see Figure 3.5).

However, this excludes the option of new packets being created or existing packets being removed inside a component. For example, a fragmentation component breaking up each packet into several fragments will never have an equal balance between the amount of packets that enter and leave that component. Each component, therefore, must keep track in this case of the amount of packets that it creates and removes. The affected component then has processed all packets once the amount of incoming, outgoing, created and removed packets is in balance. To be precise, this balanced state is achieved when the number of entered and created packets equals the number of packets that are removed or have left that component.

Impose safe state: deactivate all active objects

As a next step to impose a safe state, CuPS stops the affected component's active objects once this component has processed all accepted packets. To accomplish this, DiPS+ component developers must provide specific support to deactivate every active object that a component employs. This support is encapsulated in a "(de)activation module", and becomes invoked by CuPS' "(de)activator" when needed (see Figure 3.5). Note that this (de)activation module also encapsulates functionality to start the component's active objects, so as to assist CuPS also in activating this component's processes.

Impose safe state: state transfer

Finally, when the affected component contains execution state that goes beyond the execution of a single packet, CuPS' "state-transfer support" transfers this state from the old towards the new DiPS+ component. Note that CuPS only captures a generic state representation from the old component, and then reinstates this generic state in the new component. Support to generate and interpret this generic representation (again) must be provided by the component developer. As illustrated in Figure 3.5, this support must be encapsulated in "state-transfer modules", which are responsible for converting component specific state-information into a generic representation and vice versa [68].

3.3.3 Completing all accepted protocol-transactions

When the affected component communicates according to a predefined communication protocol, simply preventing this component from being invoked and monitoring it until all accepted packets are processed is insufficient to reach a safe state. When this is the case, a reconfiguration-safe state can be achieved by (1) monitoring the affected component until it is not participating anymore in ongoing transactions, and (2) preventing this component from participating in new protocol-transactions until after completing the reconfiguration[4]. This requires to first intercept (some of) the packets directed towards the affected component. Once this is achieved, imposing a safe state involves

1. resuming intercepted packets one by one until all accepted protocol-transactions complete,
2. stopping the activity of active objects once all accepted protocol-transactions are completed, and
3. transferring remaining state that results from (a history of) previously completed transactions.

We discuss in the remainder of this subsection how CuPS implements this scenario. This scenario is illustrated in Figure 3.6 as backing.

Intercept packets

When instructing CuPS to intercept packets, it accomplishes this in the same way as it does when adopting the previous approach to reach a safe state – that is, by holding up packets at the outports of neighboring components. Note, however, that in this case not necessarily all packet-flows directed towards the affected component must be intercepted.

Depending on whether CuPS intercepts packets directed to a client or a server process, it holds up different packet-flows. When a client process is involved, CuPS only intercepts packets that invoke the *service-external* inports associated with that process. Packets that enter this client process through a service-internal port are sent by a collaborating server process and thus belong to an ongoing protocol-transaction. Consequently, intercepting these packets will prevent ongoing protocol-transactions to complete. In case of R_{old}, CuPS only intercepts packets that are directed towards R_{old}'s data-inport (as illustrated in Figure 3.7). This suffices to prevent the initiation of new protocol-transactions.

When a server process in involved, CuPS can intercept packets in two different ways. A first approach includes preventing the associated client processes from transmitting new packets (based on what is proposed in [80]). As we published in [67], this method has two important disadvantages. First, it cannot be applied in

[4] as discussed in Section 3.2.2

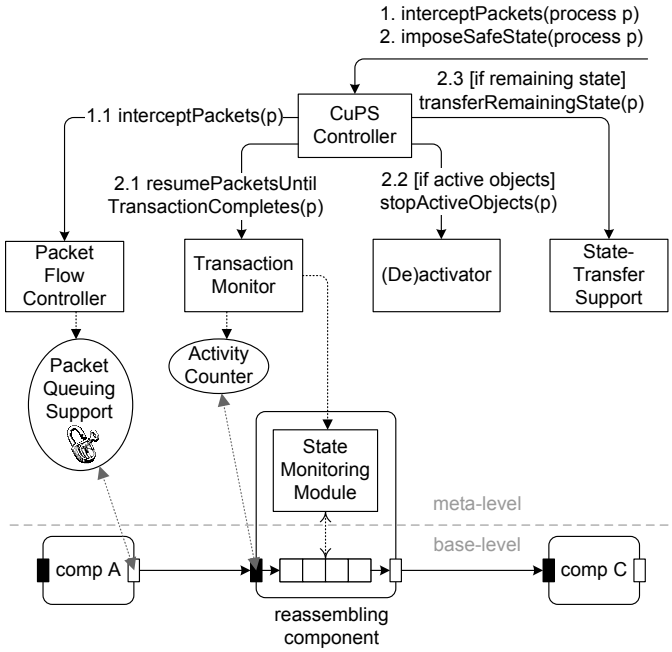


Figure 3.6: This model illustrates how CuPS imposes a safe state over a reassembling component by waiting until the affected component does not participate anymore in ongoing protocol-transactions.

a (hybrid) network where only the node hosting the affected server process supports dynamic reconfiguration[5]. Second, this approach involves (unnecessarily) draining the network pipe between client and server processes.

CuPS therefore holds up packet-flows of neighboring components (located on the same node) to prevent a server process from accepting new service requests. Note that in this case CuPS has to intercept all packet-flows directed to the service-internal inports of the affected (server) processes[6]. When A_{old} is involved, for instance, CuPS intercepts packets that are directed towards A_{old}'s data-inport (as illustrated in Figure 3.8).

Impose safe state: resume intercepted packets until not engaged in ongoing transactions

Once packets are intercepted, CuPS can be instructed to impose a safe state. This involves, among others, controlling the execution of accepted protocol-transactions

[5]for instance because the client processes belong to legacy protocol stacks, or because performance issues exclude interfering in the operation of these stacks

[6]Recall that server processes expose no service-external inports, only service-internal inports.

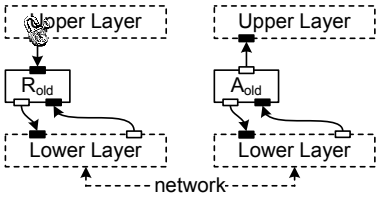


Figure 3.7: Intercepting packets directed towards R_{old}

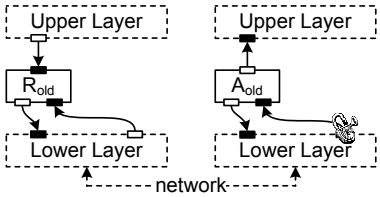


Figure 3.8: Intercepting packets directed towards A_{old}

until they complete. CuPS' "transaction monitor" therefore observes the execution state of the affected processes and, if needed, resumes packets one by one for these protocol-transactions to complete.

We illustrate this with two examples. First, consider replacing the reassembling component of a fragmentation service (see Figure 3.6). Recall that this component encapsulates a *server* process. At the moment when CuPS intercepts packets that are directed towards this component's (service-internal) inports, there is no knowledge about the state of ongoing protocol-transactions. CuPS therefore first checks if each fragment of the packet that is currently being reassembled has arrived, and if the restored packet is delivered to the next component in the pipeline. If this is not the case, CuPS resumes intercepted packets one by one until this required state comes about. As a second example, consider again replacing the old retransmission component, which encapsulates a *client* process. This retransmission process is not participating in ongoing protocol-transactions once every transmitted packet has been acknowledged. When instructed to impose a safe state, CuPS monitors R_{old}'s retransmission queue until it is empty. Note that in this case there is no need to resume intercepted packets.

To assist CuPS in completing accepted protocol-transactions, DiPS+ component developers must provide functionality that determines when a component reaches a consistent execution state. This functionality is encapsulated by a "state-monitoring module", as illustrated in Figure 3.6. In case of the reassembling component, this module controls if the component's buffer (to store all fragments of a packet that is being reassembled) is empty. The state-monitoring module of R_{old}, in contrast, checks whether or not this component's retransmission queue is empty. Besides, the component developer must also specify which intercepted packet flows are to be resumed in order to reach a safe state.

Impose safe state: deactivate all active objects

Once all accepted protocol-transactions are completed, CuPS stops the active objects of the affected processes. Similar as for the previous approach to reach a safe state, DiPS+ component developers must provide (de)activation modules to assist CuPS in accomplishing this.

Impose safe state: state transfer

Finally, when the affected component contains state-information that goes beyond the execution of a single protocol-transaction, CuPS transfers this state from the old towards the new DiPS+ component. An example of such state-information includes the sequence number that R_{old} has last used. Again, to enable state transfer, DiPS+ component developers must provide support for converting component specific state-information into a generic representation and vice versa.

3.3.4 State transfer

As a last approach to reach a reconfiguration-safe state, CuPS supports to deactivate a component immediately instead of monitoring it until all accepted packets or protocol-transactions are processed. Again this involves first intercepting all packets that are directed to the affected component. After that, CuPS imposes a safe state by

1. stopping the active objects of this component, and
2. monitoring this component until it is *idle*, and
3. transferring this component's current execution state towards the new component, and
4. transferring all remaining state-information towards the new component

In the remainder of this subsection we explain how CuPS puts this scenario into practice. To illustrate this scenario, Figure 3.9 depicts the implementation of both operations to intercept packets and to impose a safe state over R_{old}.

Intercepting packets

When instructing CuPS to intercept packets, all packets directed towards the affected component will be held up. Similar as for the first approach to reach a safe state, in this case CuPS makes no distinction between packet-flows that are directed to service-internal or service-external inports.

Impose safe state: deactivate all active objects

Once all packets directed towards the affected component are intercepted, CuPS can be invoked to impose a safe state. This involves first deactivating all active objects that this component employs. Similar as for both previous approaches to reach a safe state, CuPS invokes the component's (de)activation module to accomplish this.

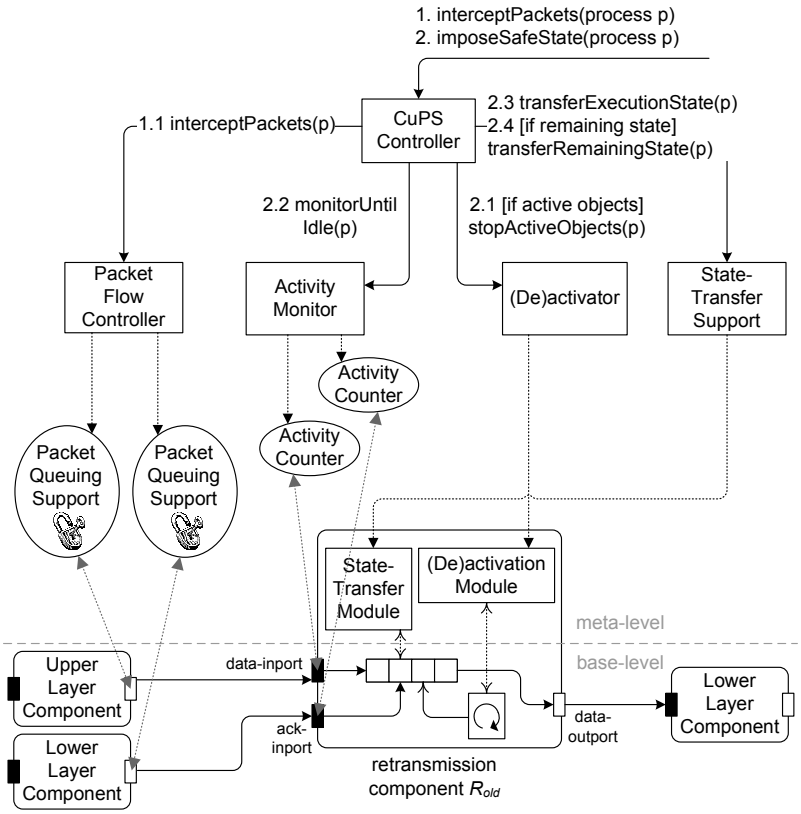


Figure 3.9: This model illustrates how CuPS imposes a safe state by deactivating the affected retransmission component and transferring its current execution state to the new component.

Impose safe state: monitor until idle

Next, CuPS monitors the affected component until it is idle. Because its active objects are already stopped, this component is idle once all (external) threads delivering packets to this component have returned. CuPS therefore monitors the affected component by means of activity-counters that are located at the component's inports (as illustrated in Figure 3.9). These activity-counters are identical to the ones that CuPS uses for monitoring a component until it has processed all accepted packets. Hence, a component is idle if its activity-counters are zero.

Impose safe state: transfer execution state and remaining state-information

Finally, and most important, CuPS captures the state of the affected component and reinstates this state in the new component. This way consistency will be recovered. Similar as for the previous approaches to reach a safe state, it is the component developers' responsibility to provide support for converting component specific state-information into a generic representation and vice versa.

3.4 Notation for reconfiguration conditions

In the previous section we explained in detail how CuPS supports NeCoMan to reach a reconfiguration-safe state. The complete execution of a reconfiguration, however, involves extra actions besides reaching a reconfiguration-safe state. These include, among others, loading a new component into the node's reconfiguration support, activating this component, removing the old version, etc. For a reconfiguration to be effective, the execution of these actions must be coordinated correctly. Activating a new component before it has been installed, for instance, obviously makes a correct reconfiguration impossible.

For that reason we will define in the remainder of this and the following chapters a number of "reconfiguration conditions" that NeCoMan must fulfill to carry out a correct reconfiguration. These reconfiguration conditions define the partial order in which the involved reconfiguration actions must be executed. Hence, they (1) provide some guidance for the development of reconfiguration support, and (2) allow to reason about the reconfiguration process. By checking NeCoMan's reconfiguration algorithms against these reconfiguration conditions, we can demonstrate (in an intuitive way) the correctness of these algorithms.

The reconfiguration conditions that we identify in the remainder of this and the following chapters express logical implication relationships (defined by the $\leftarrow$ operator) in which the right operand specifies the reconfiguration actions that must be carried out before the actions specified in the left operand can be initiated[7]. Reconfiguration condition $C \leftarrow B \wedge A$, for instance, expresses that reconfiguration action C can only be initiated when actions B and ($\wedge$) A have completed. When condition $D \leftarrow B \wedge A$ must be fulfilled as well, both reconfiguration conditions can be combined into condition $D \wedge C \leftarrow B \wedge A$. The resulting reconfiguration condition expresses that actions D and C can only be initiated when B and A have completed. Besides, note that the $\leftarrow$ operator is transitive[8], antisymmetric[9] and irreflexive[10].

[7] Note that we did not use an existing calculus because of the simplicity of these reconfiguration conditions, and because we do not intend to provide mathematical proof of the correctness of NeCoMan's algorithms.

[8] if $C \leftarrow B$ and $B \leftarrow A$ are fulfilled, then we can conclude that $C \leftarrow A$ is fulfilled as well

[9] if $B \leftarrow A$ and $B \neq A$ then $A \leftarrow B$ cannot occur

[10] $B \leftarrow B$ cannot occur

Furthermore, we use the $\equiv$ operator to denote how a high-level action is composed out of more fine-grained actions. Expression $A \equiv E \curlywedge F$, for instance, specifies that the execution of A includes actions E and ($\curlywedge$) F. Consequently, this implies that action A in expression $C \leftarrow B \wedge A$ can be substituted for actions E and F, which results in $C \leftarrow B \wedge E \wedge F$.

Finally, note that we do not intend to use this notation for providing mathematical proof of the correctness of NeCoMan's algorithms. Instead, we use this notation to illustrate in a systematic manner how NeCoMan's reconfiguration conditions have come about.

3.5 Local reconfigurations of distributed services

The remainder of this chapter builds up to both basic algorithms that NeCoMan uses for conducting local reconfigurations. One of these algorithms coordinates the replacement of components that participate in a distributed network service. The other one, in contrast, coordinates the addition, replacement, and removal of isolated network-service components. Both general-purpose algorithms do not take into account the characteristics of the affected services nor the reconfiguration semantics. Instead, they each seek to safely reconfigure as many network services as possible, and thus serve the same purpose as the single algorithm that a black-box reconfiguration system typically employs.

This section builds up to the algorithm that NeCoMan uses for replacing a component of a distributed service. First, Subsection 3.5.1 distinguishes four high-level reconfiguration phases and defines the associated reconfiguration conditions. Next, Subsection 3.5.2 presents for each reconfiguration phase the reconfiguration actions to implement the behavior of these phases. Besides, this subsection also specifies for each reconfiguration phase the reconfiguration conditions that NeCoMan must fulfill to correctly execute these phases. Next, Subsection 3.5.3 completes this set of reconfiguration conditions by refining the high-level reconfiguration conditions defined in Subsection 3.5.1 in terms of the reconfiguration actions defined in Subsection 3.5.2. Finally, Subsection 3.5.4 presents the algorithm itself, and indicates that this algorithm fulfills all required reconfiguration conditions.

3.5.1 High-level reconfiguration phases and conditions

NeCoMan's algorithm to replace a component of a distributed service comprises the following four phases:

- *installing the new service component* – by making the new component available on the affected node,
- *activating the new service component* – by bringing this new component into use,

- *finishing the old service component* – by driving the old component to a reconfiguration-safe state, and
- *removing the old service component* – by deleting the old component from the affected node.

With respect to these reconfiguration phases, NeCoMan must satisfy the following (high-level) reconfiguration conditions:

1. The new component can only be activated when it is made available on the node where needed. We express this trivial safety condition as

$$AC_{new} \leftarrow IC_{new} \tag{H.1}$$

   where AC_{new} and IC_{new} represent activating and installing the new component, respectively[11].

2. The new component can only be activated safely when the old version has reached a reconfiguration-safe state – that is, when the old component is finished (FC_{old}). Hence, the new component will be initialized in a state which is consistent with the rest of the network before it is brought in use. We express this reconfiguration condition as

$$AC_{new} \leftarrow FC_{old} \tag{H.2}$$

3. The removal of the old component (RC_{old}) can only be initiated safely when it has reached a reconfiguration-safe state. This gives network service components which are being removed the opportunity to leave the network in a consistent state. We express this reconfiguration condition as follows:

$$RC_{old} \leftarrow FC_{old} \tag{H.3}$$

Both the above-mentioned reconfiguration conditions determine a partial order in which the four high-level reconfiguration phases must be executed. This order is depicted in Figure 3.10.

3.5.2 Detailed overview of each reconfiguration phase

We now zoom in on each of the four reconfiguration phases and identify the reconfiguration actions that NeCoMan executes to implement these high-level phases (all NeCoMan's reconfiguration actions are summarized in Appendix A as backing). As will be clarified soon, these reconfiguration actions each implement a specific reconfiguration subtask, such as loading a new component into the node's protocol stack

[11] High-level reconfiguration conditions are labelled (H.x)

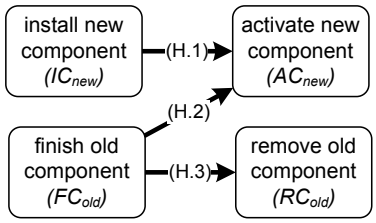


Figure 3.10: Partial ordering of NeCoMan's (high-level) reconfiguration phases for carrying out local reconfigurations of distributed services. According to the associated reconfiguration conditions, each of these high-level reconfiguration phase can only be initiated safely if all its referring phases are completed.

composition, activating this component's active objects, disconnecting the old component from the protocol stack composition, etc. Note that these reconfiguration actions have been chosen thoroughly. On the one hand, they should not be defined at a too low abstraction level, since this has a negative effect on error-proneness and comprehensibility. At the same time, however, their abstraction level should not be raised too far either, as this compromises the ability to customize the employed reconfiguration algorithms (for instance by discarding actions that are redundant for a particular reconfiguration).

Besides, note that reconfiguration *actions* and reconfiguration *operations* represent different abstraction levels. A node's reconfiguration support is expected to provide the eight reconfiguration *operations* (specified in Section 3.3.1) to assist a reconfiguration middleware in carrying out a reconfiguration. A reconfiguration *action*, instead, combines the execution of a number of these reconfiguration operations to implement a specific reconfiguration subtask.

Installation phase

Let us start with the installation phase. The installation of a new service component on the affected node involves loading and connecting this component into the node's protocol stack composition. Because the new component is not yet activated in this phase, only the bindings that will not cause service invocation on the new component can be constructed – that is, only the new component's outports become connected. By partially connecting the new component into the protocol stack composition during installation, we aim to limit the communication disruption that a reconfiguration causes. When activating the new component, only bindings that mediate packets to the incoming ports of the new component will have to be created (instead of constructing all outgoing bindings as well). The duration of the activation phase hence becomes slightly reduced. To illustrate this installation phase, Figure 3.11 sketches the installation of R_{new}. The black bold component and bindings in this figure are created after completing this phase.

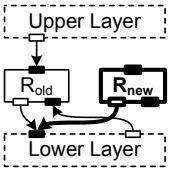


Figure 3.11: Installation of R_{new}

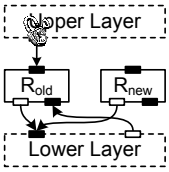


Figure 3.12: Finishing of R_{old}

NeCoMan implements this installation phase by executing reconfiguration actions CC_{new}, LO^{ext}_{new} and LO^{int}_{new}, where

- CC_{new} denotes a reconfiguration action responsible for loading the new service component into the node's protocol stack composition (which involves invoking the node's "create" operation), and
- LO^{ext}_{new} and LO^{int}_{new} symbolize reconfiguration actions for linking this new component's service-external and service-internal outports, respectively (which involves invoking the node's "link" operation).

Hence, we express the implementation of this installation phase as[12]

$$IC_{new} \equiv CC_{new} \curlywedge LO^{ext}_{new} \curlywedge LO^{int}_{new} \tag{P.1}$$

Finally, it is obvious that NeCoMan can only instruct the affected node to bind a component's outports if this component has already been created. We express this reconfiguration condition as

$$LO^{ext}_{new} \wedge LO^{int}_{new} \leftarrow CC_{new} \tag{3.1}$$

Finishing phase

The finishing phase involves driving the client and server processes of the affected component to a reconfiguration-safe state. As we discussed in detail in Section 3.3, this includes instructing the underlying node's reconfiguration support to intercept packets and to impose a safe state over each of these processes. NeCoMan therefore implements this finishing phase by executing reconfiguration actions IP^{client}_{old}, ISS^{client}_{old}, IP^{server}_{old}, and ISS^{server}_{old}, where

- IP^{client}_{old} and ISS^{client}_{old} symbolize reconfiguration actions for intercepting packets and imposing a safe state over a component's client processes (which involve invoking the node's "intercept packets" and "impose safe state" operations), and

[12] Representations of reconfiguration phases are labelled (P.x).

- IP_{old}^{server} and ISS_{old}^{server} denote the same actions but targeted at this component's server processes.

Hence, we express the implementation of this finishing phase as

$$FC_{old} \equiv IP_{old}^{client} \curlywedge ISS_{old}^{client} \curlywedge IP_{old}^{server} \curlywedge ISS_{old}^{server} \tag{P.2}$$

To illustrate (part of) this finishing phase, Figure 3.12 depicts the execution of IP_{old}^{client} on the node hosting component R_{old}[13]. Note that the lock symbolizes that packets are intercepted at the neighboring components.

To correctly finish an old component, NeCoMan must coordinate the order in which it executes IP_{old} and ISS_{old}. To be precise, NeCoMan must first instruct the affected node to intercept packets before invoking it to impose a safe state. This applies to finishing a component's client processes as well as its server processes. We denote this reconfiguration condition as follows

$$ISS_{old}^{client} \leftarrow IP_{old}^{client} \tag{3.2}$$
$$ISS_{old}^{server} \leftarrow IP_{old}^{server} \tag{3.3}$$

Activation phase

The activation phase includes instructing the underlying node to

- rebind all service-external and service-internal inports such that packets (which were originally directed to the old component) will become mediated to the new component after resuming the intercepted packet-flows, and to
- activate all client and server processes that the new component encapsulates, and to
- resume packets that have been intercepted to finish the old component.

NeCoMan implements this activation phase by executing reconfiguration actions $LI_{old-new}^{int}$, $LI_{old-new}^{ext}$, AP_{new}^{client}, AP_{new}^{server}, RP_{new}^{client}, and RP_{new}^{server}, where

- $LI_{old-new}^{int}$ and $LI_{old-new}^{ext}$ represent reconfiguration actions for rebinding all service-internal and service-external inports, respectively (which involves invoking the node's "link" and "unlink" operations),

[13]To be complete, for this example the execution of ISS_{old}^{client} involves first monitoring R_{old}'s retransmission queue until it is empty. After that, the node's reconfiguration support stops R_{old}'s retransmission timer, captures the last sequence number that R_{old} has attached to an outgoing packet, and reinstates this information in R_{new}.

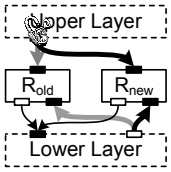


Figure 3.13: Binding inports of R_{new}

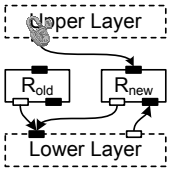


Figure 3.14: Releasing intercepted packets

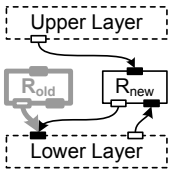


Figure 3.15: Removal of R_{old}

- AP_{new}^{client} and AP_{new}^{server} symbolize reconfiguration actions responsible for activating all encapsulated client and server processes (which involves invoking the node's "activate processes" operation), and
- RP_{new}^{client} and RP_{new}^{server} denote reconfiguration actions for resuming all intercepted packets to invoke the new client and server processes (which involves invoking the node's "release packets" operation).

Hence, we express the implementation of this activation phase as

$$AC_{new} \equiv LI_{old-new}^{int} \curlywedge LI_{old-new}^{ext} \curlywedge AP_{new}^{client} \curlywedge AP_{new}^{server} \curlywedge RP_{new}^{client} \curlywedge RP_{new}^{server} \tag{P.3}$$

To illustrate (part of) this activation phase, Figure 3.13 depicts redirecting (intercepted) packet-flows to the inports of component R_{new}. This involves unlinking R_{old}'s inports and simultaneously linking those of R_{new}, which is symbolized by grey and black bold connections, respectively. In addition, Figure 3.14 illustrates the resuming of intercepted packets.

To correctly activate a new component, NeCoMan must coordinate the order in which these actions are executed. To be precise, intercepted packets can only be resumed once the new component's processes are fully operational and prepared to accept and service these packets. This implies that their active objects are started, and that all inports involved have been bound correctly.

When targeted at *client processes*, this reconfiguration condition can be expressed as follows

$$RP_{new}^{client} \quad \leftarrow \quad AP_{new}^{client} \wedge LI_{old-new}^{int} \wedge LI_{old-new}^{ext} \tag{3.4}$$

In case of *server processes*, however, $LI_{old-new}^{ext}$ becomes redundant. Because a server process merely reacts to client processes, it only employs service-internal inports. Hence, intercepted packets that will invoke the new server processes can be released if the service-internal inports of the new component are bound, and if the active objects of the affected server processes are started. We denote this

reconfiguration condition as follows

$$RP_{new}^{server} \quad \leftarrow \quad AP_{new}^{server} \wedge LI_{old-new}^{int} \tag{3.5}$$

Removal phase

The removal phase involves both disconnecting the affected component from the stack composition and deleting this component. This is illustrated in Figure 3.15, where the grey bold component and bindings are removed after completing this phase of the reconfiguration. Because the old component must be in a reconfiguration-safe state according to reconfiguration condition (H.3), removing the old component does not compromise the correct operation of the network.

NeCoMan implements this removal phase by executing reconfiguration actions DC_{old}, UO_{old}^{ext} and UO_{old}^{int}, where

- DC_{old} denotes a reconfiguration action responsible for deleting the old service component from the node's protocol stack composition (which involves invoking the node's "remove" operation), and
- UO_{old}^{ext} and UO_{old}^{int} symbolize reconfiguration actions for unlinking this component's service-external and service-internal outports, respectively (which involves invoking the node's "unlink" operation).

Hence, we express the implementation of this removal phase as

$$RC_{old} \equiv DC_{old} \curlywedge UO_{old}^{ext} \curlywedge UO_{old}^{int} \tag{P.4}$$

Finally, NeCoMan can only delete a component once its communication ports (that is, both its inports and outports) are disconnected. We denote this reconfiguration condition as

$$DC_{old} \leftarrow UO_{old}^{ext} \wedge UO_{old}^{int} \wedge LI_{old-new}^{ext} \wedge LI_{old-new}^{int} \tag{3.6}$$

3.5.3 Refining high-level reconfiguration conditions

In the previous subsection we presented the reconfiguration actions that NeCoMan uses to implement the four high-level reconfiguration phases. In addition, we also defined the reconfiguration conditions that NeCoMan must fulfill to correctly execute these reconfiguration phases. We now combine all this with the partial ordering of the four reconfiguration-phases defined in Subsection 3.5.1. As a result, we achieve a more detailed partial ordering of the employed reconfiguration actions. This ordering is illustrated in Figure 3.16.

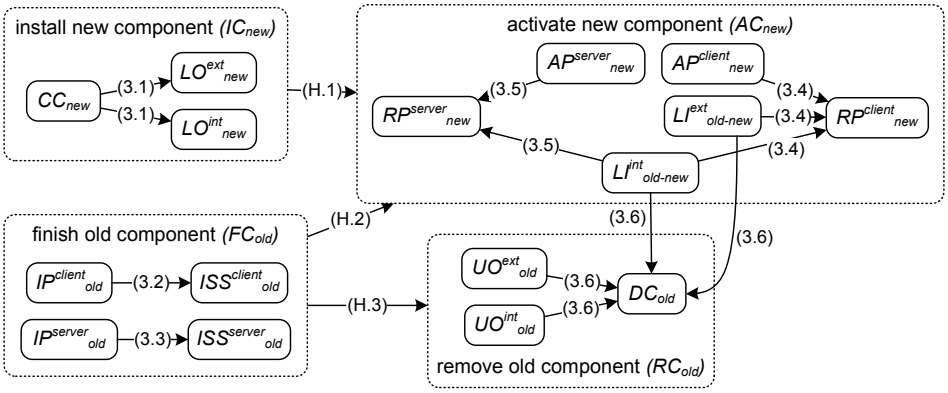


Figure 3.16: Preliminary partial ordering of NeCoMan's reconfiguration actions for carrying out local reconfigurations of distributed services. According to the associated reconfiguration conditions, each of these reconfiguration actions can only be initiated safely if all its referring actions are completed.

To complete this partial ordering, we must refine reconfiguration conditions (H.1), (H.2), and (H.3) in terms of the reconfiguration actions presented in the previous subsection. Note that these refinements are all achieved in a similar way. For each high-level reconfiguration condition, we first substitute IC_{new}, FC_{old}, AC_{new} and RC_{old} for expressions (P.1), (P.2), (P.3), and (P.4), respectively. Next, we investigate if the resulting reconfiguration condition can be made less stringent without compromising the correctness of the reconfiguration scenario (for instance because part of this condition is already covered by another reconfiguration condition).

Installing new component before activation (H.1)

Let us start with reconfiguration condition (H.1), which imposes to only activate a new component once it is installed properly on the affected node. To refine this condition, we replace IC_{new} and AC_{new} by expressions (P.1) and (P.3), respectively. This results in the following reconfiguration condition[14]

$$LI^{ext}_{old-new} \wedge LI^{int}_{old-new} \wedge AP^{client}_{new} \wedge AP^{server}_{new} \wedge RP^{client}_{new} \wedge RP^{server}_{new} \leftarrow CC_{new} \wedge LO^{ext}_{new} \wedge LO^{int}_{new} \quad \text{(I.1)}$$

This condition can be made less stringent without compromising the reconfiguration correctness. To illustrate this, we split up condition (I.1) into (I.2a), (I.2b) and (I.2c), and reduce each of these conditions.

[14] Intermediate reconfiguration conditions are labelled (I.x).

$$LI^{ext}_{old-new} \wedge LI^{int}_{old-new} \leftarrow CC_{new} \wedge LO^{ext}_{new} \wedge LO^{int}_{new} \quad \text{(I.2a)}$$
$$AP^{client}_{new} \wedge AP^{server}_{new} \leftarrow CC_{new} \wedge LO^{ext}_{new} \wedge LO^{int}_{new} \quad \text{(I.2b)}$$
$$RP^{client}_{new} \wedge RP^{server}_{new} \leftarrow CC_{new} \wedge LO^{ext}_{new} \wedge LO^{int}_{new} \quad \text{(I.2c)}$$

Condition (I.2a). Unlike what condition (I.2a) defines, the execution of $LI^{ext}_{old-new}$ and $LI^{int}_{old-new}$ does not require for actions CC_{new}, LO^{ext}_{new} and LO^{int}_{new} to be completed. Instead, the new component's inports can safely be bound once this component is available on the affected node. Condition (I.2a) therefore can be reduced to

$$LI^{ext}_{old-new} \wedge LI^{int}_{old-new} \leftarrow CC_{new} \quad \text{(I.3)}$$

Condition (I.2b). The same reduction does not apply to condition (I.2b). Before activating a new component's client and/or server processes, its outports must be bound correctly. This is required to manage when the node's reconfiguration support does not impose a safe state by monitoring the affected processes, but instead deactivates these processes immediately and transfers their execution state towards the new processes. To illustrate this, consider imposing a safe state over R_{old} by deactivating its retransmission timer and transferring all unacknowledged packets towards R_{new}. Once R_{new}'s timer is started afterwards, R_{new} continues retransmitting these packets. All outports of the new component therefore must be bound correctly before starting these active objects.

Nevertheless, condition (I.2b) can still be made less stringent because part of its right operand is already covered by condition (3.1). The latter defines to only initiate LO^{ext}_{new} and LO^{int}_{new} once CC_{new} has completed. So, when LO^{ext}_{new} and LO^{int}_{new} are executed, then CC_{new} will have been completed already. Hence, we can safely remove CC_{new} from the right operand of condition (I.2b), which results in

$$AP^{client}_{new} \wedge AP^{server}_{new} \leftarrow LO^{ext}_{new} \wedge LO^{int}_{new} \quad \text{(I.4)}$$

In addition, recall that a client process does not expose service-external outports (as illustrated in Figure 2.16). Condition (I.4) therefore can safely be reduced to

$$AP^{client}_{new} \leftarrow LO^{int}_{new} \quad \text{(I.5)}$$
$$AP^{server}_{new} \leftarrow LO^{ext}_{new} \wedge LO^{int}_{new} \quad \text{(I.6)}$$

Condition (I.2c). Finally, we can safely weaken condition (I.2c) in an analogous way. When reconfiguration condition (3.1) is fulfilled, then CC_{new} can safely be removed from the right operand of condition (I.2c). This results in

$$RP_{new}^{client} \wedge RP_{new}^{server} \leftarrow LO_{new}^{ext} \wedge LO_{new}^{int} \tag{I.7}$$

In addition, because a client process does not employ service-external outports, condition (I.7) can safely be reduced to

$$RP_{new}^{client} \leftarrow LO_{new}^{int} \tag{I.8}$$
$$RP_{new}^{server} \leftarrow LO_{new}^{ext} \wedge LO_{new}^{int} \tag{I.9}$$

Conclusion. To conclude, NeCoMan thus meets reconfiguration condition (H.1) when it fulfills conditions (3.1), (I.3), (I.5), (I.6), (I.8), and (I.9). To simplify the representation of these constraints, we combine all conditions except (3.1). This results in reconfiguration conditions (3.7), (3.8), and (3.9).

$$LI_{old-new}^{ext} \wedge LI_{old-new}^{int} \leftarrow CC_{new} \tag{3.7}$$
$$AP_{new}^{client} \wedge RP_{new}^{client} \leftarrow LO_{new}^{int} \tag{3.8}$$
$$AP_{new}^{server} \wedge RP_{new}^{server} \leftarrow LO_{new}^{ext} \wedge LO_{new}^{int} \tag{3.9}$$

Finishing old component before activating new version (H.2)

Next, we refine reconfiguration condition (H.2). This condition dictates to only activate the new network service when all components that belong to the old service have reached a reconfiguration-safe state. Replacing FC_{old} and AC_{new} in (H.2) by expressions (P.2) and (P.3) results in

$$\begin{aligned} LI_{old-new}^{ext} \wedge LI_{old-new}^{int} \wedge AP_{new}^{client} \wedge AP_{new}^{server} \wedge RP_{new}^{client} \wedge RP_{new}^{server} \\ \leftarrow IP_{old}^{client} \wedge ISS_{old}^{client} \wedge IP_{old}^{server} \wedge ISS_{old}^{server} \end{aligned} \tag{I.10}$$

We can weaken this condition without compromising the correct functioning of the programmable network or its services in the course of a reconfiguration. We do this in two steps. First, we simplify condition (I.10) by relying on conditions (3.2) and (3.3). These conditions impose to (1) only initiate ISS_{old}^{client} when IP_{old}^{client} is

completed, and (2) only initiate ISS_{old}^{server} when IP_{old}^{server} is completed. Given that these conditions will be fulfilled, we can safely remove IP_{old}^{client} and IP_{old}^{server} from the right operand of expression (I.10). This results in

$$LI_{old-new}^{ext} \wedge LI_{old-new}^{int} \wedge AP_{new}^{client} \wedge AP_{new}^{server} \wedge RP_{new}^{client} \wedge RP_{new}^{server} \leftarrow ISS_{old}^{client} \wedge ISS_{old}^{server} \tag{I.11}$$

Next, we split up condition (I.11) into (I.12a), (I.12b) and (I.12c), and reduce each of these conditions.

$$LI_{old-new}^{ext} \wedge LI_{old-new}^{int} \leftarrow ISS_{old}^{client} \wedge ISS_{old}^{server} \tag{I.12a}$$

$$AP_{new}^{client} \wedge AP_{new}^{server} \leftarrow ISS_{old}^{client} \wedge ISS_{old}^{server} \tag{I.12b}$$

$$RP_{new}^{client} \wedge RP_{new}^{server} \leftarrow ISS_{old}^{client} \wedge ISS_{old}^{server} \tag{I.12c}$$

Condition (I.12a). Condition (I.12a) expresses that $LI_{old-new}^{ext}$ and $LI_{old-new}^{int}$ can only be executed when the old client and server processes of the affected component are finished. Otherwise, packets that should be delivered to the old processes will become directed towards the new one, which compromises the correct network functioning. As illustrated in Figure 2.16, a component's service-external inports will only be used by its client processes (in contrast to the service-internal inports which are used by both client and server processes). Hence, only these client processes must be finished before rebinding service-external inports. We can therefore safely reduce condition (I.12a) to

$$LI_{old-new}^{ext} \leftarrow ISS_{old}^{client} \tag{I.13}$$

$$LI_{old-new}^{int} \leftarrow ISS_{old}^{client} \wedge ISS_{old}^{server} \tag{I.14}$$

Condition (I.12b). Next, we reduce condition (I.12b). Before starting the active objects of the new client and server processes, the old processes must be in a reconfiguration-safe state. This is required again to manage when the node's reconfiguration does not impose a safe state by monitoring the affected processes, but instead deactivates these processes immediately and transfers their execution state towards the new processes. When replacing R_{old}, for instance, the timer of the new retransmission process can only be started once the unacknowledged packets and the sequence number that is last used are transferred from R_{old} towards R_{new}.

Note, however, that there is no need wait for both the old client and server processes to reach a safe state before activating the new client or server processes. This is because a component's client and server processes operate independently

from each other (see Figure 2.16), and therefore will never invoke each other (not directly nor indirectly). AP_{new}^{client} thus can safely be initiated once ISS_{old}^{client} has completed, while the execution of AP_{new}^{server} only requires for ISS_{old}^{client} to be ended. Hence, condition (I.12b) can be reduced to

$$AP_{new}^{client} \leftarrow ISS_{old}^{client} \tag{I.15}$$
$$AP_{new}^{server} \leftarrow ISS_{old}^{server} \tag{I.16}$$

Condition (I.12c). Finally, we examine condition (I.12c). This condition dictates to only release intercepted packets once the old client and server processes are finished. Similar as for starting the active objects that these processes employ, resuming intercepted packets that will be delivered to the new client processes does only require for the old client processes to be finished (instead of waiting for the old server process to finish as well). The same holds for resuming intercepted packets that will be mediated towards the new server processes. NeCoMan can safely release these packets once the old server processes are finished. Hence, condition (I.12c) can be weakened to

$$RP_{new}^{client} \leftarrow ISS_{old}^{client} \tag{I.17}$$
$$RP_{new}^{server} \leftarrow ISS_{old}^{server} \tag{I.18}$$

However, these conditions are already fulfilled when conditions (3.4) and (3.5) are met. Condition (3.4) dictates that RP_{new}^{client} can only be executed if AP_{new}^{client}, $LI_{old-new}^{int}$ and $LI_{old-new}^{ext}$ are completed. According to condition (I.13), $LI_{old-new}^{ext}$ in turn can only be executed once ISS_{old}^{client} is completed. Hence, condition (I.17) becomes redundant and can safely be removed. The same applies to condition (I.18). Condition (3.5) dictates that RP_{new}^{server} can only be executed if AP_{new}^{server} and $LI_{old-new}^{int}$ are completed. According to condition (I.14), $LI_{old-new}^{int}$ in turn can only be executed once ISS_{old}^{client} and ISS_{old}^{server} are completed. Hence, condition (I.18) also becomes redundant and thus can be removed safely as well.

Conclusion. To conclude, NeCoMan thus meets reconfiguration condition (H.2) when it fulfills conditions (3.2), (3.3), (3.4), (3.5), (I.13), (I.14), (I.15), and (I.16). To simplify the representation of these constraints, we combine all conditions except (3.2), (3.3), (3.4), and (3.5). This results in reconfiguration conditions (3.10), (3.11), and (3.12).

$$LI_{old-new}^{int} \leftarrow ISS_{old}^{client} \wedge ISS_{old}^{server} \tag{3.10}$$
$$AP_{new}^{client} \wedge LI_{old-new}^{ext} \leftarrow ISS_{old}^{client} \tag{3.11}$$
$$AP_{new}^{server} \leftarrow ISS_{old}^{server} \tag{3.12}$$

Finishing old component before removal (H.3)

Finally, we can refine reconfiguration condition (H.3) in an analogous way. This reconfiguration condition imposes to only initiate the removal of the old component when all its processes are finished. Replacing FC_{old} and RC_{old} in (H.3) by expressions (P.2) and (P.4) results in

$$DC_{old} \wedge UO^{ext}_{old} \wedge UO^{int}_{old} \leftarrow IP^{client}_{old} \wedge ISS^{client}_{old} \wedge IP^{server}_{old} \wedge ISS^{server}_{old} \tag{I.19}$$

Again we can safely weaken the right operand of this condition. Recall that conditions (3.2) and (3.3) dictate to initiate ISS^{client}_{old} and ISS^{server}_{old} only when IP^{client}_{old} and IP^{server}_{old} are completed, respectively. Given that these conditions will be fulfilled, we can safely remove IP^{client}_{old} and IP^{server}_{old} from the right operand of expression (I.19). This results in

$$DC_{old} \wedge UO^{ext}_{old} \wedge UO^{int}_{old} \leftarrow ISS^{client}_{old} \wedge ISS^{server}_{old} \tag{I.20}$$

Next, we split up condition (I.20) into (I.21a), (I.21b) and (I.21c), and examine each of these conditions.

$$UO^{int}_{old} \leftarrow ISS^{client}_{old} \wedge ISS^{server}_{old} \tag{I.21a}$$

$$UO^{ext}_{old} \leftarrow ISS^{client}_{old} \wedge ISS^{server}_{old} \tag{I.21b}$$

$$DC_{old} \leftarrow ISS^{client}_{old} \wedge ISS^{server}_{old} \tag{I.21c}$$

Condition (I.21a). Condition (I.21a) cannot be reduced. Because both a component's client and server processes may employ service-internal outports (see Figure 2.16), these processes must be finished before NeCoMan can unbind the affected outports.

Condition (I.21b). Because only server processes employ service-external outports (as illustrated in Figure 2.16), NeCoMan does not have to wait for ISS^{client}_{old} to complete before unbinding the old component's service-external outports. Hence, condition (I.21b) can safely be reduced to

$$UO^{ext}_{old} \leftarrow ISS^{server}_{old} \tag{I.22}$$

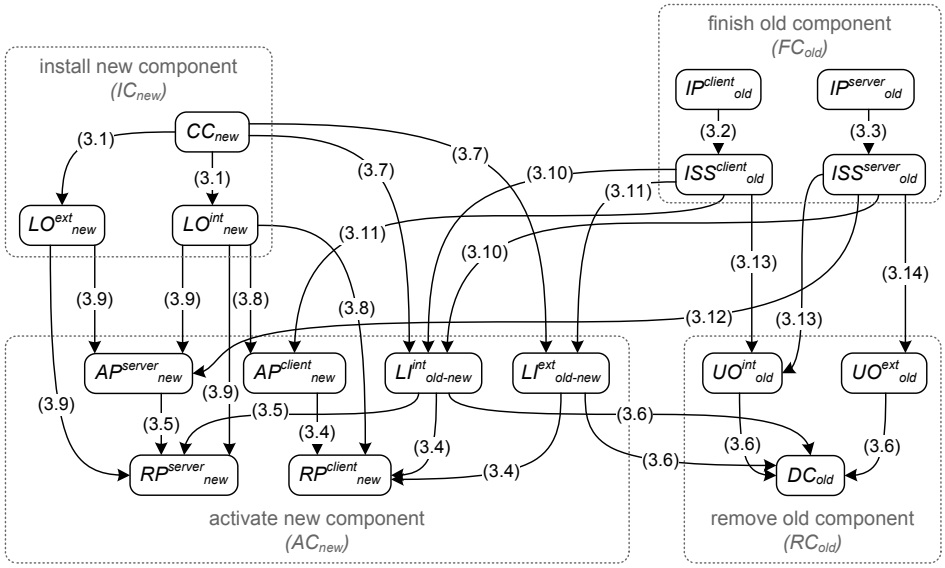


Figure 3.17: Overview of the partial ordering of NeCoMan's reconfiguration actions for carrying out local reconfigurations of distributed services.

Condition (I.21c). Finally, condition (I.21c) is already fulfilled when condition (3.6) is met. The latter dictates that DC_{old} can only be executed if UO^{ext}_{old} and UO^{int}_{old}, $LI^{ext}_{old-new}$, and $LI^{int}_{old-new}$ are completed. According to conditions (I.21a) and (I.22), UO^{ext}_{old} and UO^{int}_{old} in turn can only be executed after completing ISS^{client}_{old} and ISS^{server}_{old}. Hence, condition (I.21c) becomes redundant and can safely be removed.

Conclusion. To conclude, NeCoMan thus meets reconfiguration condition (H.3) when it fulfills conditions (3.2), (3.3), (3.6), (3.13), and (3.14).

$$UO^{int}_{old} \leftarrow ISS^{client}_{old} \wedge ISS^{server}_{old} \tag{3.13}$$

$$UO^{ext}_{old} \leftarrow ISS^{server}_{old} \tag{3.14}$$

Partial ordering of reconfiguration actions

By combining all these reconfiguration conditions (which are summarized in Table 3.3), we can specify the partial ordering of reconfiguration actions that NeCoMan must fulfill. This ordering is illustrated in Figure 3.17.

	reconfiguration condition	h-l cond.	place
(3.1)	$LO^{ext}_{new} \wedge LO^{int}_{new} \leftarrow CC_{new}$		p_2
(3.2)	$ISS^{client}_{old} \leftarrow IP^{client}_{old}$		p_5
(3.3)	$ISS^{server}_{old} \leftarrow IP^{server}_{old}$		p_7
(3.4)	$RP^{client}_{new} \leftarrow AP^{client}_{new} \wedge LI^{int}_{old-new} \wedge LI^{ext}_{old-new}$		p_{12}
(3.5)	$RP^{server}_{new} \leftarrow AP^{server}_{new} \wedge LI^{int}_{old-new}$		p_{11}
(3.6)	$DC_{old} \leftarrow UO^{ext}_{old} \wedge UO^{int}_{old} \wedge LI^{ext}_{old-new} \wedge LI^{int}_{old-new}$		p_{16}
(3.7)	$LI^{ext}_{old-new} \wedge LI^{int}_{old-new} \leftarrow CC_{new}$	(H.1)	p_2
(3.8)	$AP^{client}_{new} \wedge RP^{client}_{new} \leftarrow LO^{int}_{new}$	(H.1)	p_4
(3.9)	$AP^{server}_{new} \wedge RP^{server}_{new} \leftarrow LO^{ext}_{new} \wedge LO^{int}_{new}$	(H.1)	p_4
(3.10)	$LI^{int}_{old-new} \leftarrow ISS^{client}_{old} \wedge ISS^{server}_{old}$	(H.2)	p_8
(3.11)	$AP^{client}_{new} \wedge LI^{ext}_{old-new} \leftarrow ISS^{client}_{old}$	(H.2)	p_6
(3.12)	$AP^{server}_{new} \leftarrow ISS^{server}_{old}$	(H.2)	p_8
(3.13)	$UO^{int}_{old} \leftarrow ISS^{client}_{old} \wedge ISS^{server}_{old}$	(H.3)	p_8
(3.14)	$UO^{ext}_{old} \leftarrow ISS^{server}_{old}$	(H.3)	p_8

Table 3.3: Overview of all reconfiguration conditions that must be fulfilled to correctly execute local reconfigurations that involve components of a distributed network service. Column "h-l cond." specifies the high-level conditions from which some of these reconfiguration conditions are derived. Besides, the right column lists the place as from which the associated pre-condition is fulfilled.

3.5.4 Reconfiguration algorithm

We now present the algorithm that NeCoMan employs to conduct the local reconfiguration of distributed network services. For this we use Petri nets [51] as a modelling language to visualize the execution of the reconfiguration actions[15]. Note that the mathematic representation of Petri nets will not be employed. Instead, we use this representation to verify that this algorithm fulfills all reconfiguration conditions which have been defined in the previous subsections.

Figure 3.18 depicts the Petri net that models NeCoMan's algorithm for conducting local reconfigurations that involve distributed network services. Every transition

[15]Because NeCoMan does not execute reconfiguration actions in parallel to perform local reconfigurations, using Petri nets to model these reconfigurations provides only little added value. However, we use Petri nets anyway to be consistent with the representation of NeCoMan's algorithms for distributed reconfiguration. As we explain in Chapter 5, the latter coordinate the *concurrent* recomposition of multiple nodes.

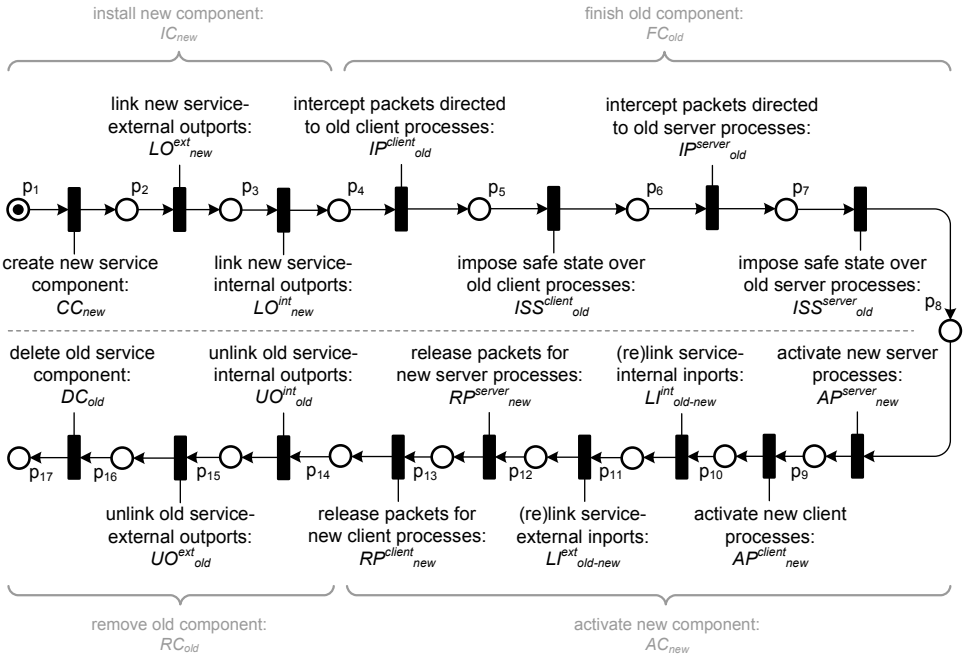


Figure 3.18: Petri net representation of NeCoMan's reconfiguration algorithm to conduct local recompositions that involve components of a distributed network service

in this model represents the execution of a single reconfiguration action. The algorithm first executes all reconfiguration actions related to the installation phase. Next, the reconfiguration actions to finish the old component are carried out. As soon as the old component is finished, all reconfiguration actions for activating the new component become initiated. This way the algorithm aims to limit the communication disruption that the reconfiguration causes[16]. Finally, the reconfiguration actions to remove the old components are executed. Appendix B illustrates this reconfiguration algorithm in more detail with the replacement of R_{old} with R_{new}.

Since no reconfiguration actions are executed in parallel, the initial marking of this model contains only one single token which is located in place p_1. Besides, we presume that every reconfiguration action becomes completed correctly – that is, the execution of a reconfiguration action will never fail. Therefore, when the single token reaches a specific place p_x, NeCoMan has correctly executed all reconfiguration

[16]To satisfy the requirement for limited reconfiguration overhead, the communication disruption that a reconfiguration causes must be kept to a minimum. This disruption results from intercepting packets to finish the old network service component. So, the period in which service continuity will be disrupted during a reconfiguration equals the time-interval between starting to finish the old component and having the new one activated.

place	**reconfiguration actions that are completed**	**place**	**reconfiguration actions that are completed**
p_1	none	p_{10}	$ac(p_9) \wedge AP_{new}^{client}$
p_2	CC_{new}	p_{11}	$ac(p_{10}) \wedge LI_{old-new}^{int}$
p_3	$ac(p_2) \wedge LO_{new}^{ext}$	p_{12}	$ac(p_{11}) \wedge LI_{old-new}^{ext}$
p_4	$ac(p_3) \wedge LO_{new}^{int}$	p_{13}	$ac(p_{12}) \wedge RP_{new}^{server}$
p_5	$ac(p_4) \wedge IP_{old}^{client}$	p_{14}	$ac(p_{13}) \wedge RP_{new}^{client}$
p_6	$ac(p_5) \wedge ISS_{old}^{client}$	p_{15}	$ac(p_{14}) \wedge UO_{old}^{int}$
p_7	$ac(p_6) \wedge IP_{old}^{server}$	p_{16}	$ac(p_{15}) \wedge UO_{old}^{ext}$
p_8	$ac(p_7) \wedge ISS_{old}^{server}$	p_{17}	$ac(p_{16}) \wedge DC_{old}$
p_9	$ac(p_8) \wedge AP_{new}^{server}$		

Table 3.4: NeCoMan's reconfiguration algorithm to conduct local recompositions that involve components of a distributed network service: definition of all places. Note that $ac(p_x)$ represents all reconfiguration actions that have been completed when the token reaches place p_x.

actions modelled by p_x's input transitions as well as by its ancestor transitions. To clarify all possible execution states, Table 3.4 lists the reconfiguration actions that NeCoMan has executed when the single token reaches each one of the places modelled in Figure 3.18.

This brings us to the correctness of this algorithm. Related to the algorithm's Petri net model, each one of the reconfiguration conditions listed in Table 3.3 formalizes a pre-condition to fire a transition. So, to verify that the presented algorithm conducts correct reconfigurations, we must check for every transition modelled in Figure 3.18 if all pre-conditions (that the associated reconfiguration conditions impose) are met. The execution of LO_{new}^{ext} and LO_{new}^{int}, for instance, can only be initiated when CC_{new} is finished, as dictated by reconfiguration condition (3.1). We conclude from Figure 3.18 and Table 3.4 that this condition is fulfilled as from reaching place p_2. Because p_2 is the input place of LO_{new}^{ext} and the ancestor place of LO_{new}^{int}, the algorithm meets this reconfiguration condition.

To illustrate that all other reconfiguration actions are fulfilled as well, the right column of Table 3.3 identifies for each reconfiguration condition the place as from which the associated pre-condition is fulfilled. This way, one can verify for each reconfiguration action that the associated conditions are met. Appendix C demonstrates in more detail that this is the case.

3.6 Local reconfigurations of isolated services

Besides replacing components that belong to a distributed network-service, NeCoMan also coordinates the local addition, replacement and removal of isolated network-service components. These components are self-contained, and thus encapsulate

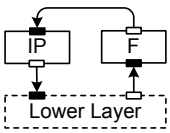


Figure 3.19: Protocol stack composition of a DiPS+ router before removing the filter component

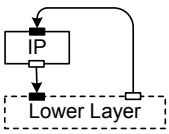


Figure 3.20: Protocol stack composition of a DiPS+ router after removing the filter component

only processes that do not collaborate with other ones to complete a service. Hence, the distinction between client and server processes, as well as between service-external and service-internal communication ports becomes irrelevant.

In the remainder of this section, we build up to the algorithm that NeCoMan uses to conduct these local reconfigurations. The structure of this section bears a close resemblance to Section 3.5. Subsection 3.6.1 first distinguishes the high-level reconfiguration phases of this algorithm and defines the associated reconfiguration conditions. Next, Subsection 3.6.2 presents for each of these reconfiguration phases the reconfiguration actions that the algorithm executes, as well as the associated reconfiguration conditions. After that, Subsection 3.6.3 completes the set of reconfiguration conditions that NeCoMan must fulfill by refining the high-level reconfiguration conditions defined in Subsection 3.6.1. Finally, Subsection 3.6.4 presents the algorithm itself, and indicates that this algorithm fulfills all required reconfiguration conditions.

All these steps are briefly illustrated by the removal of a filter component from a network node that is not congested anymore. Figures 3.19 and 3.20 sketch the protocol stack of the affected node before and after this removal has occurred as backing. Component F represents the filter component, while component IP encapsulates the routing functionality.

3.6.1 High-level reconfiguration phases and conditions

NeCoMan's second local reconfiguration algorithm comprises the same four phases as the previous algorithm does. In addition, reconfiguration conditions (H.1), (H.2), and (H.3) still apply as well.

3.6.2 Detailed overview of each reconfiguration phase

The implementation of these phases, however, is different form the previous algorithm. To explain this in more detail, we briefly discuss the reconfiguration actions that are executed for each phase. These actions slightly differ from the ones that the first basic reconfiguration algorithm executes.

Installation phase

Installing a new component still involves loading this component and binding its outports. Because for self-contained components the distinction between service-external and service-internal outports is irrelevant, NeCoMan implements this installation phase by executing reconfiguration actions CC_{new} and LO_{new}. With this, LO_{new} symbolizes a reconfiguration action for linking all outports of these new components[17]. Hence, we express the implementation of this installation phase as

$$IC_{new} \equiv CC_{new} \curlywedge LO_{new} \tag{P.5}$$

Note that when removing component F, these reconfiguration actions are redundant. The removal of a component obviously does not involve the installation of new components.

Besides, the associated reconfiguration condition that imposes to only bind these outports once their components have been created can be expressed as

$$LO_{new} \leftarrow CC_{new} \tag{3.15}$$

Finishing phase

Similar as for distributed network-service components, finishing the old (isolated) component involves instructing the node's reconfiguration support to intercept packets and to impose a safe state over this component. Because for self-contained components the distinction between client and server processes is irrelevant, NeCoMan implements this finishing phase by executing reconfiguration actions IP_{old} and ISS_{old}. With this, IP_{old} and ISS_{old} symbolize reconfiguration actions for intercepting packets and imposing a safe state over the affected component. We express the implementation of this finishing phase therefore as

$$FC_{old} \equiv IP_{old} \curlywedge ISS_{old} \tag{P.6}$$

To illustrate this finishing phase, consider again the removal of component F. In this example, finishing component F can be accomplished by intercepting the packets directed to F's single inport, and monitoring F until it has processed all accepted packets. Beside, the order in which both actions are executed must (again) be coordinated. That is, NeCoMan must first instruct the affected node to intercept packets before invoking it to impose a safe state. We denote this reconfiguration condition as

$$ISS_{old} \leftarrow IP_{old} \tag{3.16}$$

[17] CC_{new} is identical as for the previous algorithm.

Activation phase

Activating the new isolated component involves (1) rebinding all affected inports, (2) starting this component's active objects, and (3) resuming packets that have been intercepted to finish the old component. NeCoMan implements this activation phase by executing reconfiguration actions $LI_{old-new}$, AP_{new}, and RP_{new}, which are responsible for rebinding all affected inports, activating all encapsulated processes, and resuming all intercepted packets, respectively. Hence, we express the implementation of this activation phase as

$$AC_{new} \equiv LI_{old-new} \curlywedge AP_{new} \curlywedge RP_{new} \tag{P.7}$$

When removing component F, the activation phase reduces to rebinding inports and resuming intercepted packets. Besides, rebinding inports in this case involves (1) removing the connection between the lower layer's outport and the inport of component F, and (2) connecting the outport of the lower layer to the inport of *IP*.

Besides, to correctly activate a new component, the order in which the associated reconfiguration actions are executed must (again) be coordinated. That is, all affected inports must be bound correctly and the active objects must be started before resuming intercepted packets. We express this reconfiguration condition as

$$RP_{new} \leftarrow LI_{old-new} \wedge AP_{new} \tag{3.17}$$

Removal phase

Finally, removing the old component involves disconnecting all remaining (outport) bindings and deleting this component from the affected node. When removing component F, this involves unlinking the remaining connection between F's outport and the inport of component *IP*.

NeCoMan implements this removal phase by executing reconfiguration actions DC_{old} and UO_{old}, where UO_{old} represents a reconfiguration action for unlinking the old components' outports[18]. We express this removal phase as

$$RC_{old} \equiv DC_{old} \curlywedge UO_{old} \tag{P.8}$$

In addition, we express the associated reconfiguration condition which defines that NeCoMan can only delete a component after unlinking its outports as

$$DC_{old} \leftarrow UO_{old} \wedge LI_{old-new} \tag{3.18}$$

[18] DC_{old} is identical as for the previous algorithm.

3.6.3 Refining high-level reconfiguration conditions

Next, we refine reconfiguration conditions (H.1), (H.2), and (H.3) in terms of the reconfiguration actions that implement the four (high-level) reconfiguration phases. The method that has been applied to refine these conditions is similar as for the previous reconfiguration algorithm. Therefore, we only list the resulting reconfiguration conditions without elaborating on the refinement itself.

Installing new component before activation (H.1)

To refine condition (H.1), we substitute IC_{new} and AC_{new} for expressions (P.5) and (P.7), respectively. This results in condition

$$LI_{old-new} \wedge AP_{new} \wedge RP_{new} \leftarrow CC_{new} \wedge LO_{new} \tag{I.23}$$

After reducing (I.23), we conclude that NeCoMan meets reconfiguration condition (H.1) when it fulfills conditions (3.15), (3.19), and (3.20).

$$LI_{old-new} \leftarrow CC_{new} \tag{3.19}$$
$$RP_{new} \wedge AP_{new} \leftarrow LO_{new} \tag{3.20}$$

Finishing old component before activating new one (H.2)

Next, we refine reconfiguration condition (H.2). Replacing FC_{old} and AC_{new} in (H.2) by expressions (P.6) and (P.7) results in

$$LI_{old-new} \wedge AP_{new} \wedge RP_{new} \leftarrow IP_{old} \wedge ISS_{old} \tag{I.24}$$

After reducing this condition, we conclude that NeCoMan fulfills reconfiguration condition (H.2) when it meets conditions (3.16), (3.21), and (3.22).

$$LI_{old-new} \leftarrow IP_{old} \tag{3.21}$$
$$RP_{new} \wedge AP_{new} \leftarrow ISS_{old} \tag{3.22}$$

Finishing old component before removal (H.3)

Finally, we refine reconfiguration condition (H.3). Substituting FC_{old} and RC_{old} by expressions (P.6) and (P.8) results in

$$DC_{old} \wedge UO_{old} \leftarrow IP_{old} \wedge ISS_{old} \tag{I.25}$$

reconfiguration condition		h-l cond.	place
(3.15)	$LO_{new} \leftarrow CC_{new}$		p_2
(3.16)	$ISS_{old} \leftarrow IP_{old}$		p_4
(3.17)	$RP_{new} \leftarrow LI_{old-new} \wedge AP_{new}$		p_7
(3.18)	$DC_{old} \leftarrow UO_{old} \wedge LI_{old-new}$		p_9
(3.19)	$LI_{old-new} \leftarrow CC_{new}$	(H.1)	p_2
(3.20)	$RP_{new} \wedge AP_{new} \leftarrow LO_{new}$	(H.1)	p_3
(3.21)	$LI_{old-new} \leftarrow IP_{old}$	(H.2)	p_4
(3.22)	$RP_{new} \wedge AP_{new} \leftarrow ISS_{old}$	(H.2)	p_5
(3.23)	$UO_{old} \leftarrow ISS_{old}$	(H.3)	p_5

Table 3.5: Overview of all reconfiguration conditions that must be fulfilled to correctly execute local reconfigurations that involve isolated network services. Column "h-l cond." specifies the high-level conditions from which some of these reconfiguration conditions are derived. The right column lists the place as from which the associated pre-condition is fulfilled.

After reducing (I.25), we conclude that NeCoMan meets reconfiguration condition (H.3) when it fulfills conditions (3.16), (3.18) and (3.23).

$$UO_{old} \leftarrow ISS_{old} \tag{3.23}$$

Partial ordering of reconfiguration actions

By combining all these reconfiguration conditions (which are summarized in Table 3.5), we can specify the partial ordering of these reconfiguration actions. This ordering is illustrated in Figure 3.21.

3.6.4 Reconfiguration algorithm

Finally, we present the algorithm that NeCoMan uses to add, replace or remove isolated network-service components. Figure 3.22 depicts the Petri net that models this second local reconfiguration algorithm. Similar as for replacing components of a distributed service, this algorithm begins with installing the new service component. Next, all reconfiguration actions to finish the old service component are

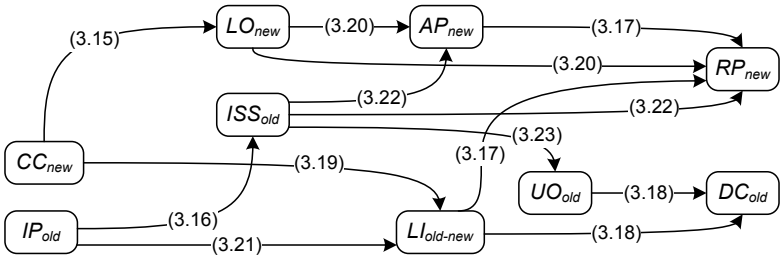


Figure 3.21: Overview of the partial ordering of NeCoMan's reconfiguration actions for carrying out local reconfigurations of isolated services

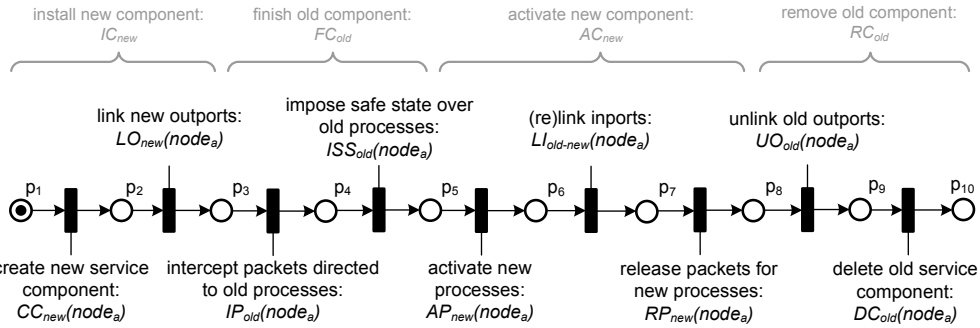


Figure 3.22: Petri net representation of NeCoMan's algorithm for conducting local reconfigurations that involve isolated network services

executed. After that, NeCoMan initiates the execution of the reconfiguration actions responsible for activating the new component. Finally, the removal actions are is carried out.

Table 3.5 lists all reconfiguration conditions that must be satisfied to correctly execute these reconfiguration actions. Besides, Table 3.5 also identifies for each of these reconfiguration condition the place (of the Petri net) as from which the associated pre-condition is fulfilled. From this, one can verify that the algorithm meets all required reconfiguration conditions, and thus conducts correct reconfigurations.

3.7 Conclusion

To conclude, we check both local reconfiguration algorithms against the four requirements that NeCoMan must fulfill to achieve its objectives.

3.7.1 Correct reconfigurations

As demonstrated in Sections 3.5.4 and 3.6.4, both local reconfiguration algorithms fulfill all imposed reconfiguration conditions, and thus conduct correct reconfigurations. Besides, recall that NeCoMan does not enforce a consistent execution state by itself, but instead expects the node's reconfiguration support to accomplish this.

3.7.2 Limited reconfiguration overhead

Limiting the reconfiguration overhead involves, among others, minimizing the communication disruption that a reconfiguration causes. This communication disruption results from intercepting packets to finish the old network service component. Hence, the period in which service continuity will be disrupted during reconfiguration equals the time-interval between starting to finish the old component and having the new one activated. To minimize this period, both reconfiguration algorithms (1) activate the new service as soon as the old one is finished, and (2) shorten the activation phase through partially connecting the new component into the protocol stack compositions during the installation phase.

These general-purpose reconfiguration algorithms, however, do not take into account the services' characteristics nor the reconfiguration semantics. For a number of services, therefore, the conducted reconfiguration will not be optimized. To limit the reconfiguration overhead if possible, NeCoMan must allow customizing both algorithms to exploit service specific characteristics and reconfiguration semantics.

3.7.3 Limited openness

NeCoMan aims to restrict the contribution needed to conduct a reconfiguration. We believe that network service developers should not be burdened with the development of ad hoc reconfiguration algorithms. Both local reconfiguration algorithms, therefore, coordinate the safe reconfiguration of a broad set of network services. For many services, however, these general-purpose reconfiguration algorithms must be tailored. To illustrate this, consider again the removal of a filter component. As explained in Subsection 3.6.2, various reconfiguration actions (including CC_{new}, LO_{new}, and AP_{new}) become redundant when carrying out this reconfiguration. As an additional example, Figure B.1 depicts the customized implementation of the (first) local reconfiguration algorithm to replace R_{old}.

To restrict the contribution needed from a service developer, NeCoMan must itself tailor each phase of the employed reconfiguration algorithm to the services' characteristics and the reconfiguration semantics. This way, the developer of the network service should only provide a declarative description of these characteristics and semantics (that is, besides a declarative description of the reconfiguration that must be executed) for NeCoMan to conduct a reconfiguration.

3.7.4 Reusability

Finally, to improve reusability, NeCoMan has been prepared to recompose various flow-oriented, component-based protocol stack architectures. To accomplish this, the implementation of each reconfiguration action restricts to the execution of (some of) the reconfiguration operations that the affected programmable node must provide. This has been (briefly) demonstrated in Section 3.5.2. In addition, Appendix B demonstrates in more detail that NeCoMan uses only the eight node operations specified in Section 3.3.1 to replace R_{old} with R_{new}.

> *We conclude that both local reconfiguration algorithms conduct safe local reconfigurations and that NeCoMan is prepared to be used on top of various flow-oriented, component-based protocol stack architectures. To fully comply with the second and third requirement, however, NeCoMan must customize both reconfiguration algorithms (if needed) to fully exploit the service specific characteristics and the reconfiguration semantics. We discuss these customizations in the next chapter.*

Chapter 4

Customizations to local reconfigurations

Since both algorithms presented in the previous chapter seek to conduct the local reconfiguration of a broad variety of services, they lack two important requirements that we impose on service reconfiguration in programmable networks. First, these basic reconfiguration algorithms do not guarantee limited reconfiguration overhead for every reconfiguration. Hence, in some cases the employed algorithm must be customized to optimize the reconfiguration process. Second, NeCoMan must restrict the contribution needed to conduct a correct and optimized reconfiguration. Besides a specification of the recomposition that must be executed, a reconfiguration description should only contain a declarative specification of the service characteristics and reconfiguration semantics. Based on these specifications, the NeCoMan middleware must be able to carry out a tailored reconfiguration.

To fulfill both requirements, NeCoMan incorporates an extensive set of customizations that apply to its local reconfiguration algorithms. Section 4.1 briefly lists these customizations and explains how they have been identified. Next, Sections 4.2 to 4.7 describe these customizations in full detail. Finally, Section 4.8 elaborates (again) on the four requirements that NeCoMan must fulfill to achieve its objectives.

4.1 Overview

Since we aim to limit the contribution needed from the service developer, NeCoMan should provide an extensive set of customizations to its basic algorithms. We defined these customizations by re-ordering and discarding all reconfiguration actions that both local algorithms include. From the resulting combinations, we selected those customizations that limit the reconfiguration overhead, and still yield a valid

reconfiguration – that is, given that some additional pre-conditions are fulfilled.

A first (resulting) customization involves switching the order of the activation and finishing phase. Section 4.2 explains how NeCoMan customizes its local reconfiguration algorithms when this customization is applicable. A next customization involves discarding all finishing actions. This customization is presented in Section 4.3. As will be clarified soon, both customizations affect the high-level reconfiguration conditions that must be fulfilled.

The other customizations, in contrast, do not affect high-level reconfiguration conditions. Instead, these customizations include only omitting reconfiguration actions that are redundant for a particular reconfiguration. Section 4.4, for instance, explains how NeCoMan tailors the employed reconfiguration algorithm when the new service components do not use active objects. Next, Section 4.5 discusses which reconfiguration actions are omitted when the affected components employ only client or server processes, instead of both. Section 4.6 then explains how NeCoMan customizes its local reconfiguration algorithm for distributed services when the affected components expose only service-internal inports or outports, instead of both. After that, Section 4.7 discusses how NeCoMan tailors the algorithm that it uses for conducting reconfigurations of isolated services when these reconfigurations involve service addition or removal instead of replacement.

All these customizations are presented in a similar fashion. We first specify the pre-condition(s) that must be fulfilled to safely apply a specific customization. Next, we explain its impact by identifying all changes that it causes. These include the high-level conditions that are affected and the reconfiguration actions that become redundant. After that, we discuss the effect of these changes on the reconfiguration conditions. Furthermore, for the first two customizations (which involve activating the new service before finishing the old one, and discarding the finishing actions, respectively) we also illustrate the partial ordering of the (remaining) reconfiguration actions as well as the resulting reconfiguration algorithm. In addition, Appendix E evaluates the effect of these two customizations on the reconfiguration overhead.

Before focussing on the customizations themselves, we briefly illustrate how we identified the reconfiguration conditions that a customization changes. When a customization changes a high-level condition (which is the case for the first two customizations), then all reconfiguration conditions derived from this high-level condition become redundant. If a customization includes discarding condition (H.2), for instance, we can deduce from Tables 3.3 and 3.5 that reconfiguration conditions (3.10), (3.11), (3.12), (3.21) and (3.22) are of no use anymore. In addition, when a customization involves discarding a specific reconfiguration action, then the partial ordering sketched in Figures 3.17 and 3.21 assists in defining the safety conditions that become affected. When a customization involves omitting the execution of $LI^{int}_{old-new}$, for instance, we can deduce from Figure 3.17 that conditions (3.4), (3.5), (3.6), (3.7) and (3.10) become affected.

As will be illustrated soon, the safety conditions that a customization affects must be replaced with new ones (which are customization specific). Note, however,

that we will not discuss in detail how these conditions have come about to avoid losing focus. Besides, the principles to determine these new reconfiguration conditions are identical to those applied for defining the conditions that have been presented in the previous chapter.

4.2 Activate before finishing

The first customization that applies to both local reconfiguration algorithms involves switching the order of the finishing and activating phases such that the new network service component becomes activated before the old one is finished. This enables NeCoMan to drive the old service components to a reconfiguration-safe state while the new service is already operational. Because finishing a network service component can be very time-consuming, this way communication disruption will be reduced.

4.2.1 Local reconfigurations of distributed services

We first explain how this customization applies to NeCoMan's algorithm for replacing components of a distributed service.

Pre-conditions

Three pre-conditions must be fulfilled to safely switch the order of the activation and finishing phase. These include that

1. the old component is stateless,
2. the new service component is able to process all ongoing protocol-transactions, and
3. the network tolerates packet re-ordering.

1) Stateless components. When the old network service component is stateless, its state-information is irrelevant to other network components. Such stateless components, therefore, do not have to be driven to a consistent execution state before their new counterpart can be activated safely. When replacing a compression component, for instance, packets can be re-directed to the new compression component immediately instead of first waiting until the old component has processed all accepted packets. Because this compression component is stateless, no other components depend on its execution state, and thus consistency will not be compromised.

2) Ongoing protocol-transactions. When NeCoMan activates the new component before finishing the old one, there is no knowledge about the status of ongoing protocol-transactions at the moment when the new component is brought into use. Hence, to safely apply this customization, the new component must be able to continue processing all ongoing protocol-transactions.

3) Packet re-ordering tolerated. Besides, the new version of a component can only be activated before the old version is finished when packet re-ordering is tolerated. When a new component is brought into use before the old one has processed all accepted packets, both components (temporarily) execute in parallel during reconfiguration. Consequently, the packets that both versions process will most likely be shuffled. NeCoMan, therefore, can only apply this customization when packet re-ordering does not compromise the correct functioning of the network or its applications. This is the case, for instance, when a stateless component (such as the affected compression component) operates in a TCP network, which guarantees that packets always arrive at their destination in the same order as they were sent.

Modifications

Switching the order of the activation and finishing phases causes the following changes.

1. First, reconfiguration condition (H.2) does not apply anymore. Recall that this condition dictates to finish the old component before activating the new one so as to initialize the latter in a state that is consistent with the rest of the network. Because activating a component before finishing the old one can only be accomplished when stateless components are involved, however, this high-level reconfiguration condition can safely be discarded. To be precise, we replace condition (H.2) with reconfiguration condition (H.4)

$$FC_{old} \leftarrow AC_{new} \tag{H.4}$$

   which imposes to only finish the old network service component when the new one is brought into use. Note that this reconfiguration condition does not enforce a correct reconfiguration, but instead seeks to limit the communication disruption that a reconfiguration causes.

2. Second, reconfiguration actions RP^{client}_{new} and RP^{server}_{new} become redundant. Activating a new (compression) component before the old one is finished does not require intercepting and resuming packets. Instead, this only requires rebinding all affected inports and starting the active objects of the new compression component (if any). So, when NeCoMan applies this customization, the implementation of its activation phase does not include actions RP^{client}_{new}

and RP_{new}^{server} anymore. Hence, we express this customized activation phase as

$$AC_{new} \equiv LI_{old-new}^{int} \curlywedge LI_{old-new}^{ext} \curlywedge AP_{new}^{client} \curlywedge AP_{new}^{server} \tag{P.9}$$

3. Besides, reconfiguration actions IP_{old}^{client} and IP_{old}^{server} become redundant as well. This is because finishing the old service component does not involve intercepting packets anymore. Once the new (compression) compression component is activated, all affected inports are (re-)bound such that packets are delivered exclusively to the new component. The old (compression) component, therefore, does not have to be prevented anymore from accepting new packets to bring about a reconfiguration-safe state. So, when activating a new component before finishing the old one, the implementation of the finishing phase does not include IP_{old}^{client} and IP_{old}^{server} anymore. Hence, we express this customized finishing phase as

$$FC_{old} \equiv ISS_{old}^{client} \curlywedge ISS_{old}^{server} \tag{P.10}$$

4. Finally, the ordering defined by conditions (I.5) and (I.6) can be discarded as well when the activation and finishing phases are switched[1]. These conditions define that the new component's outports must be bound correctly before starting its active objects. As we discussed in Section 3.5.3, this is required to manage when the node's reconfiguration support does not impose a safe state by monitoring the affected processes, but instead deactivates these processes immediately and transfers their execution state towards the new processes. After executing AP_{new}^{client} and AP_{new}^{server} in that case, the new component may continue processing (ongoing) requests that the old component had already accepted.

   When NeCoMan activates the new component before finishing the old one, however, this new (stateless) component becomes initialized with a default execution state. Hence, this component will not continue processing (ongoing) requests once its active objects are started. Because of this, conditions (I.5) and (I.6) are of no use anymore.

Result

The above-mentioned changes affect many of the reconfiguration conditions listed in Table 3.3. By discarding condition (H.2), reconfiguration conditions (3.10), (3.11), and (3.12) are of no use anymore. In addition, because IP_{old}^{client} and IP_{old}^{server}

[1] Recall that conditions (I.5) and (I.6) are part of safety conditions (3.8) and (3.9), as explained in Section 3.5.3 page 74.

have become redundant, reconfiguration conditions (3.2) and (3.3) do not apply anymore either (this can be deduced from Figure 3.17). Similarly, reconfiguration conditions (3.4) and (3.5) are redundant because RP_{new}^{client} and RP_{new}^{server} will not be executed anymore. Finally, reconfiguration conditions (3.8) and (3.9) are of no use anymore either because (1) conditions (I.5) and (I.6) do not apply any longer, and (2) RP_{new}^{client} and RP_{new}^{server} are discarded.

These affected reconfiguration conditions become replaced with conditions (4.1) to (4.5), which define the new pre-conditions that NeCoMan must fulfill to correctly initiating the execution of AP_{new}^{client}, AP_{new}^{server}, $LI_{old-new}^{int}$, $LI_{old-new}^{ext}$, ISS_{old}^{client} and ISS_{old}^{server} when it activates the new component before finishing the old one.

$$AP_{new}^{client} \wedge AP_{new}^{server} \leftarrow CC_{new} \quad (4.1)$$

$$LI_{old-new}^{int} \leftarrow AP_{new}^{client} \wedge AP_{new}^{server} \wedge LO_{new}^{int} \wedge LO_{new}^{ext} \quad (4.2)$$

$$LI_{old-new}^{ext} \leftarrow AP_{new}^{client} \wedge LO_{new}^{int} \wedge LI_{old-new}^{int} \quad (4.3)$$

$$ISS_{old}^{client} \leftarrow LI_{old-new}^{int} \wedge LI_{old-new}^{ext} \quad (4.4)$$

$$ISS_{old}^{server} \leftarrow LI_{old-new}^{int} \quad (4.5)$$

Condition (4.1) imposes to only activate the new component's client and server processes once this component has been created. Condition (4.2) defines to redirect packet flows from the old towards the new component's service-internal inports once the new component is prepared to process these packets – that is, after starting its active objects and binding its outports. Similarly, condition (4.3) imposes to only redirect packet flows towards the new component's service-external inports if the new component is prepared to handle these service requests. This requires for (1) the active objects of the new client process to be started[2], and (2) the new component's service-internal inports and outports to be bound. Finally, conditions (4.4) and (4.5) result from refining high-level condition (H.4). Condition (4.4) specifies to only start finishing the old component's client processes once packet flows are redirected towards the (service-external and service-internal) inports of the new component. Similarly, condition (4.5) imposes to only finish the old server processes if packets are redirected to the service-internal inports of the new component[3].

To conclude, Table 4.1 summarizes all new and remaining reconfiguration conditions that NeCoMan must fulfill to correctly activate the new component before finishing the old one. The resulting partial ordering of reconfiguration actions is sketched in Figure 4.1. In addition, Figure 4.2 depicts the Petri net that models NeCoMan's local reconfiguration algorithm for distributed services after applying this customization. Note that the right column of Table 4.1 specifies for each reconfiguration condition the place (of this Petri net model) as from which the associated pre-condition is fulfilled. This enables to verify that the algorithm modelled in Figure 4.2 meets all required reconfiguration conditions.

[2]Recall that only client processes expose service-external inports.
[3]Recall that server processes do not expose service-external inports.

	reconfiguration condition	place
(3.1)	$LO^{ext}_{new} \wedge LO^{int}_{new} \leftarrow CC_{new}$	p_2
(3.6)	$DC_{old} \leftarrow UO^{ext}_{old} \wedge UO^{int}_{old} \wedge LI^{ext}_{old-new} \wedge LI^{int}_{old-new}$	p_{12}
(3.7)	$LI^{ext}_{old-new} \wedge LI^{int}_{old-new} \leftarrow CC_{new}$	p_2
(3.13)	$UO^{int}_{old} \leftarrow ISS^{client}_{old} \wedge ISS^{server}_{old}$	p_{10}
(3.14)	$UO^{ext}_{old} \leftarrow ISS^{server}_{old}$	p_{10}
(4.1)	$AP^{client}_{new} \wedge AP^{server}_{new} \leftarrow CC_{new}$	p_2
(4.2)	$LI^{int}_{old-new} \leftarrow AP^{client}_{new} \wedge AP^{server}_{new} \wedge LO^{int}_{new} \wedge LO^{ext}_{new}$	p_6
(4.3)	$LI^{ext}_{old-new} \leftarrow AP^{client}_{new} \wedge LO^{int}_{new} \wedge LI^{int}_{old-new}$	p_7
(4.4)	$ISS^{client}_{old} \leftarrow LI^{int}_{old-new} \wedge LI^{ext}_{old-new}$	p_8
(4.5)	$ISS^{server}_{old} \leftarrow LI^{int}_{old-new}$	p_7

Table 4.1: Customization of NeCoMan's algorithm to replace a component of a distributed service: overview of all reconfiguration conditions that must be fulfilled when activating the new component before finishing the old one.

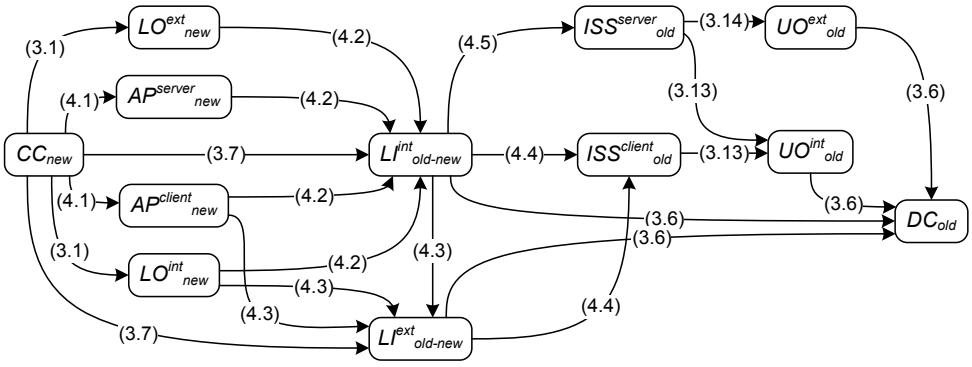


Figure 4.1: Customization of NeCoMan's algorithm to replace a component of a distributed service: overview of the partial ordering of reconfiguration actions that the reconfiguration conditions listed in Table 4.1 define.

4.2.2 Local reconfigurations of isolated services

Activating the new component before finishing the old one can be applied as well when adding, replacing or removing isolated services.

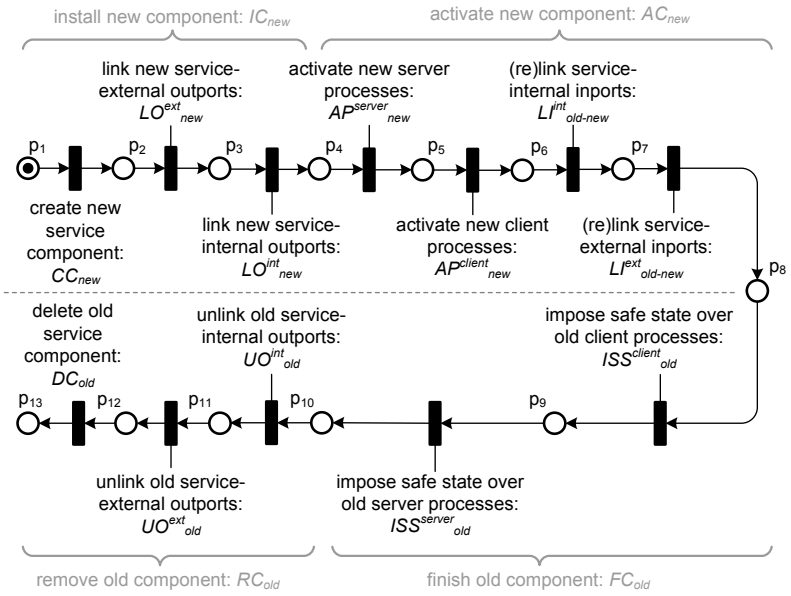


Figure 4.2: Customization of NeCoMan's algorithm to replace a component of a distributed service: Petri net representation of the resulting algorithm when activating the new component before finishing the old one.

Pre-conditions

The pre-conditions to apply this customization are similar as when distributed services are involved. Because isolated services do not participate in protocol-transactions, however, the second pre-condition becomes redundant[4]. Hence, NeCoMan can safely activate a new isolated service before finishing the old one if (1) the old component is stateless and (2) packet re-ordering is tolerated.

Modifications

Switching the order of the activation and finishing phase causes the following changes to local reconfigurations of isolated services. First, condition (H.2) becomes replaced with (H.4). Second, activating a new component does not involve the execution of RP_{new} anymore. NeCoMan therefore implements this customized activation phase as follows:

$$AC_{new} \equiv LI_{old-new} \curlywedge AP_{new} \tag{P.11}$$

[4]this pre-condition defines that the new component must be able to process all ongoing protocol-transactions

Finally, there is no need anymore to intercept packets directed to the old component so as to finish the latter. Once all inports are bound correctly, no more packets will be delivered to the old component's inports. Hence, NeCoMan does not include IP_{old} anymore to implement this customized finishing phase. We express the implementation of this customized finishing phase as

$$FC_{old} \equiv ISS_{old} \tag{P.12}$$

Result

These changes affect various reconfiguration conditions. By discarding condition (H.2), reconfiguration conditions (3.21) and (3.22) are of no use anymore. In addition, because IP_{old} has become redundant, reconfiguration condition (3.16) does not apply anymore either. Finally, reconfiguration conditions (3.17) and (3.20) are of no use anymore either. Instead, these conditions become replaced with conditions (4.6), (4.7) and (4.8).

$$AP_{new} \leftarrow CC_{new} \tag{4.6}$$
$$LI_{old-new} \leftarrow AP_{new} \wedge LO_{new} \tag{4.7}$$
$$ISS_{old} \leftarrow LI_{old-new} \tag{4.8}$$

Condition (4.6) imposes to only activate the new component's processes once this component has been created. Condition (4.7) defines to redirect packet flows from the old towards the new component's inports after starting the active objects that the latter includes and binding its outports. Finally, condition (4.8) specifies to only start finishing the old component's processes once packet flows are redirected towards the inports of the new component. This condition results from refining high-level condition (H.4).

To conclude, Table 4.2 summarizes all new and remaining reconfiguration conditions that NeCoMan must fulfill when it applies this customization. The resulting partial ordering of reconfiguration actions is sketched in Figure 4.3. Furthermore, Figure 4.4 depicts the Petri net that models NeCoMan's local reconfiguration algorithm for isolated services after applying this customization.

4.3 No finishing

A next customization that applies to both local reconfiguration algorithms involves omitting their finishing actions. Driving a network service component to a reconfiguration-safe state may be very time-consuming. When the network or the new service component is able to cope with inconsistencies, however, there is no need for the reconfiguration middleware to preserve consistency. In that case, NeCoMan can safely omit its finishing actions, thus reducing the overhead that a reconfiguration causes.

	reconfiguration condition	place
(3.15)	$LO_{new} \leftarrow CC_{new}$	p_2
(3.18)	$DC_{old} \leftarrow UO_{old} \wedge LI_{old-new}$	p_7
(3.19)	$LI_{old-new} \leftarrow CC_{new}$	p_2
(3.23)	$UO_{old} \leftarrow ISS_{old}$	p_6
(4.6)	$AP_{new} \leftarrow CC_{new}$	p_2
(4.7)	$LI_{old-new} \leftarrow AP_{new} \wedge LO_{new}$	p_4
(4.8)	$ISS_{old} \leftarrow LI_{old-new}$	p_5

Table 4.2: Customization of NeCoMan's algorithm to add, replace or remove isolated services: overview of all reconfiguration conditions that must be fulfilled when activating the new component before the old one is finished.

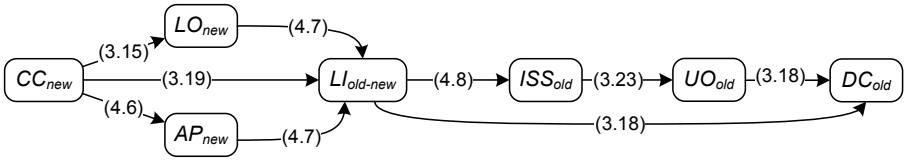


Figure 4.3: Customization of NeCoMan's algorithm to add, replace or remove isolated services: overview of the partial ordering of reconfiguration actions that the reconfiguration conditions listed in Table 4.2 define.

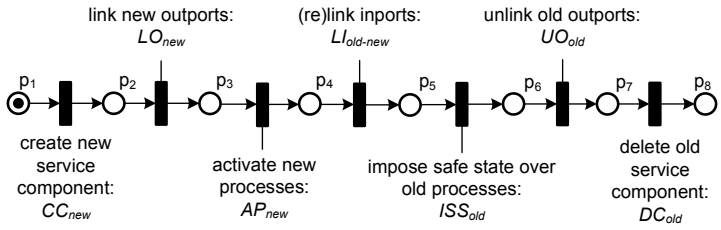


Figure 4.4: Customization of NeCoMan's algorithm to add, replace or remove isolated services: Petri net representation of the resulting algorithm when activating the new component before finishing the old one.

4.3.1 Local reconfigurations of distributed services

Pre-conditions

To apply this customization, the network or the new service component must be able to recover from or tolerate inconsistencies that may occur in the course of

a reconfiguration. These inconsistencies come about when packets get lost, when ongoing protocol-transactions cannot be further processed, or when inconsistent execution states come about. NeCoMan, therefore, can only safely omit its finishing actions when

1. the affected service components operate in a best-effort network,

2. the new service component is able to process all ongoing protocol-transactions, and

3. inconsistent execution states (if any) do not compromise the correct functioning of the network.

1) Best-effort network. Best-effort networks (which employ protocols such as IP and UDP) do not guarantee packet delivery. Applications and higher-layer protocols that employ these networks, therefore, must be able to cope with unreliable packet delivery. If this is the case, NeCoMan can safely remove the old component immediately instead of first monitoring until all accepted packets are processed. The packet loss that this may bring about will be dealt with, if necessary, by the applications or by higher layer protocols using this best-effort network.

2) Ongoing protocol-transactions. When NeCoMan discards its finishing actions, there is no knowledge about the status of ongoing protocol-transactions at the moment when the new component is brought into use. This is similar as when NeCoMan activates the new component before finishing the old one. The new component therefore must be able to continue processing all ongoing protocol-transactions. If this is not the case, finishing actions cannot be omitted without breaking the network's correct functioning.

3) Inconsistent execution states. In addition, NeCoMan can only omit its finishing actions when this causes no inconsistent execution states or when the network restores from or tolerates inconsistent execution states. This requirement is always fulfilled when replacing stateless components (such as a compression component). Because these components do not share their execution state with other network components, there is no need to bring about a consistent execution state before removing them[5].

Modifications

This customization brings along the following changes:

[5] that is, given that they operate in a best-effort environment and are able to accept and process all pending service request

1. Reconfiguration conditions (H.2) and (H.3) become redundant and are therefore discarded. Instead, we replace these (high-level) conditions with (H.5)

$$RC_{old} \leftarrow AC_{new} \tag{H.5}$$

which imposes to only remove the old network service component when the new one is activated. Although this condition is not essential to conduct a correct reconfiguration, it is required to minimize communication disruption. If the old compression component becomes removed (long) before the new one is activated, then a reconfiguration causes significant overhead. This would undo the advantage of omitting finishing actions.

2. Omitting all finishing actions implies that NeCoMan discards the execution of IP_{old}^{client}, IP_{old}^{server}, ISS_{old}^{server}, and ISS_{old}^{client}. Besides, since packets are not intercepted during reconfiguration, there is no need to resume intercepted packets either. So, the execution of RP_{new}^{client} and RP_{new}^{server} will also be discarded. Hence, in this case NeCoMan implements its activation phase as specified by (P.9).

3. Conditions (I.5) and (I.6) are not relevant anymore either when NeCoMan omits the execution of all its finishing actions. These conditions only apply to reconfigurations that employ state-transfer to reach a reconfiguration-safe state (instead of monitoring the affected processes). Because omitting all finishing actions implies that no safe state will be reached at all, these conditions are of no use anymore.

Result

The above-mentioned changes affect reconfiguration conditions (3.10), (3.11), (3.12), (3.13), (3.14), (3.2), (3.3), (3.4), (3.5), (3.8) and (3.9). These reconfiguration conditions become replaced with conditions (4.1), (4.2), (4.3), (4.9) and (4.10), which define the new pre-conditions that NeCoMan must fulfill to correctly initiate the execution of AP_{new}^{client}, AP_{new}^{server}, $LI_{old-new}^{int}$ and $LI_{old-new}^{ext}$ when it discards all finishing actions. Note that conditions (4.1), (4.2) and (4.3) originate from the previous customization (which involves activating the new service before finishing the old one).

$$UO_{old}^{int} \leftarrow LI_{old-new}^{int} \wedge LI_{old-new}^{ext} \tag{4.9}$$

$$UO_{old}^{ext} \leftarrow LI_{old-new}^{int} \tag{4.10}$$

Conditions (4.9) and (4.10), in contrast, result from refining high-level condition (H.5). Condition (4.9) specifies to only unlink a component's service-internal outports once this component will not be invoked anymore – that is, after redirecting packet flows towards the (service-external and service-internal) inports of the new component. Similarly, condition (4.10) imposes to only unlink a component's

	reconfiguration condition	place
(3.1)	$LO^{ext}_{new} \wedge LO^{int}_{new} \leftarrow CC_{new}$	p_2
(3.6)	$DC_{old} \leftarrow UO^{ext}_{old} \wedge UO^{int}_{old} \wedge LI^{ext}_{old-new} \wedge LI^{int}_{old-new}$	p_{10}
(3.7)	$LI^{ext}_{old-new} \wedge LI^{int}_{old-new} \leftarrow CC_{new}$	p_2
(4.1)	$AP^{client}_{new} \wedge AP^{server}_{new} \leftarrow CC_{new}$	p_2
(4.2)	$LI^{int}_{old-new} \leftarrow AP^{client}_{new} \wedge AP^{server}_{new} \wedge LO^{ext}_{new} \wedge LO^{int}_{new}$	p_6
(4.3)	$LI^{ext}_{old-new} \leftarrow AP^{client}_{new} \wedge LO^{int}_{new} \wedge LI^{int}_{old-new}$	p_7
(4.9)	$UO^{int}_{old} \leftarrow LI^{int}_{old-new} \wedge LI^{ext}_{old-new}$	p_8
(4.10)	$UO^{ext}_{old} \leftarrow LI^{int}_{old-new}$	p_7

Table 4.3: Customization of NeCoMan's algorithm to replace a component of a distributed service: overview of all reconfiguration conditions that must be fulfilled when finishing actions are omitted.

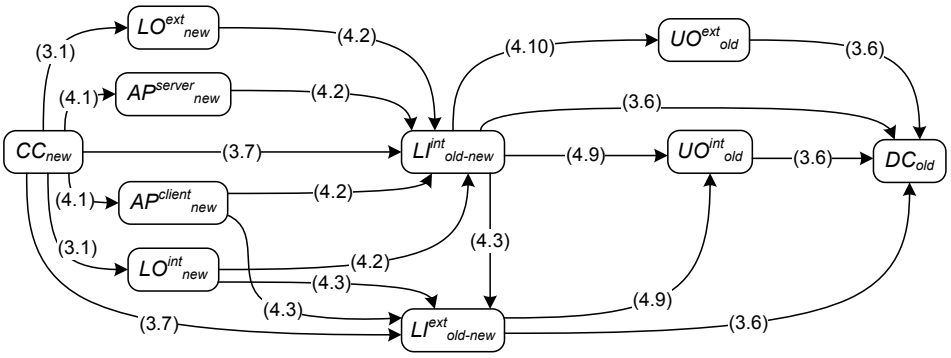


Figure 4.5: Customization of NeCoMan's algorithm to replace a component of a distributed service: overview of the partial ordering of reconfiguration actions that the reconfiguration conditions listed in Table 4.3 define.

service-external outports once its service-internal inports will not be invoked anymore. This is because (1) only server processes expose service-external outports, and (2) these server processes can only be invoked via their service-internal inports.

To conclude, Table 4.3 summarizes all new and remaining reconfiguration conditions that NeCoMan must fulfill when it applies this customization. The resulting partial ordering of reconfiguration actions is sketched in Figure 4.5. Furthermore, Figure 4.6 depicts the Petri net that models NeCoMan's local reconfiguration algorithm for distributed services after applying this customization.

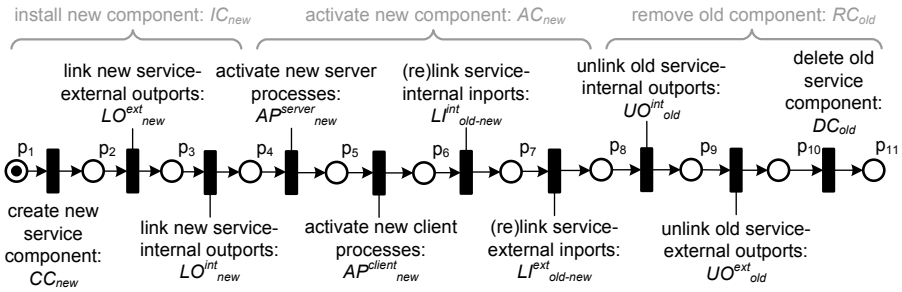


Figure 4.6: Customization of NeCoMan's algorithm to replace a component of a distributed service: Petri net representation of the resulting algorithm when all finishing actions are omitted.

4.3.2 Local reconfigurations of isolated services

Pre-conditions

The pre-conditions to apply this customization when isolated services are involved are similar as for distributed network services. The requirement for the new component to be able to continue processing all ongoing protocol-transaction, however, can be discarded. This is because self-contained network services do not communicate by protocol-transactions, as protocol-transactions expose dependencies with other components.

So, to safely omit all finishing actions, (1) the affected service components must operate in a best-effort network, and (2) inconsistent execution states must be tolerated or dealt with by the network itself. Note that the last of both requirements is always fulfilled when a reconfiguration involves the *addition* of a new isolated service component, for instance when adding a filter component to a congested node. This reconfiguration is similar to the replacement of a dummy service (which contains no functionality) by the new filter service. Because a dummy service is always in a consistent execution state, there is no need to finish this service before activating the newly added filter component.

Modifications

Similar to local reconfigurations of distributed services, omitting all finishing actions imposes several adjustments to the algorithm for conducting local reconfigurations of isolated services. This results, among others, from replacing conditions (H.2) and (H.3) by condition (H.5). Besides, omitting all finishing actions implies that NeCoMan discards the execution of IP_{old} and ISS_{old}. If this is the case, the execution of RP_{new} becomes redundant and will be omitted as well. Hence, NeCoMan implements the activation phase as defined by expression (P.11).

These changes again affect various reconfiguration conditions, including (3.21),

	reconfiguration condition	place
(3.15)	$LO_{new} \leftarrow CC_{new}$	p_2
(3.18)	$DC_{old} \leftarrow UO_{old} \wedge LI_{old-new}$	p_6
(3.19)	$LI_{old-new} \leftarrow CC_{new}$	p_2
(4.6)	$AP_{new} \leftarrow CC_{new}$	p_2
(4.7)	$LI_{old-new} \leftarrow AP_{new} \wedge LO_{new}$	p_4
(4.11)	$UO_{old} \leftarrow LI_{old-new}$	p_5

Table 4.4: Customization of NeCoMan's algorithm to conduct local reconfigurations of isolated services: overview of all reconfiguration conditions that must be fulfilled when all finishing actions are omitted.

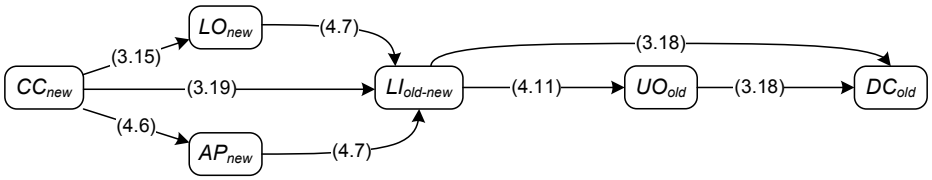


Figure 4.7: Customization of NeCoMan's algorithm to conduct local reconfigurations of isolated services: overview of the partial ordering of reconfiguration actions that the reconfiguration conditions listed in Table 4.4 define.

(3.22), (3.23), (3.16), (3.17), and (3.20). These conditions become replaced with conditions (4.6), (4.7) and (4.11)

$$UO_{old} \quad \leftarrow \quad LI_{old-new} \tag{4.11}$$

where condition (4.11) specifies to only unlink a component's outports after redirecting packet flows towards the inports of the new component. This reconfiguration condition result from refining high-level condition (H.5).

Result

Table 4.4 summarizes all new and remaining reconfiguration conditions that must be fulfilled. The resulting partial ordering of reconfiguration actions is sketched in Figure 4.7. Furthermore, Figure 4.8 depicts the Petri net that models NeCoMan's local reconfiguration algorithm for isolated services after applying this customization.

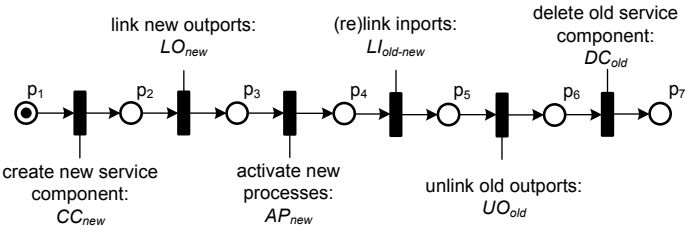


Figure 4.8: Customization of NeCoMan's algorithm to conduct local reconfigurations of isolated services: Petri net representation of the local reconfiguration algorithm when all finishing actions are omitted.

4.4 No active objects

The following (trivial) customization involves the absence of active objects. Obviously, NeCoMan should only invoke the affected node to start active objects if the new service processes employ active objects. If this is not the case, NeCoMan can omit the reconfiguration actions that involve initiating active objects – that is, AP_{new}^{client} and/or AP_{new}^{server}. This customization can be applied to both local reconfiguration algorithms, as well as to the algorithms that result from applying one of the previous customizations to these algorithms.

4.4.1 Local reconfigurations of distributed services

Pre-conditions

The only (trivial) pre-condition that must be fulfilled to omit the execution of AP_{new}^{client} and/or AP_{new}^{server} includes that the new component's client and/or server processes do not employ active objects.

Modifications

To illustrate the effect of this customization, Table D.1 lists all reconfiguration conditions that are changed when AP_{new}^{client} becomes redundant. For each of these conditions, the resulting reconfiguration conditions (if any) are presented in the right column of this table. Similarly, Table D.2 lists the reconfiguration conditions that are affected when omitting AP_{new}^{server}.

4.4.2 Local reconfigurations of isolated services

Pre-conditions

NeCoMan can only omit the execution of AP_{new} when the new component does not employ active objects. Note that this is always the case when a reconfiguration

involves *removing* an isolated component from a node's protocol stack composition.

Modifications

When the new component does not employ active objects, NeCoMan discards AP_{new}. Table D.3 lists all reconfiguration conditions that this customization changes.

4.5 Only client or server processes instead of both

The next customization relates to the client and server processes that a component encapsulates. NeCoMan omits some of its reconfiguration actions when the old and new version of the affected component encapsulate only client or server processes, instead of both. This customization, therefore, can only be applied to NeCoMan's local reconfiguration algorithm for distributed services (not to the algorithm for isolated services), as well as to the algorithms that result from previous customizations to this basic algorithm.

Pre-conditions

No additional pre-conditions must be fulfilled to apply this customization – that is, besides the obvious condition that the affected component should only encapsulate client or server processes.

Modifications

When the affected components encapsulate only client processes, NeCoMan (obviously) omits the execution of IP_{old}^{server}, ISS_{old}^{server}, AP_{new}^{server}, and RP_{new}^{server}. In addition, LO_{new}^{ext} and UO_{old}^{ext} become redundant as well and will be omitted. This is because a client process exposes no service-external outports according to our component model[6]. To illustrate the impact of this customization, Table D.4 lists all reconfiguration conditions that are changed when a reconfiguration involves only client processes.

When the affected components encapsulate only server processes, NeCoMan discards the execution of IP_{old}^{client}, RP_{new}^{client}, ISS_{old}^{client}, and AP_{new}^{client}. Besides, in this case also $LI_{old-new}^{ext}$ becomes redundant and will be omitted. This is because, according to our component model, a server process does not expose service-external inports. Table D.5 presents all reconfiguration conditions that are changed when a reconfiguration involves only server processes.

[6] as explained in Section 2.5

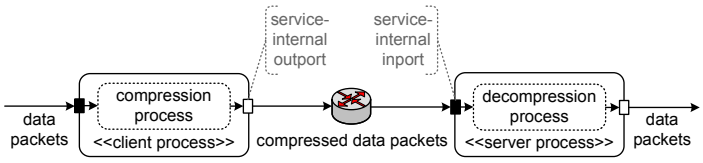


Figure 4.9: A compression service.

4.6 Only service-internal inports or outports instead of both

The next customization is targeted at omitting reconfiguration actions that involve *service-internal* inports and outports. To be precise, this customization seeks to discard the execution of LO^{int}_{new}, UO^{int}_{old}, or $LI^{int}_{old-new}$, if possible. Recall that collaborating client and server processes use these service-internal communication ports to exchange packets during the execution of a protocol-transaction. This customization, therefore, can only be applied to NeCoMan's local reconfiguration algorithm for distributed services, as well as to the algorithms that result from previous customizations to this basic algorithm.

Pre-conditions

The execution of LO^{int}_{new}, UO^{int}_{old} or $LI^{int}_{old-new}$ can only be discarded when the affected components

1. encapsulate only client or server processes, instead of both, and
2. communicate by a unidirectional communication protocol.

To illustrate this, consider the (independent) replacement of a compression and a decompression component. As illustrated in Figure 4.9, these components encapsulate only a client or a server process – that is, to compress or decompress accepted packets, respectively. Besides, these processes communicate to each other by a unidirectional communication protocol. Hence, the compression component does not expose service-internal inports, and the decompression component does not provide service-internal outports.

Modifications

Consequently, replacing the compression component does not require the execution of $LI^{int}_{old-new}$. In addition, when replacing the decompression component, LO^{int}_{new} and UO^{int}_{old} are redundant and will be discarded. To illustrate the impact of these customizations, Table D.6 lists all reconfiguration conditions that are changed when $LI^{int}_{old-new}$ becomes redundant on node x. Besides, Table D.7 lists all reconfiguration conditions that are changed when LO^{int}_{new} and UO^{int}_{old} are redundant on that node x.

4.7 Service addition or removal

The last customizations are targeted at the reconfiguration type – that is, service addition, removal, or replacement. To be precise, NeCoMan applies two different customizations in case of service addition and removal, respectively. Recall that NeCoMan's local reconfiguration algorithm for distributed services supports only component *replacement*, so as to avoid breaking structural integrity[7]. Both customizations, therefore, only apply to NeCoMan's local reconfiguration algorithm for isolated services.

Pre-conditions

No additional pre-conditions must be fulfilled for NeCoMan to safely apply these customizations.

Modifications

NeCoMan applies a different customization depending on whether service addition or removal is involved.

a) Service addition. As already explained in Section 4.3.2, adding a new service is similar to replacing a dummy service (which contains no functionality) with this new version. Since a dummy service is inherently consistent at any moment in time, there is no need to finish this dummy service before activating the newly added service. Hence, in this case NeCoMan omits the execution of IP_{old}, ISS_{old}, and RP_{new}.

Besides, the addition of a new service component does not require the execution of UO_{old} and DC_{old} either. Adding a filter component F to a congested node, for instance, involves first loading this component and binding its outports[8]. Next, component F becomes activated by redirecting the packet-flow from component IP to the inport of component F. The execution of $LI_{old-new}$ thus involves removing the connection between the lower layer and IP, and simultaneously connecting this lower layer to F. After that, there is no need anymore to execute UO_{old} and DC_{old} since no old component must be disconnected and removed. This reconfiguration thus completes after executing $LI_{old-new}$.

To illustrate the impact of this customizations, Table D.8 lists all reconfiguration conditions that become changed when a reconfiguration involves service addition.

b) Service removal. Similarly, removing an old service is comparable to replacing this old service with a dummy one. Because the latter (by definition) does

[7] as explained in Section 3.1

[8] Recall that Figure 3.20 and 3.19 depict the composition of the affected node before and after adding this filter component, respectively.

not employ active objects, AP_{new} can safely be omitted. Besides, since service removal does not involve new components, the execution of CC_{new} and LO_{new} will be discarded as well. To illustrate the impact of this customization, Table D.9 lists all reconfiguration conditions that become affected when a reconfiguration involves service removal.

4.8 Conclusion

Finally, we conclude this chapter by revisiting the four requirements that NeCoMan must fulfill to achieve its objectives.

4.8.1 Correct reconfigurations

None of the presented customizations compromise the correctness of the reconfiguration process as long as all associated pre-conditions are fulfilled. Besides, the resulting algorithms meet all (adapted) reconfiguration conditions. So, NeCoMan can safely apply these customizations without compromising the correct functioning of the network.

4.8.2 Limited reconfiguration overhead

All customizations seek to optimize the reconfiguration scenario. For the first two customizations we evaluate their effect on reconfiguration overhead in Appendix E. The other customizations involve omitting all redundant reconfiguration actions. These customizations thus optimize the reconfiguration scenario as well.

Note, however, that applying customization "no finishing" does not always undeniably reduce the reconfiguration overhead. To safely apply this customization, the network must be able to deal with potential inconsistencies. These inconsistencies, however, may impact the network performance. When various packets get lost during reconfiguration, for instance, TCP reduces its congestion window (which affects the network performance). It is clear that such (context specific) performance penalties must also be taken into account when evaluating the benefit associated with omitting all finishing actions.

4.8.3 Limited openness

To carry out a tailored reconfiguration, NeCoMan must be able to identify which customization it can apply to its reconfiguration algorithm[9]. This involves checking for each customization if all associated pre-conditions are fulfilled. To accomplish this, NeCoMan requires from the network administrator to specify (1) the characteristics of the affected services by answering the questions listed in Table 4.5,

[9]Note that we discuss this customization procedure in more detail in Chapter 7.

Question	Cust.
Does the affected service components encapsulate an isolated network service, or do they belong to a distributed network service?	N/A
Is the old service component stateless or stateful?	4.2, 4.3
Is the new service component able to process ongoing protocol-transaction?	4.2, 4.3
Does the new service component restore from or tolerates inconsistent execution states?	4.3
Do the new component's (client and/or server) processes employ active objects?	4.4
Do the affected components encapsulate only client or server processes instead of both?	4.5
Do the affected components employ a unidirectional or a bidirectional communication protocol?	4.6

Table 4.5: Local reconfigurations: questions that the network administrator must answer to specify the service characteristics. The right column lists the related customizations.

Question	Cust.
Does the network tolerate packet re-ordering?	4.2
Do the affected components operate in a best-effort network?	4.3
Does the network restore from or tolerate inconsistent execution states?	4.3
Does the reconfiguration involve service addition, replacement or removal?	4.7

Table 4.6: Local reconfigurations: questions that the network administrator must answer to specify the reconfiguration semantics.

as well as (2) the reconfiguration semantics (which specify the adaptation properties defined by the service developer or the network environment) by answering the questions listed in Table 4.6. This way, NeCoMan restricts the contribution needed from the network administrator to conduct a (tailored) reconfiguration.

4.8.4 Reusability

The customizations that NeCoMan incorporates are based on re-ordering and omitting the reconfiguration actions of both local algorithms. Hence, even after tailoring a reconfiguration algorithm only the predefined set of operations that a node's reconfiguration support must provide will be invoked. These customizations thus will not compromise NeCoMan's ability to be reused on top of other flow-oriented, component-based protocol stack architectures besides DiPS+.

We conclude that the local reconfiguration algorithms extended with the customizations presented in this chapter enable NeCoMan to achieve its objectives.

Chapter 5

Distributed reconfigurations

In the previous two chapters we discussed how NeCoMan conducts local protocol stack reconfigurations. When the unit of adaptation is a distributed network service, however, a reconfiguration obviously involves recomposing multiple stacks. Changing the composition of these stacks independently from each other may possibly break the correct functioning of the network. This chapter and the next one, therefore, focus on how NeCoMan coordinates distributed protocol stack recompositions.

The structure of this chapter bears a close resemblance to Chapter 3. Sections 5.1 and 5.2 first specify how to preserve structural integrity and mutually consistent execution states when adding, replacing, or removing distributed network services. Next, Section 5.3 extends the employed pseudo-formal notation such that reconfiguration conditions which cover multiple nodes can be expressed as well.

The next chapters focus on NeCoMan's two distributed reconfiguration algorithms. The first one of these algorithms is used when all nodes employ monitoring support to drive the affected components to a reconfiguration-safe state. Section 5.4 presents this algorithm and indicates that it fulfills all requires reconfiguration conditions. Next, Section 5.5 explains NeCoMan's second algorithm for conducting distributed reconfigurations. In contrast to the first one, NeCoMan uses this algorithm when all nodes deactivate the affected components immediately and use state transfer support to recover from an inconsistent execution state – that is, instead of monitoring the affected components until a safe state is reached.

Finally, Section 5.6 concludes this chapter by checking both distributed reconfiguration algorithms against the four requirements that NeCoMan must fulfill to achieve its objectives.

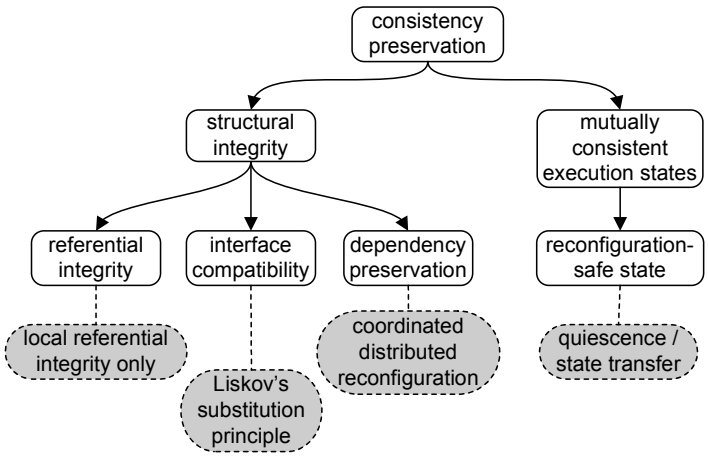


Figure 5.1: Consistency preservation when executing distributed recompositions in (uniform) pipe-and-filter based network architectures

5.1 Structural integrity

Similar to local reconfigurations, preserving the structural integrity of a programmable network during and after completing a distributed reconfiguration involves maintaining *referential integrity*, *interface compatibility*, and *distributed dependencies*.

Referential integrity

How to preserve referential integrity differs in nothing from local reconfigurations. That is, a reconfiguration middleware must only control *local referential integrity* such that after completing a distributed reconfiguration no component bindings are broken and only the new service components will be invoked. This is because the components of a distributed network service do not reference each other directly – that is, by means of remote references.

Interface compatibility

To accomplish a distributed reconfiguration, at every node the interfaces of all components that become (re)connected must be compatible. So, again this is identical to local reconfigurations.

Dependency preservation

Finally, all distributed dependencies between cooperating components must be preserved to prevent a distributed reconfiguration from jeopardizing a programmable

network's correct functioning. That is, at any moment during the reconfiguration a correct distributed service composition must be available. To illustrate this, suppose an MPEG software encoding service is dynamically deployed on two neighboring nodes. When the MPEG software encoder is installed and activated before bringing the decoder into use, packets will arrive at their destination in an encoded form. Breaking the distributed dependencies between collaborating service components during a reconfiguration thus compromises the correct functioning of the network.

To prevent breaking these distributed dependencies, the recomposition of all affected nodes must be *coordinated*. In case of the MPEG service, this includes among others enforcing that the encoding component will only be activated when the neighboring node is prepared to decode all encoded packets when they arrive.

5.2 Mutually consistent execution states

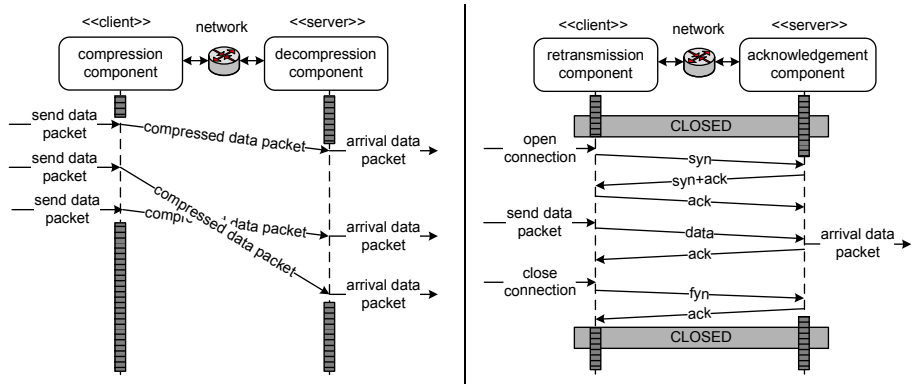
Besides maintaining structural integrity, mutually consistent execution states must be preserved as well to prevent a distributed reconfiguration from compromising the network's correct functioning. Similar to local reconfigurations, this implies that the affected components must be in a reconfiguration-safe state before a recomposition can be executed. How to reach such a safe state, however, slightly differs from the approaches presented in Chapter 3. The remainder of this subsection, therefore, describes the two approaches that we adopted to drive the components of a distributed network service to a reconfiguration-safe state.

5.2.1 Quiescence

The first approach involves completing all ongoing protocol-transactions. This approach is based on Kramer and Magee's definition of "quiescence". In [80], Kramer and Magee define that a node in a distributed system is quiescent when

1. it is not currently engaged in a transaction that it initiated,
2. it will not initiate new transactions,
3. it is not currently engaged in servicing a transaction, and
4. no transactions have been or will be initiated by other nodes which require service from this node.

If these conditions are fulfilled, the application state of a node is both consistent and frozen. That is, it does not include results of partially completed transactions, and its execution state will not change as a result of new transactions [80]. Quiescent nodes are thus in a reconfiguration-safe state, and can safely be removed without leaving the system in an inconsistent state.



(a) Compression service (b) TCP-like service

Figure 5.2: These examples illustrate when collaborating network service components reach a quiescent state. The latter is depicted by gray shaded blocks.

Note that quiescence is a stronger pre-condition to reach a reconfiguration-safe state than what Goudarzi and Kramer proposed in [101]. As explained in Subsection 3.2.2, Goudarzi and Kramer define that a safe state comes about if the affected components are not involved in servicing *accepted* transactions[1]. This implies that pending transaction in which the affected components are not yet participating do not have to be completed. Instead, Goudarzi and Kramer suggest to temporarily ignore these transactions and postpone servicing them until after completing the reconfiguration [101]. Hence, they neglect the last requirement for quiescence, which defines that no transactions have been or will be initiated until after completing the reconfiguration. This requirement is essential, however, when a reconfiguration involves *removing* a distributed service. These reconfigurations cannot be achieved when new service components are expected to complete pending transactions. Therefore, we apply Kramer and Magee's definition of quiescence to reach a reconfiguration-safe state.

If we apply this definition to the collaborating components of a distributed network service, then these components reach a quiescent execution state if

1. all their ongoing protocol-transactions have completed, and

2. they will not initiate new protocol-transactions until after the reconfiguration actions have terminated.

So, if we apply this to the compression service depicted in Figure 5.2(a), then this service reaches quiescence once (1) all compressed packets in transit have been decompressed and (2) the compression component will not be invoked anymore to

[1] as we already discussed in Section 3.2.2

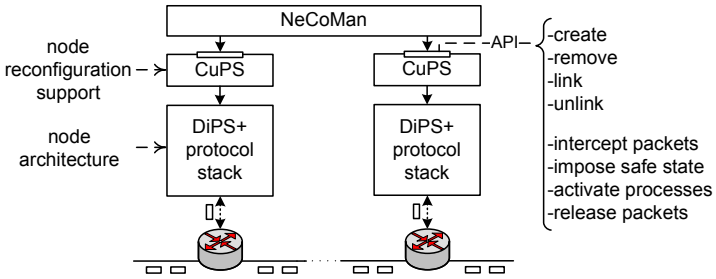


Figure 5.3: CuPS assisting NeCoMan in conducting a distributed reconfiguration of DiPS+ protocol stacks

compress and transmit new data packets. For the TCP-like service depicted in Figure 5.2(b), quiescence is reached after (1) the last ack-message has returned and (2) calls to open new connections are intercepted.

How to drive a distributed network service to a quiescent state depends on the semantics of the collaborating components. Similar to local reconfigurations, NeCoMan therefore does not bring about this safe state by itself. Instead, to reach quiescence, NeCoMan coordinates the distributed execution of both operations to *intercept packets* and to *impose a safe state* that the reconfiguration support of the affected nodes must provide (as illustrated in Figure 5.3)[2]. To clearly understand NeCoMan's first distributed reconfiguration algorithm, we briefly illustrate how CuPS implements these operations when it has to assist NeCoMan in driving DiPS+ components to a quiescent state.

Intercept packets

A first step to drive the collaborating components of a distributed network service to a quiescent state involves intercepting all packets (representing service requests) that introduce the execution of new protocol-transactions. CuPS uses the same mechanism to interfere with the network traffic as it does for all other approaches to reach a safe state – that is, it holds up packets at the outports of (local) neighboring components[3].

Besides, recall that only client processes initiate the execution of new protocol-transactions. When instructed to intercept packets, CuPS therefore only intercepts packets directed towards the *service-external inports* of the affected component's

[2] these operations have been presented in Section 3.3

[3] Note that this is different from how Kramer and Magee reach quiescence. As explained in [80], Kramer and Magee do not intercept packets at component level, but at node level. Hence, they propose to deactivate all nodes that directly or indirectly invoke the affected node so as to freeze the latter (instead of intercepting packets at the nodes that will be reconfigured). We already explained in Section 3.3.2 on page 56 why we did not apply the approach of Kramer and Magee in the context of programmable networks.

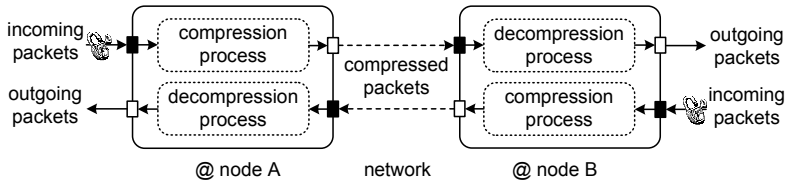


Figure 5.4: Intercepting packets to bring about quiescence over a distributed compression service.

client processes. When removing a compression service, for instance, this involves intercepting packets directed to the service-external inports of the compression processes (as illustrated in Figure 5.4).

Impose safe state

Once packets are prevented from initiating new service activity, NeCoMan can instruct CuPS on every node to impose a safe (quiescent) state. CuPS then monitors the execution state of the affected processes and, if needed, resumes packets one by one until all ongoing protocol-transactions complete. On the node equipped with a compression component, for instance, this involves monitoring until all packets that the compression component has received are compressed (see Figure 5.2(a)). In addition, on the node hosting a decompression component CuPS awaits the decompression of all compressed data packets in transit. Once this is accomplished, the decompression component is in a quiescent state as well and both components can safely be removed.

As an additional example, consider driving a TCP-like service to a quiescent state. As illustrated in Figure 5.2(b), a client application triggers the execution of the associated communication protocol by sending invocations to open a connection, to transmit data, and to close the connection. After intercepting packets that are directed to the retransmission component's service-external inports, however, these packets become held up. On the node that hosts the retransmission component, CuPS therefore resumes these packets one by one until the CLOSED (quiescent) state is reached. As illustrated in Figure 5.2(b), the latter is accomplished after receiving the last ack message. After this occurs, both components of the TCP-like service are quiescent and can safely be removed without leaving the network in an inconsistent state.

Similar to local reconfigurations, DiPS+ component developers must provide "state-monitoring modules" to assist CuPS in determining when a component reaches a quiescent state. In case of the TCP-like service, for instance, this module checks if the retransmission component has reached the CLOSED state. One could argue that this extra user input opposes the requirement for limited user contribution to conduct a reconfiguration. This has been a conscious choice, however, as otherwise

NeCoMan must provide (generic) support to drive a wide variety of distributed services to a quiescent execution state, which in turn compromises its reusability and performance. Besides, for the reconfigurations that have been carried out to validate NeCoMan, the implementation of these state-monitoring modules has restricted to a single method that checks the execution state of the affected component.

5.2.2 State transfer

Monitoring components until all ongoing protocol-transactions complete can be very time consuming. This is especially true when many protocol-transactions are active at the same time (for instance when numerous compressed packets are in transit), or when it takes a long time for the ongoing protocol-transaction to complete (such as for the TCP-like service). In both cases driving the affected service components to a quiescent state significantly delays the reconfiguration. Besides, in some cases it may even be impossible to reach a quiescent state, for instance because the employed protocol is non-deterministic. NeCoMan's second distributed reconfiguration algorithms, therefore, builds upon a different approach to reach a safe state. This approach involves (1) deactivating the affected components immediately at each node, and (2) restoring consistency by transferring the execution state of the old components to the new ones. How CuPS implements its *intercept packets* and *impose a safe state* operations to assist NeCoMan in reaching a safe state in this way has already been discussed in detail in Section 3.3.4.

5.3 Extensions to pseudo-formal notation

Similar to local reconfigurations, we will identify a number of reconfiguration conditions that NeCoMan must fulfill to carry out correct distributed reconfigurations. These reconfiguration conditions specify the order in which NeCoMan must initiate both local and distributed reconfiguration actions. The pseudo-formal notation that is used in Chapter 3, however, only serves to express local reconfiguration conditions. We extend the employed notation, therefore, to express distributed reconfiguration conditions as well.

First, we specify for each reconfiguration action the node on which it becomes executed. $A(node_x)$ and $B(node_y)$, for instance, denote the execution of reconfiguration actions A and B on nodes x and y, respectively. Note that nodes x and y are instances of the set of nodes that participate in a reconfiguration.

Besides, for each reconfiguration condition we also specify the affected nodes. Reconfiguration condition

$$\forall node_x : [A(node_x) \leftarrow \forall node_y : B(node_y)]$$

for instance, expresses that NeCoMan can only initiate the execution of action A on every node x once action B has been completed on every node y (which includes

node x as well). When action B does not have to be completed on node x, the previous reconfiguration condition becomes expressed as

$$\forall node_x : [A(node_x) \leftarrow \forall node_y \neq node_x : B(node_y)]$$

Furthermore, we express local reconfiguration conditions as follows:

$$\forall node_x : [A(node_x) \leftarrow B(node_x)]$$

This reconfiguration condition dictates that on every node x action A can only be initiated once B has been completed on that node.

Finally, we still use the $\equiv$ operator to denote how a high-level action is composed out of fine-grained actions. For distributed reconfigurations, these high-level actions (can) cover multiple nodes. Therefore, expression

$$C \equiv \forall node_x : [A(node_x) \curlywedge B(node_x)]$$

denotes that the distributed execution of high-level action C includes the execution of actions A and B on every node x participating in the reconfiguration.

5.4 Distributed reconfigurations that include reaching quiescence

The remainder of this chapter builds up to both reconfiguration algorithms that NeCoMan employs for conducting distributed reconfigurations[4]. Similar to local reconfigurations, these basic algorithms seek to conduct the distributed reconfiguration of a broad variety of network services. These general-purpose algorithms, therefore, do not take into account the characteristics of the affected services nor the reconfiguration semantics.

This section builds up to the algorithm that NeCoMan uses for conducting distributed reconfigurations that include reaching quiescence. Subsection 5.4.1 first briefly describes the four (distributed) reconfiguration phases that are involved as well as the high-level reconfiguration conditions that NeCoMan must fulfill when executing these phases. Next, Subsection 5.4.2 zooms in on each of these reconfiguration phases to elaborate on the distributed execution of the associated reconfiguration actions. After that, Subsection 5.4.3 completes the set of reconfiguration conditions by refining the high-level conditions defined in Subsection 5.4.1. Finally, Subsection 5.4.4 presents the algorithm itself, and indicates that this algorithm fulfills all required reconfiguration conditions.

[4]Recall that NeCoMan uses the first algorithms when the reconfiguration support on every node drives the affected component to a quiescent execution state – that is, to reach a reconfiguration-safe state. The other algorithm, in contrast, is used when each node deactivates the affected component immediately and restores consistency afterwards by capturing and reinstating the current execution state.

5.4.1 High-level reconfiguration phases and conditions

The four reconfiguration phases that are involved are similar as for local reconfigurations, except that they are targeted at distributed recompositions. To be precise, these phases include (1) the installation of the new distributed network service – by making the new service components available on all nodes involved, (2) the activation of the new distributed network service – by bringing these new service components into use, (3) finishing the old distributed network service – by driving the old service components to a quiescent state, and (4) the removal of the old distributed network service – by deleting the old service components from all participating nodes. These four phases will be denoted as IS_{new}, AS_{new}, FS_{old}, and RS_{old}, respectively.

Furthermore, three high-level reconfiguration conditions need to be fulfilled to carry out correct distributed-service reconfiguration:

1. The new network service can only be activated safely when all its components are made available on all programmable nodes where needed. This is a trivial reconfiguration condition, which can be expressed as

$$AS_{new} \quad \leftarrow \quad IS_{new} \tag{H.1}$$

2. The new network service can only be activated safely when all components that belong to the old network service have reached a quiescent state – that is, when the old service is finished. We formalize this reconfiguration condition as

$$AS_{new} \quad \leftarrow \quad FS_{old} \tag{H.2}$$

3. The removal of the old network service (RS_{old}) can only be initiated safely when all its components have reached a reconfiguration-safe state. We formalize this reconfiguration condition as follows:

$$RS_{old} \quad \leftarrow \quad FS_{old} \tag{H.3}$$

5.4.2 Detailed overview of each reconfiguration phase

We now zoom in on each of these four distributed reconfiguration phases. Note that the implementation of these phases introduces no new reconfiguration actions. In contrast to local reconfigurations, however, some reconfiguration actions become redundant. Besides, the distributed execution of some of the remaining reconfiguration actions must be synchronized. As will be clarified soon, all this results in new reconfiguration conditions that NeCoMan must fulfill to conduct correct distributed reconfigurations.

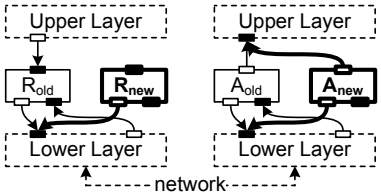


Figure 5.5: Installation of new reliability components

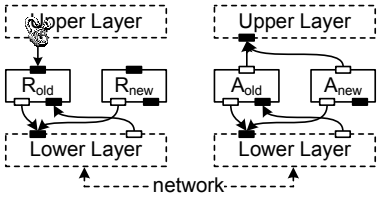


Figure 5.6: Finishing old reliability components

Installation phase

The installation of a new distributed service involves loading the new components on all affected nodes and binding their outports. To illustrate this installation phase, Figure 5.5 sketches the installation of R_{new} and A_{new} to replace R_{old} and A_{old}. The black bold components and bindings in this figure symbolize components and bindings that are created after completing the installation phase.

NeCoMan implements this distributed installation phase by executing reconfiguration actions CC_{new}, LO^{ext}_{new} and LO^{int}_{new} on every node where needed. Hence, we express the implementation of this distributed installation phase as

$$IS_{new} \quad \equiv \quad \forall node_x : [CC_{new}(node_x) \curlywedge LO^{ext}_{new}(node_x) \curlywedge LO^{int}_{new}(node_x)] \text{ (P.1)}$$

Recall that NeCoMan must fulfill reconfiguration condition (3.1) to correctly install new service components[5]. In case of distributed reconfigurations, this condition must be fulfilled at every affected node. We therefore redefine conditions (3.1) as

$$\forall node_x : [LO^{ext}_{new}(node_x) \wedge LO^{int}_{new}(node_x) \leftarrow CC_{new}(node_x)] \tag{5.1}$$

Furthermore, because the newly installed components are not enabled yet to receive packets, there is no need to synchronize their distributed installation. Hence, no distributed reconfiguration conditions apply to the execution of the reconfiguration actions that are involved.

Finishing phase

NeCoMan drives the old service components to a quiescent execution state by instructing the affected nodes to

[5]This reconfiguration condition imposes to only bind a component's outports when the associated component has been created.

- intercept packets that are directed towards the old components' service-external inports, so as to prevent these components' client processes from accepting and processing new service requests, and to
- impose a safe state over these components' collaborating client and server processes by monitoring them and resuming packets until quiescence comes about.

To illustrate (part of) this finishing phase, Figure 5.6 depicts how new packets directed to R_{old} are intercepted so as to drive the old reliability service to a quiescent state. Furthermore, NeCoMan implements this finishing phase by executing reconfiguration actions IP_{old}^{client}, ISS_{old}^{client}, and ISS_{old}^{server} on every node where needed[6]. Hence, we express the implementation of this distributed finishing phase as

$$FS_{old} \equiv \forall node_x : [IP_{old}^{client}(node_x) \curlywedge ISS_{old}^{client}(node_x) \curlywedge ISS_{old}^{server}(node_x)] \quad \text{(P.2)}$$

To correctly finish a distributed service, NeCoMan must coordinate the order in which these actions are executed on every node. To be precise, NeCoMan can only instruct a node x to impose a safe state over its *client processes* once packets directed towards the service-external inports of these processes are intercepted. This reconfiguration condition can be denoted as[7]

$$\forall node_x : [ISS_{old}^{client}(node_x) \leftarrow IP_{old}^{client}(node_x)] \tag{5.2}$$

Besides, NeCoMan must also coordinate the order in which the old client and server processes are driven towards a quiescent execution state. Because a client process initiates the activity of a server process, NeCoMan should only instruct a node x to impose a safe state over its server processes once all invoking client processes (which can be located on every node $y \neq x$) are quiescent. We express this reconfiguration condition as

$$\forall node_x : [ISS_{old}^{server}(node_x) \leftarrow \forall node_y \neq node_x : ISS_{old}^{client}(node_y)] \tag{5.3}$$

Activating every new service component

To activate a new distributed service, NeCoMan instructs all affected nodes to bring the new components into use[8]. To illustrate (part of) this activation phase,

[6] Note that, in contrast to local reconfigurations, IP_{old}^{server} is not included anymore.

[7] This reconfiguration condition is identical to condition (3.2)

[8] As explained in Section 3.5.2, the activation of new service components involves

- rebinding all affected inports, such that packets become delivered to the new service com-

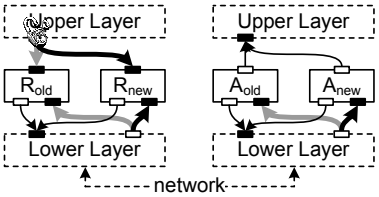


Figure 5.7: Binding inports of new reliability components

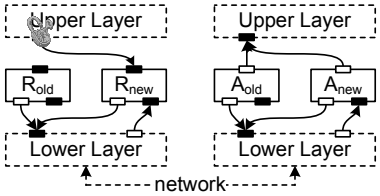


Figure 5.8: Releasing intercepted packets

Figure 5.7 depicts redirecting (intercepted) packets towards the inports of components R_{new} and A_{new}. This involves unlinking the affected inports of R_{old} and A_{old} (symbolized by grey bold connections), and simultaneously linking those of R_{new} and A_{new} (depicted by black bold connections). In addition, Figure 5.8 illustrates resuming the intercepted packets.

NeCoMan implements this distributed activation phase by executing $LI^{int}_{old-new}$, $LI^{ext}_{old-new}$, AP^{client}_{new}, AP^{server}_{new} and RP^{client}_{new} on every node where needed. Hence, we express the implementation of this activation phase as

$$\begin{aligned} AS_{new} \equiv \forall node_x :[&LI^{int}_{old-new}(node_x) \curlywedge LI^{ext}_{old-new}(node_x) \curlywedge \\ &AP^{client}_{new}(node_x) \curlywedge AP^{server}_{new}(node_x) \curlywedge RP^{client}_{new}(node_x)] \end{aligned} \tag{P.3}$$

Note that, in contrast to the local activation of a service component, distributed service activation in this case does not involve the execution of RP^{server}_{new}. This is because packets directed to server processes are not intercepted when bringing about quiescence.

To correctly activate a new distributed service, NeCoMan must coordinate the order in which these actions are executed on every node. In general, a component C can only be activated safely when all other components on which C depends are already operational and prepared to accept and service C's invocations. For distributed network services, these dependencies are formalized by the employed communication protocol. Safe activation of a distributed network service, therefore, requires to first enable the execution of this new communication protocol by making it possible for the new (reactive) service components to process each others invocations. This involves rebinding all inports and initiating the active objects of the new collaborating components[9]. Once this is accomplished, intercepted packets can safely be released to initiate the execution of the new protocol.

ponents after resuming the intercepted packet flows,

- activating all new collaborating client and server processes, and
- resuming packets that are intercepted to finish the old service.

[9]Recall that outports are already bound during installation.

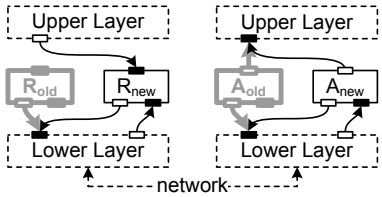


Figure 5.9: Removal of old reliability components

Hence, NeCoMan must synchronize the distributed execution of the reconfiguration actions that implement this activation phase. To be precise, NeCoMan can only instruct node x to release intercepted packets if

1. the service-external and service-internal inports of the new client processes on node x have been bound correctly, and

2. the active objects that these client processes employ have been started, and

3. the service-internal inports of the new server processes (located on every node $y \neq x$) that will service requests from the new client processes located on node x have been bound correctly, and

4. the active objects that these server processes employ have also been started.

We denote this reconfiguration condition as follows:

$$\begin{aligned} \forall node_x : [& RP^{client}_{new}(node_x) \leftarrow \\ & LI^{int}_{old-new}(node_x) \wedge LI^{ext}_{old-new}(node_x) \wedge AP^{client}_{new}(node_x) \wedge \\ & \forall node_y \neq node_x : [LI^{int}_{old-new}(node_y) \wedge AP^{server}_{new}(node_y)]] \end{aligned} \tag{5.4}$$

Removal phase

Removing the old distributed network services involves unlinking and deleting every old service component from the affected nodes. To illustrate this, Figure 5.9 sketches the removal of the old reliability service. NeCoMan implements this distributed removal phase by executing DC_{old}, UO^{ext}_{old}, and UO^{ext}_{old} on every node where needed. Hence, we express the implementation of this removal phase as

$$RS_{old} \quad \equiv \quad \forall node_x : [DC_{old}(node_x) \curlywedge UO^{ext}_{old}(node_x) \curlywedge UO^{int}_{old}(node_x)] \quad \text{(P.4)}$$

Recall that NeCoMan must fulfill reconfiguration condition (3.6) to correctly remove old service components[10]. In case of distributed reconfigurations, this condition must be fulfilled at every affected node. We therefore redefine condition (3.6) as

$$\forall node_x : [DC_{old}(node_x) \leftarrow UO^{ext}_{old}(node_x) \wedge UO^{int}_{old}(node_x) \wedge LI^{ext}_{old-new}(node_x) \wedge LI^{ext}_{old-new}(node_x)] \tag{5.5}$$

Besides, because the old reliability components must be in a reconfiguration-safe state when being removed, their removal cannot compromise the correct operation of the network. So, there is no need to synchronize the distributed execution of the reconfiguration actions that are involved.

5.4.3 Refining high-level reconfiguration conditions

Figure 5.10 depicts a preliminary partial ordering of the employed reconfiguration actions that NeCoMan must fulfill to conduct correct distributed reconfigurations. This ordering results from the reconfiguration conditions specified in the previous two subsections. To complete this partial ordering, we refine the high-level reconfiguration conditions presented in Subsection 5.4.1. This will be accomplished in a similar way as for NeCoMan's local reconfigurations. First, for each high-level reconfiguration condition, we substitute IS_{new}, FS_{old}, AS_{new} and RS_{old} for expressions (P.1), (P.2), (P.3), and (P.4), respectively. Next, we investigate if the resulting reconfiguration condition can be made less stringent without compromising the correctness of the reconfiguration scenario.

Installing new service components before activation (H.1)

Let us start with reconfiguration condition (H.1), which imposes to only activate the new network service when all its components are installed properly on the nodes where needed. To refine this condition, we first replace IS_{new} and AS_{new} by expressions (P.1) and (P.3), respectively. This results in the following expression

$$\begin{aligned} \forall node_x : [&LI^{int}_{old-new}(node_x) \wedge LI^{ext}_{old-new}(node_x) \wedge \\ &AP^{client}_{new}(node_x) \wedge AP^{server}_{new}(node_x) \wedge RP^{client}_{new}(node_x) \\ &\leftarrow \forall node_y : [CC_{new}(node_y) \wedge LO^{ext}_{new}(node_y) \wedge LO^{int}_{new}(node_y)]] \end{aligned} \tag{I.1}$$

This condition can be made less stringent without compromising the reconfiguration correctness. To illustrate this, we split up condition (I.1) in (I.2a), (I.2b),

[10] This (local) condition imposes to only delete a component after disconnecting all its communication ports.

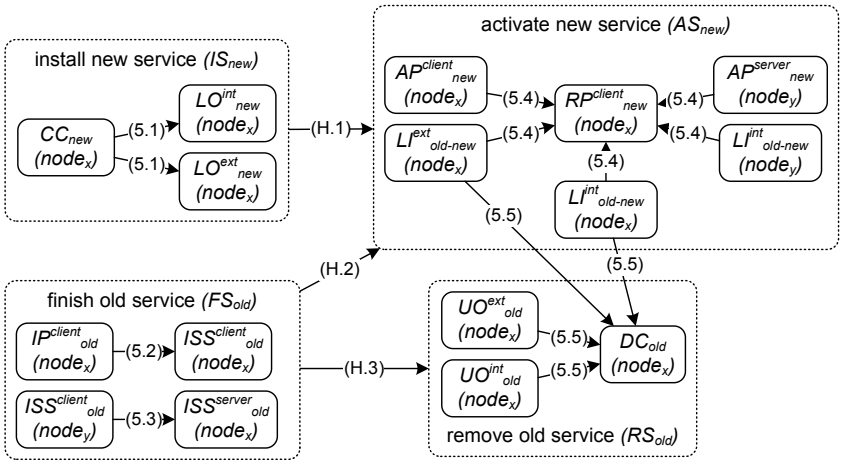


Figure 5.10: Preliminary partial ordering of NeCoMan's reconfiguration actions for carrying out distributed reconfigurations that include reaching quiescence. According to the associated reconfiguration conditions, each of these reconfiguration actions can only be initiated safely if all its referring actions are completed.

and (I.2c), and reduce each of these conditions.

$$\forall node_x : [LI^{int}_{old-new}(node_x) \wedge LI^{ext}_{old-new}(node_x) \leftarrow \forall node_y : [CC_{new}(node_y) \wedge LO^{ext}_{new}(node_y) \wedge LO^{int}_{new}(node_y)]] \quad \text{(I.2a)}$$

$$\forall node_x : [AP^{client}_{new}(node_x) \wedge AP^{server}_{new}(node_x) \leftarrow \forall node_y : [CC_{new}(node_y) \wedge LO^{ext}_{new}(node_y) \wedge LO^{int}_{new}(node_y)]] \quad \text{(I.2b)}$$

$$\forall node_x : [RP^{client}_{new}(node_x) \leftarrow \forall node_y : [CC_{new}(node_y) \wedge LO^{ext}_{new}(node_y) \wedge LO^{int}_{new}(node_y)]] \quad \text{(I.2c)}$$

Condition (I.2a). Unlike what condition (I.2a) defines, there is no need to delay the execution of $LI^{int}_{old-new}$ and $LI^{ext}_{old-new}$ on node x until CC_{new}, LO^{ext}_{new}, and LO^{int}_{new} have been completed on every affected node y. Instead, a new component's service-external and service-internal inports can safely be bound on node x once this component has been made available on that node. Hence, we can reduce condition (I.2a) to

$$\forall node_x : [LI^{int}_{old-new}(node_x) \wedge LI^{ext}_{old-new}(node_x) \leftarrow CC_{new}(node_x)] \quad (5.6)$$

Condition (I.2b). Condition (I.2b), in turn, can be reduced in a similar way. Starting the active objects of the new client and server processes on node x can be accomplished without waiting for CC_{new}, LO^{ext}_{new}, and LO^{int}_{new} to complete on every node y that is involved in this reconfiguration. Instead, these active objects can safely be started on node x once the component where they belong to is available on that node. This is because the reconfiguration scenario that we discuss here does not support imposing a safe state by transferring the affected components' execution state. Hence, starting a component's active objects will not cause service activity – that is, to process all (ongoing) requests that were interrupted when deactivating the old processes. So, we reduce condition (I.2b) to

$$\forall node_x : [AP^{client}_{new}(node_x) \wedge AP^{server}_{new}(node_x) \leftarrow CC_{new}(node_x)] \tag{5.7}$$

Condition (I.2c). Finally, we examine condition (I.2c), which dictates that RP^{client}_{new} can only be executed on node x when CC_{new}, LO^{ext}_{new}, and LO^{int}_{new} have been completed on every node y (thus including node x as well). Again, this pre-condition can safely be made less stringent.

First, there is no need to wait until all service-external outports on node x are connected before initiating RP^{client}_{new}. These outports will only be used by the new server processes on node x, which do not collaborate with the new client processes that are brought in use by executing $RP^{client}_{new}(node_x)$. Hence, condition (I.2c) can safely be reduced to

$$\begin{aligned}\forall node_x :&[RP^{client}_{new}(node_x) \leftarrow CC_{new}(node_x) \wedge LO^{int}_{new}(node_x) \wedge \\ &\forall node_y \neq node_x : [CC_{new}(node_y) \wedge LO^{int}_{new}(node_y) \wedge LO^{ext}_{new}(node_y)]]\end{aligned} \tag{I.3}$$

Second, the right operand of this expression can again be made less stringent. When condition (5.1) is fulfilled, then $LO^{int}_{new}(node_x)$ and $LO^{ext}_{new}(node_x)$ will only be initiated once $CC_{new}(node_x)$ is completed. Hence, we can safely remove $CC_{new}(node_x)$ and $CC_{new}(node_y)$ from the right operand of condition (I.3), which results in

$$\begin{aligned}\forall node_x :&[RP^{client}_{new}(node_x) \leftarrow LO^{int}_{new}(node_x) \wedge \\ &\forall node_y \neq node_x : [LO^{int}_{new}(node_y) \wedge LO^{ext}_{new}(node_y)]]\end{aligned} \tag{5.8}$$

Conclusion. So, we conclude that to satisfy condition (H.1), NeCoMan must meet reconfiguration conditions (5.1), (5.6), (5.7), and (5.8).

Finishing old service components before activating new ones (H.2)

Next, we refine reconfiguration condition (H.2). This condition dictates to only activate the new network service when all components that belong to the old service have reached a reconfiguration-safe state. Replacing FS_{old} and AS_{new} in (H.2) by expressions (P.2) and (P.3) results in

$$\begin{aligned}\forall node_x :[&LI^{int}_{old-new}(node_x) \wedge LI^{ext}_{old-new}(node_x) \wedge \\ &AP^{client}_{new}(node_x) \wedge AP^{server}_{new}(node_x) \wedge RP^{client}_{new}(node_x) \\ &\leftarrow \forall node_y : [IP^{client}_{old}(node_y) \wedge ISS^{client}_{old}(node_y) \wedge ISS^{server}_{old}(node_y)]]\end{aligned} \tag{I.4}$$

We can weaken this condition without compromising the correct functioning of the network and its services. We do this in two steps. First, we simplify condition (I.4) by relying on reconfiguration condition (5.2). This reconfiguration condition imposes to only initiate ISS^{client}_{old} on node x once IP^{client}_{old} has completed on that node. So, given that this condition will be fulfilled, we can safely remove IP^{client}_{old} from the right operand of condition (I.4). This results in

$$\begin{aligned}\forall node_x :[&LI^{int}_{old-new}(node_x) \wedge LI^{ext}_{old-new}(node_x) \wedge \\ &AP^{client}_{new}(node_x) \wedge AP^{server}_{new}(node_x) \wedge RP^{client}_{new}(node_x) \\ &\leftarrow \forall node_y : [ISS^{client}_{old}(node_y) \wedge ISS^{server}_{old}(node_y)]]\end{aligned} \tag{I.5}$$

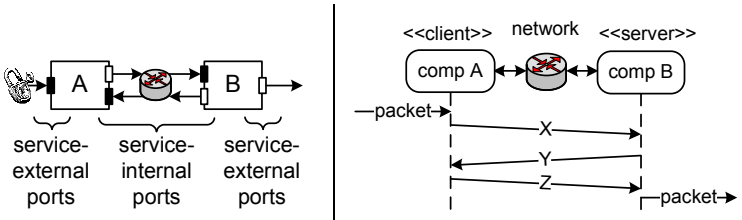
Next, we split up condition (I.5) into (I.6a), (I.6b), and (I.6c), and reduce each of these conditions.

$$\begin{aligned}\forall node_x :[&LI^{int}_{old-new}(node_x) \wedge LI^{ext}_{old-new}(node_x) \\ &\leftarrow \forall node_y : [ISS^{client}_{old}(node_y) \wedge ISS^{server}_{old}(node_y)]]\end{aligned} \tag{I.6a}$$

$$\begin{aligned}\forall node_x :[&AP^{client}_{new}(node_x) \wedge AP^{server}_{new}(node_x) \\ &\leftarrow \forall node_y : [ISS^{client}_{old}(node_y) \wedge ISS^{server}_{old}(node_y)]]\end{aligned} \tag{I.6b}$$

$$\begin{aligned}\forall node_x :[&RP^{client}_{new}(node_x) \\ &\leftarrow \forall node_y : [ISS^{client}_{old}(node_y) \wedge ISS^{server}_{old}(node_y)]]\end{aligned} \tag{I.6c}$$

Condition (I.6a). Condition (I.6a) expresses that NeCoMan can only execute $LI^{int}_{old-new}$ and $LI^{ext}_{old-new}$ on node x once all client and server processes on every node y are finished. This pre-condition can safely be made less stringent. To be precise, NeCoMan does not have to wait until all server processes located on every



(a) Two collaborating components (b) Employed communication protocol

Figure 5.11: The client process is quiescent after sending message **Z**. The server process, in turn, reaches quiescence after receiving and processing this message **Z**.

node $y \neq x$ have reached quiescence before executing $LI^{int}_{old-new}$ and $LI^{ext}_{old-new}$ on node x.

To illustrate this, consider the setup in Figure 5.11(a). Both components communicate by the protocol depicted in Figure 5.11(b). Once the client process is quiescent (that is, after sending message **Z**), it will not be invoked anymore by the collaborating server process[11]. NeCoMan therefore can safely execute $LI^{ext}_{old-new}$ and $LI^{int}_{old-new}$ on node x without waiting until the server process on node y reaches quiescence as well (that is, until is has received and processed message **Z**). So, we can weaken condition (I.6a) to

$$\begin{aligned} \forall node_x : [&LI^{int}_{old-new}(node_x) \wedge LI^{ext}_{old-new}(node_x) \\ &\leftarrow ISS^{client}_{old}(node_x) \wedge ISS^{server}_{old}(node_x) \wedge \\ &\forall node_y \neq node_x : ISS^{client}_{old}(node_y)] \end{aligned} \tag{I.7}$$

Furthermore, we can make the right operand of condition (I.7) less stringent by relying on condition (5.3). The latter imposes to only execute ISS^{server}_{old} on node x once ISS^{client}_{old} has been completed on every node $y \neq x$. Hence, condition (I.7) can be reduced to

$$\begin{aligned} \forall node_x :[&LI^{int}_{old-new}(node_x) \wedge LI^{ext}_{old-new}(node_x) \\ &\leftarrow ISS^{client}_{old}(node_x) \wedge ISS^{server}_{old}(node_x)] \end{aligned} \tag{I.8}$$

Finally, because only client processes expose service-external inports, condition (I.8) can safely be weakened to conditions (5.9) and (5.10).

$$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow ISS^{client}_{old}(node_x) \wedge ISS^{server}_{old}(node_x)] \tag{5.9}$$

$$\forall node_x : [LI^{ext}_{old-new}(node_x) \leftarrow ISS^{client}_{old}(node_x)] \tag{5.10}$$

[11] If the server process still invokes the client process, then this client process did not complete its participation in all ongoing protocol-transactions, and therefore was not yet quiescent.

Condition (I.6b). Next, we reduce condition (I.6b), which dictates to only start the active objects of the new component's client and server processes once every old process on each node y has reached quiescence. This pre-condition can safely be weakened. Similar to the execution of $LI^{int}_{old-new}$ and $LI^{ext}_{old-new}$, starting the active objects of the new client and server processes on node x can be accomplished once ISS^{client}_{old} and ISS^{server}_{old} are completed on node x. Therefore, we can safely reduce condition (I.6b) to

$$\begin{aligned}\forall node_x :[&AP^{client}_{new}(node_x) \wedge AP^{server}_{new}(node_x) \\ &\leftarrow ISS^{client}_{old}(node_x) \wedge ISS^{server}_{old}(node_x)]\end{aligned} \tag{I.9}$$

In addition, there is no need to wait for both the old client and server processes on node x to reach a safe state before activating the new client and server process on that node. This is because these client and server processes operate independently from each other. So, condition (I.9) can safely be reduced to

$$\forall node_x : [AP^{client}_{new}(node_x) \leftarrow ISS^{client}_{old}(node_x)] \tag{5.11}$$

$$\forall node_x : [AP^{server}_{new}(node_x) \leftarrow ISS^{server}_{old}(node_x)] \tag{5.12}$$

Condition (I.6c). Finally, we examine condition (I.6c). This condition imposes to only release intercepted packets once the old client and server processes on every node y are quiescent. Again this condition can be made less stringent. Once NeCoMan instructs node x to release all intercepted packets, the new client processes located on that node will start collaborating with new server processes that may be located on every node $y \neq x$. NeCoMan therefore must only wait for the old counterparts of these collaborating processes to reach quiescence before initiating RP^{client}_{new} – that is, instead of waiting until all processes on every node are quiescent. Consequently, we can safely weaken condition (I.6c) to

$$\begin{aligned}\forall node_x :[&RP^{client}_{new}(node_x) \leftarrow ISS^{client}_{old}(node_x) \wedge \\ &\forall node_y \neq node_x : ISS^{server}_{old}(node_y)]\end{aligned} \tag{I.10}$$

However, this condition is already fulfilled when conditions (5.4), (5.9), and (5.10) are met[12].

[12] Condition (5.4) imposes to only execute RP^{client}_{new} on node x once $LI^{int}_{old-new}$, $LI^{ext}_{old-new}$ and AP^{client}_{new} are completed on that node x, and $LI^{int}_{old-new}$ and AP^{server}_{new} are completed on every other node $y \neq x$. According to condition (5.10), $LI^{ext}_{old-new}(node_x)$ in turn can only be executed when $ISS^{client}_{old}(node_x)$ has completed. Besides, condition (5.9) dictates that $LI^{int}_{old-new}(node_y)$ can only be executed once $ISS^{client}_{old}(node_y)$ and $ISS^{server}_{old}(node_y)$ are completed. Hence, condition (I.6c) becomes redundant and thus can safely be removed.

Conclusion. To conclude, NeCoMan thus meets condition (H.2) when it fulfills conditions (5.2), (5.3), (5.4), (5.9), (5.10), (5.11), and (5.12).

Finishing old service components before removal (H.3)

Finally, reconfiguration condition (H.3) can be refined in an analogous way. This reconfiguration condition imposes to only initiate the removal of the old network service when all its components are finished. Substituting FS_{old} and RS_{old} in (H.3) by expressions (P.2) and (P.4) results in

$$\begin{aligned}\forall node_x : [&DC_{old}(node_x) \wedge UO^{ext}_{old}(node_x) \wedge UO^{int}_{old}(node_x) \\ &\leftarrow \forall node_y : [IP^{client}_{old}(node_y) \wedge ISS^{client}_{old}(node_y) \wedge ISS^{server}_{old}(node_y)]]\end{aligned} \tag{I.11}$$

We can weaken this condition in four steps. First, similar to the refinement of condition (H.2), the right operand of this condition can be made less stringent. According to reconfiguration condition (5.2), ISS^{client}_{old} can only be initiated on node x once IP^{client}_{old} has completed on that node. Therefore we can safely reduce condition (I.11) to

$$\begin{aligned}\forall node_x : [&DC_{old}(node_x) \wedge UO^{ext}_{old}(node_x) \wedge UO^{int}_{old}(node_x) \\ &\leftarrow \forall node_y : [ISS^{client}_{old}(node_y) \wedge ISS^{server}_{old}(node_y)]]\end{aligned} \tag{I.12}$$

Next, we can again make the right operand of this expression less stringent. Once the client processes on node x are quiescent, they will not be invoked anymore by collaborating server processes. NeCoMan therefore can safely execute UO^{int}_{old}, UO^{ext}_{old} and DC_{old} on node x without waiting until the server processes on node y are quiescent. Hence, we can safely weaken the right operand of expression (I.12), which results in

$$\begin{aligned}\forall node_x :[&DC_{old}(node_x) \wedge UO^{ext}_{old}(node_x) \wedge UO^{int}_{old}(node_x) \\ &\leftarrow ISS^{client}_{old}(node_x) \wedge ISS^{server}_{old}(node_x) \wedge \\ &\forall node_y \neq node_x : ISS^{client}_{old}(node_y)]\end{aligned} \tag{I.13}$$

In addition, because reconfiguration condition (5.3) dictates NeCoMan to only execute ISS^{server}_{old} on node x once ISS^{client}_{old} has been completed on every node $y \neq x$, we can reduce condition (I.13) to

$$\begin{aligned}\forall node_x :[&DC_{old}(node_x) \wedge UO^{ext}_{old}(node_x) \wedge UO^{int}_{old}(node_x) \\ &\leftarrow ISS^{client}_{old}(node_x) \wedge ISS^{server}_{old}(node_x)]\end{aligned} \tag{I.14}$$

Finally, we split up condition (I.14) into (I.15a), (I.15b), and (I.15c), and examine each of these conditions.

$$\forall node_x : [UO^{int}_{old}(node_x) \leftarrow ISS^{client}_{old}(node_x) \wedge ISS^{server}_{old}(node_x)] \quad \text{(I.15a)}$$

$$\forall node_x : [UO^{ext}_{old}(node_x) \leftarrow ISS^{client}_{old}(node_x) \wedge ISS^{server}_{old}(node_x)] \quad \text{(I.15b)}$$

$$\forall node_x : [DC_{old}(node_x) \leftarrow ISS^{client}_{old}(node_x) \wedge ISS^{server}_{old}(node_x)] \quad \text{(I.15c)}$$

Condition (I.15a). Condition (I.15a) cannot be reduced. Both a component's client and server processes may employ service-internal outports. These processes must be quiescent before NeCoMan can unbind the associated outports. Reconfiguration condition (5.13) therefore is identical to (I.15a).

$$\forall node_x : [UO^{int}_{old}(node_x) \leftarrow ISS^{client}_{old}(node_x) \wedge ISS^{server}_{old}(node_x)] \quad (5.13)$$

Condition (I.15b). Because only server processes employ service-external outports, NeCoMan does not have to wait for ISS^{client}_{old} to complete on node x before unbinding the old component's service-external outports on that node. So, condition (I.15b) can safely be reduced to

$$\forall node_x : [UO^{ext}_{old}(node_x) \leftarrow ISS^{server}_{old}(node_x)] \quad (5.14)$$

Condition (I.15c). Finally, condition (I.15c) is already fulfilled when conditions (5.5), (5.13) and (5.14) are met. Condition (I.15c) therefore becomes redundant and can safely be removed.

Conclusion. To conclude, NeCoMan thus meets reconfiguration condition (H.3) when it fulfills conditions (5.2), (5.3), (5.5), (5.13), and (5.14).

Partial ordering of reconfiguration actions

By combining all these reconfiguration conditions (which are summarized in Table 5.1), we can specify the partial ordering of reconfiguration actions that NeCoMan must fulfill when recomposing node x as part of a distributed reconfiguration. This ordering is illustrated in Figure 5.12.

	reconfiguration condition	h-l cond.	place
(5.1)	$\forall node_x : [LO^{ext}_{new}(node_x) \wedge LO^{int}_{new}(node_x) \leftarrow CC_{new}(node_x)]$		p_2/p_{17}
(5.2)	$\forall node_x : [ISS^{client}_{old}(node_x) \leftarrow IP^{client}_{old}(node_x)]$		p_5/p_{20}
(5.3)	$\forall node_x : [ISS^{server}_{old}(node_x) \leftarrow \forall node_y \neq node_x : ISS^{client}_{old}(node_y)]$		p_6/p_{21}
(5.4)	$\forall node_x : [RP^{client}_{new}(node_x) \leftarrow LI^{int}_{old-new}(node_x) \wedge LI^{ext}_{old-new}(node_x) \wedge AP^{client}_{new}(node_x) \wedge \forall node_y \neq node_x : [LI^{int}_{old-new}(node_y) \wedge AP^{server}_{new}(node_y)]]$		p_{11}/p_{26}
(5.5)	$\forall node_x : [DC_{old}(node_x) \leftarrow UO^{ext}_{old}(node_x) \wedge UO^{int}_{old}(node_x) \wedge LI^{ext}_{old-new}(node_x) \wedge LI^{int}_{old-new}(node_x)]$		p_{14}/p_{29}
(5.6)	$\forall node_x : [LI^{int}_{old-new}(node_x) \wedge LI^{ext}_{old-new}(node_x) \leftarrow CC_{new}(node_x)]$	(H.1)	p_2/p_{17}
(5.7)	$\forall node_x : [AP^{client}_{new}(node_x) \wedge AP^{server}_{new}(node_x) \leftarrow CC_{new}(node_x)]$	(H.1)	p_2/p_{17}
(5.8)	$\forall node_x : [RP^{client}_{new}(node_x) \leftarrow LO^{int}_{new}(node_x) \wedge \forall node_y \neq node_x : [LO^{int}_{new}(node_y) \wedge LO^{ext}_{new}(node_y)]]$	(H.1)	p_6/p_{21}
(5.9)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow ISS^{client}_{old}(node_x) \wedge ISS^{server}_{old}(node_x)]$	(H.2)	p_7/p_{22}
(5.10)	$\forall node_x : [LI^{ext}_{old-new}(node_x) \leftarrow ISS^{client}_{old}(node_x)]$	(H.2)	p_6/p_{21}
(5.11)	$\forall node_x : [AP^{client}_{new}(node_x) \leftarrow ISS^{client}_{old}(node_x)]$	(H.2)	p_6/p_{21}
(5.12)	$\forall node_x : [AP^{server}_{new}(node_x) \leftarrow ISS^{server}_{old}(node_x)]$	(H.2)	p_7/p_{22}
(5.13)	$\forall node_x : [UO^{int}_{old}(node_x) \leftarrow ISS^{client}_{old}(node_x) \wedge ISS^{server}_{old}(node_x)]$	(H.3)	p_7/p_{22}
(5.14)	$\forall node_x : [UO^{ext}_{old}(node_x) \leftarrow ISS^{server}_{old}(node_x)]$	(H.3)	p_7/p_{22}

Table 5.1: Overview of all reconfiguration conditions that must be fulfilled to correctly execute distributed reconfigurations that include reaching quiescence.

5.4.4 Reconfiguration algorithm

We now present the algorithm that NeCoMan uses to execute these distributed reconfigurations. To simplify its representation, the model of this algorithm (which is illustrated in Figure 5.13) describes a distributed reconfiguration that involves only

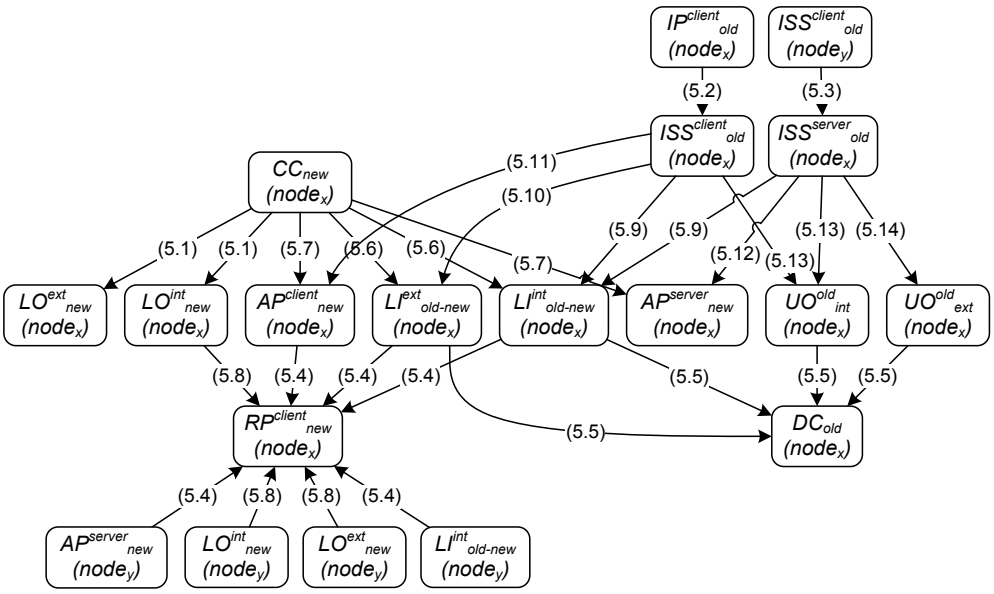


Figure 5.12: Overview of the partial ordering of NeCoMan's reconfiguration actions for carrying out distributed recompositions that include reaching quiescence.

two programmable nodes. Besides, since both nodes are reconfigured concurrently, the initial marking of this Petri net model is defined by two tokens located at places p_1 and p_{16}.

As the model in Figure 5.13 specifies, on every node NeCoMan starts this reconfiguration by executing the reconfiguration actions that implement the installation phase. Next, the reconfiguration actions for driving the old components to a quiescent execution state are carried out. After that, NeCoMan initiates the reconfiguration actions responsible for activating the new service components. Finally, the reconfiguration actions for removing the old components are executed. To clarify all possible execution states, Table 5.2 lists the reconfiguration actions that NeCoMan has executed when a token reaches places p_1 to p_{15}[13]. Besides, Appendix G exemplifies this algorithm with the dynamic replacement of the reliability service.

As explained in the previous sections, NeCoMan must synchronize the distributed execution of several reconfiguration actions. Reconfiguration condition (5.3), for instance, imposes to only initiate the execution of ISS_{old}^{server} on node x once ISS_{old}^{client} has completed on every node $y \neq x$. To accomplish this, NeCoMan exchanges synchronization message **A** between the affected nodes (see Figure 5.13).

[13]Note that these places are associated with node *A*. Places p_{16} to p_{30} represent the corresponding places related to node *B*. The reconfiguration actions that are executed when a token reaches places p_{16} to p_{30}, therefore, are similar as for reaching places p_1 to p_{15}.

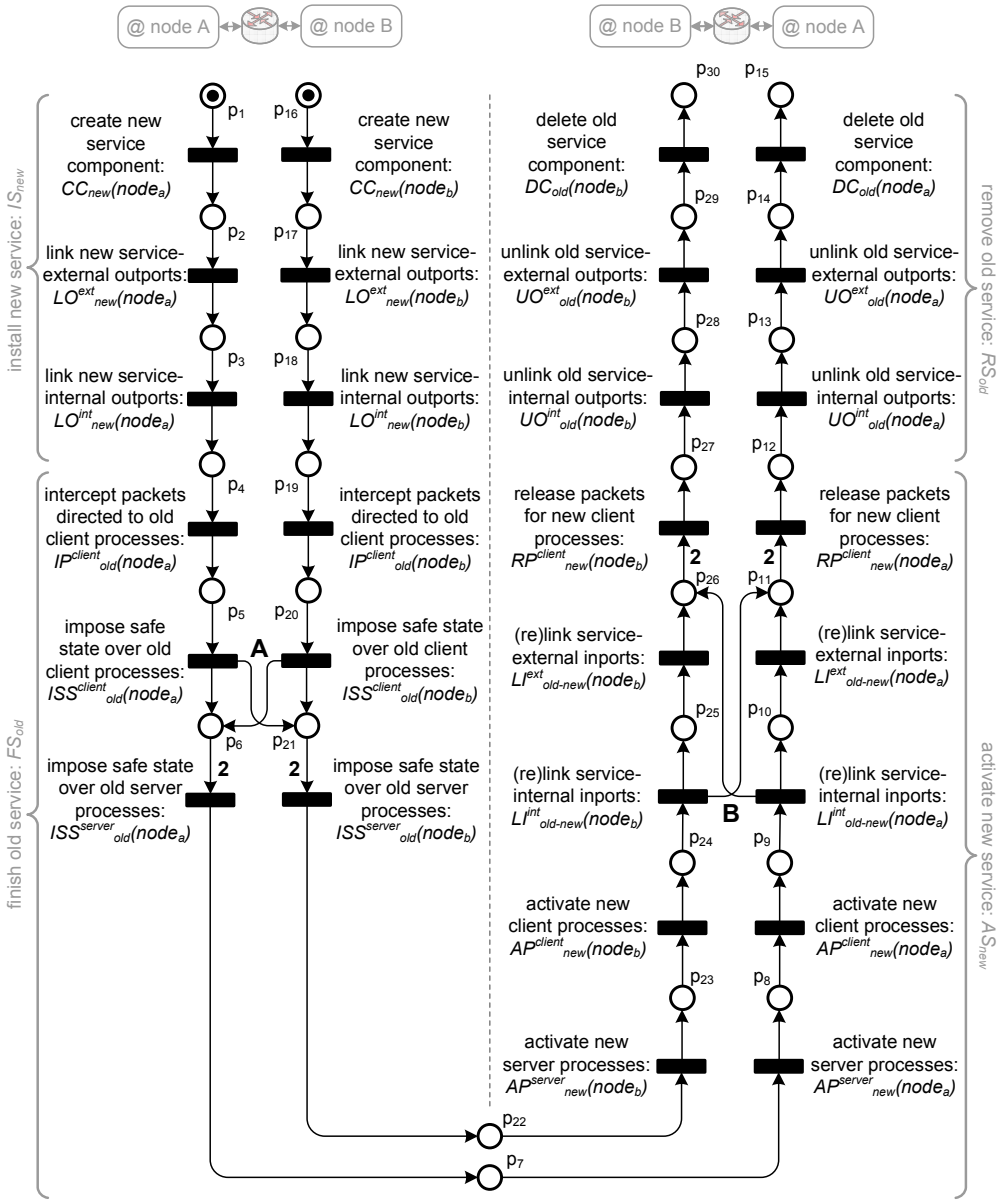


Figure 5.13: Petri net representation of the algorithm that NeCoMan uses for conducting distributed reconfigurations that include reaching quiescence.

place	reconfiguration actions that are completed	place	reconfiguration actions that are completed
p_1	none	p_8	$ac(p_7) \wedge AP^{server}_{new}(node_a)$
p_2	$CC_{new}(node_a)$	p_9	$ac(p_8) \wedge AP^{client}_{new}(node_a)$
p_3	$ac(p_2) \wedge LO^{ext}_{new}(node_a)$	p_{10}	$ac(p_9) \wedge LI^{int}_{old-new}(node_a)$
p_4	$ac(p_3) \wedge LO^{int}_{new}(node_a)$	p_{11}	$ac(p_{10}) \wedge LI^{ext}_{old-new}(node_a) \wedge$
p_5	$ac(p_4) \wedge IP^{client}_{old}(node_a)$		$ISS^{server}_{old}(node_b) \wedge$
p_6	$ac(p_5) \wedge ISS^{client}_{old}(node_a) \wedge$		$AP^{server}_{new}(node_b) \wedge$
	$CC_{new}(node_b) \wedge$		$AP^{client}_{new}(node_b) \wedge$
	$LO^{ext}_{new}(node_b) \wedge$		$LI^{int}_{old-new}(node_b)$
	$LO^{int}_{new}(node_b) \wedge$	p_{12}	$ac(p_{11}) \wedge RP^{client}_{new}(node_a)$
	$IP^{client}_{old}(node_b) \wedge$	p_{13}	$ac(p_{12}) \wedge UO^{int}_{old}(node_a)$
	$ISS^{client}_{old}(node_b)$	p_{14}	$ac(p_{13}) \wedge UO^{ext}_{old}(node_a)$
p_7	$ac(p_6) \wedge ISS^{server}_{old}(node_a)$	p_{15}	$ac(p_{14}) \wedge DC_{old}(node_a)$

Table 5.2: NeCoMan's reconfiguration algorithm to conduct distributed recompositions that include reaching quiescence: definition of places p_1 to p_{15}. Note that $ac(p_x)$ represents all reconfiguration actions that have been completed when the token reaches place p_x.

To be precise, once the execution of ISS^{client}_{old} has completed on node x, NeCoMan broadcasts this synchronization message to all its peer nodes. On these nodes, NeCoMan then awaits the arrival of all expected instances of message **A** before initiating the execution of ISS^{server}_{old}. The Petri net model specifies this by the weight of the outgoing arcs of places p_6 and p_{21}[14].

Furthermore, according to reconfiguration conditions (5.4) and (5.8), NeCoMan can only safely initiate the execution of RP^{client}_{new} on node x when LO^{int}_{new}, LO^{ext}_{new}, $LI^{int}_{old-new}$, and AP^{server}_{new} are completed on every other node $y \neq x$. As one can deduce from Figure 5.13, these four actions are completed after carrying out $LI^{int}_{old-new}$. NeCoMan therefore broadcasts synchronization message **B** to its peer nodes once this reconfiguration action is executed. Furthermore, the weight of the outgoing arcs of places p_{11} and p_{26} indicate that RP^{client}_{new} can only be executed after receiving all expected instances of message **B**.

Finally, to illustrate that the algorithm depicted in Figure 5.13 fulfills all reconfiguration actions, the right column of Table 5.1 identifies for each reconfiguration condition the place as from which the associated pre-condition is fulfilled. This way, one can verify for each reconfiguration action that the associated conditions are met. Furthermore, Appendix I demonstrates in more detail that this is the case.

[14]Note that this weight is 2 for the algorithm modelled in Figure 5.13. The weight of all other arcs is 1, and therefore has not been depicted.

5.5 Distributed reconfigurations that include transferring execution state

NeCoMan uses a different reconfiguration algorithm when the affected nodes deactivate components immediately and restore consistency afterwards by capturing and reinstating the current execution state – that is, instead of driving these components to a quiescent execution state. As we discuss in the remainder of this section, reaching a safe state in this way influences the reconfiguration scenario. Besides, NeCoMan also restricts the reconfigurations that can be executed in this case.

Service replacement only

To enable restoring consistency through state transfer, a reconfiguration must be restricted to *service replacement* only. Transferring the execution state of both collaborating reliability components, for instance, obviously requires for the old as well as the new service components to be present. Hence, service addition or removal cannot be supported.

No coordinated finishing

Finishing the old components by deactivating their processes immediately and transferring their execution state does not require distributed coordination. Recall that to reach a quiescent execution state, all ongoing protocol-transactions must be completed and new protocol-transactions must be prevented from being initiated. As defined by reconfiguration condition (5.3), this requires to coordinate the distributed execution of ISS_{old}^{client} and ISS_{old}^{server}. When the affected components become deactivated immediately, however, no distributed consistent execution state will be reached. This is because the status of ongoing protocol-transactions is not taken into account. Consequently, deactivating collaborating components (such as R_{old} and A_{old}) and transferring their execution state can be accomplished independently on all affected nodes. This makes reconfiguration condition (5.3) redundant.

No coordinated activation

When deactivating collaborating components (such as R_{old} and A_{old}) independently from each other, there may still be packets in transit that belong to ongoing protocol-transactions at the moment when new components are brought in use. To preserve consistency, therefore, these new components must be able to continue processing all ongoing protocol-transactions. In case of the reliability service, for instance, both R_{new} and A_{new} must be able to process all packets in transit that A_{old} and R_{old} have sent, respectively. Besides, NeCoMan also requires that the old components can process requests that new collaboration components may initiate once they are brought into use. For the reliability service, this implies that R_{old} and A_{old} must be able to process packets sent by A_{new} and R_{new}, respectively.

The activation of such compatible components does not have to be coordinated. For instance, there is no need to wait until A_{new} is brought into use to process R_{new}'s packets before activating the latter if A_{old} is able to service data-packets transmitted by R_{new}, and R_{new} can process ack-packets that return from A_{old}. Reconfiguration conditions (5.4) and (5.8) thus become redundant as well in this case.

Conclusion

To summarize, when programmable nodes impose a safe state by deactivating the affected components immediately and transferring their execution state, then NeCoMan (1) only supports service replacement which involves compatible components, and (2) does not coordinate the distributed execution of reconfiguration actions to finish the old service and to activate the new one. So, in this case NeCoMan recomposes all nodes independently[15]. The algorithm that conducts these reconfigurations, therefore, executes on all affected nodes NeCoMan's local algorithm to replace components of distributed network services[16]. To illustrate this, Figure K.1 depicts the Petri net model of this distributed reconfiguration algorithm.

5.6 Conclusion

To conclude, we check both distributed reconfiguration algorithms against the four requirements that NeCoMan must fulfill to achieve its objectives.

5.6.1 Correct reconfigurations

As demonstrated in Section 5.4.4, the first distributed reconfiguration algorithm meets all conditions that are imposed to conduct correct reconfigurations. For the second algorithm to conduct safe reconfigurations, a reconfiguration must be restricted to the replacement of compatible service components. This has been discussed in the previous section.

5.6.2 Limited reconfiguration overhead

Similar to the local reconfiguration algorithms, both distributed reconfiguration algorithms do not take into account the service characteristics nor the reconfiguration semantics. For a number of services, therefore, the conducted reconfiguration will not be optimized.

[15] That is, without coordinating the distributed execution of the reconfiguration actions that are involved

[16] This algorithm has been presented in Subsection 3.5.4.

5.6.3 Limited openness

Again similar to the local reconfiguration algorithms, both distributed algorithms coordinate the safe reconfiguration of an extensive set of network services. For many services, however, these general-purpose reconfiguration algorithms must be tailored. To illustrate this, Figure G.2 depicts the customized version of NeCoMan's first distributed reconfiguration algorithm for replacing a reliability service which involves reaching quiescence. To restrict the contribution needed from a network administrator to tailor a reconfiguration, NeCoMan must apply such customizations by itself, starting from a declarative description of these service characteristics and reconfiguration semantics.

5.6.4 Reusability

Both distributed algorithms employ no other reconfiguration actions than those used for implementing local reconfigurations. Hence, only the predefined set of operations that a node's reconfiguration support must provide will be invoked. So, NeCoMan is prepared to recompose various flow-oriented, component-based protocol stack architectures.

> *To fully comply with the second and the third requirement, NeCoMan must customize both distributed reconfiguration algorithms (if needed) to fully exploit the service specific characteristics and the reconfiguration semantics. These customizations are discussed in the next chapter.*

Chapter 6

Customizations to distributed reconfigurations

Similar to local reconfigurations, NeCoMan incorporates a set of customizations to its basic distributed reconfiguration algorithms. These customizations seek to accomplish the same goal as the ones presented in Chapter 4. Hence, most of the customizations listed in this chapter have a counterpart which is applicable to NeCoMan's local reconfiguration algorithms. The implementation of many of these customizations as well as their pre-conditions, however, differs when applied to a distributed instead of a local reconfiguration.

The structure of this chapter bears a close resemblance to Chapter 4. First, Section 6.1 briefly lists all customizations and explains how they have been identified. Next, Sections 6.2 to 6.9 describe these customizations in full detail. Finally, Section 6.10 concludes this chapter by elaborating on the four requirements that NeCoMan must fulfill to achieve its objectives.

6.1 Overview

Every customization listed in this chapter originates from the (basic) distributed reconfiguration algorithms presented in the previous chapter. To be precise, we defined these customizations by re-ordering and discarding the reconfiguration actions as well as the synchronization points that both algorithms include. From this set, we selected the customizations that both yield a valid reconfiguration and reduce communication disruption.

A first (resulting) customization involves omitting the synchronization that is required to correctly activate a new distributed network service. This is discussed in Section 6.2. Next, Section 6.3 explains how NeCoMan customizes its distributed reconfiguration algorithms when the order of the activation and the finishing phase can

be switched. Section 6.4 then explains the effect of discarding all finishing actions. After that, Section 6.5 discusses how NeCoMan customizes its (first) distributed reconfiguration algorithm when quiescence can be detected by only monitoring the old client processes (that is, instead of controlling all collaborating server processes as well).

The following customizations are (mainly) based on omitting reconfiguration actions that are redundant for a specific reconfiguration. Section 6.6, for instance, explains how NeCoMan customizes its distributed reconfiguration algorithms when the new service processes do not use active objects. Section 6.7 presents which reconfiguration actions are omitted when the old and/or new service components employ only client or server processes, instead of both. Next, Section 6.8 explains how NeCoMan tailors its distributed reconfiguration algorithms when the old and/or new service components expose only service-internal inports or outports, instead of both. As a last customization, Section 6.9 elaborates on how NeCoMan's distributed reconfiguration algorithms are tailored when a reconfiguration involves service addition or removal instead of replacement.

All these customizations are presented in an analogous way (similar to Chapter 4). We first elaborate on the pre-conditions that must be fulfilled to safely apply a specific customization. Next, we explain the impact of this customization by identifying all changes that it causes. These include the high-level conditions that are affected and the reconfiguration actions that are added and/or removed. After that, we discuss the effect of these changes on the reconfiguration conditions. Furthermore, for customizations "activate before finishing" and "no finishing" (see Sections 6.3 and 6.4), we also illustrate the resulting partial ordering of reconfiguration actions as well as the resulting reconfiguration algorithm. In addition, Appendix K evaluates the impact on reconfiguration overhead of the first three customizations.

Finally, because the algorithm presented in Section 5.4.4 conducts a synchronized distributed reconfiguration, we refer to this algorithm as NeCoMan's *synchronized distributed reconfiguration algorithm*. Likewise, the algorithm presented in Section 5.5 will be called the *independent distributed reconfiguration algorithm*, since this algorithm conducts on all affected nodes the independent execution of NeCoMan's local reconfiguration algorithm for distributed services.

6.2 No coordinated activation

A first customization involves omitting the distributed synchronization that is needed to correctly activate new network service components. As explained in Section 5.4.4, NeCoMan sends message **B** once $LI^{int}_{old-new}$, LO^{int}_{new}, LO^{ext}_{new}, and AP^{server}_{new} are executed on the node that it manages – that is, to give notice that this node is prepared to participate in new protocol-transactions. As we explain in the remainder of this section, for some particular reconfigurations this synchronization can safely be omitted.

Pre-conditions

Synchronization message **B** can only be omitted if

1. old and new services are compatible, or if
2. the network is able to deal with incorrect service compositions.

1) Old and new services are compatible. When this is the case, the old and new client and server processes are able to accept and service each others requests. Consequently, there is no need anymore to coordinate the distributed activation of new service processes. When an old decompression process can service packets transmitted by new compression processes, for instance, NeCoMan can safely bring these new compression processes into use regardless of whether or not the new decompression processes are activated.

2) Network handles incorrect service composition. Besides, message **B** can also be omitted when the network is able to handle incorrect service compositions. If this is the case, activating the new service components independently will not compromise the correct functioning of the network. To illustrate this, consider adding a compression service on two programmable nodes. When NeCoMan uses its synchronized distributed reconfiguration algorithm, the new compression component will only be activated if the new decompression component is prepared to deal with compressed packets. This way, NeCoMan avoids that data packets arrive at their destination in a compressed form. When compressed packets are dealt with once they reach the edges of this network, however, synchronizing the activation of the new compression component can safely be discarded.

Modifications

As a first and obvious effect, this customization involves omitting synchronization message **B**. This implies that only the local activation of new service processes must be coordinated. Hence, reconfiguration conditions (5.4) and (5.8) become replaced with (3.4), (3.5), (3.8), and (3.9).

Besides, if the required pre-conditions are fulfilled, then bringing about quiescence becomes redundant as well. When new decompression processes are able to service all compressed packets in transit (that is, regardless of whether they are compressed by old or new compression processes), for instance, there is no need to wait until all compressed packets in transit are dealt with before activating the new compression and decompression processes. Hence, if a reconfiguration does not require distributed coordination to correctly activate the new service processes, then coordinating the distributed finishing of the old service processes can also be omitted. Consequently, conditions (5.2) and (5.3) are replaced with (3.2) and (3.3), and so message **A** becomes obsolete as well.

To conclude, NeCoMan thus recomposes all affected nodes independently from each others when coordinated activation is not required. Therefore, when applying this customization to its synchronized distributed reconfiguration algorithm, NeCoMan uses its independent reconfiguration algorithm instead to conduct the reconfiguration. Note that this does not imply that the affected nodes have to use state transfer to reach a safe state. To be precise, NeCoMan uses this independent reconfiguration algorithm when

- the distributed finishing phase does not have to be synchronized because the affected nodes transfer execution state to reach a reconfiguration-safe state (as explained in Section 5.5), or when
- the distributed activation phase of the new service components does not have to be synchronized (as explained in this section).

6.3 Activate before finishing

A next customization involves activating the new service before the old one is finished. The benefit of this customization is similar for distributed as for local reconfigurations. That is, communication disruption can be reduced by finishing the old service while the new one is already taken into use. In the remainder of this section, we explain the effect of this customization on NeCoMan's synchronized and independent distributed reconfiguration algorithm.

6.3.1 Synchronized distributed reconfigurations

Pre-conditions

To safely switch the order of activation and finishing actions, a number of preconditions must be fulfilled. These include that

1. the old service components do not share their execution state (if any) with their client applications[1],
2. the affected nodes do not transfer execution state, and
3. the network tolerates packet re-ordering.

1) No service-external state dependencies. When the components of a distributed network services share their execution state with their client applications, the state of the old and new versions of that service must be kept consistent with the state that these client applications expect. To illustrate this, consider the replacement of a TCP-like service (as the one depicted in Figure 5.2(b)) by a revised

[1]The term "client application" covers all service-external functionality that make use of the affected network service.

version after the former has successfully completed a session setup – that is, having reached the ESTAB state. Activating the new TCP-like service before the old version reaches the CLOSED (finished) state would compromises the correct functioning of its client applications. These client applications still expect the service to be in the ESTAB state, while the new version of this service, be default, becomes initialized in the CLOSED state. The old TCP-like service therefore must be finished first before the new one can be activated safely.

When the old service components share their execution state (if any) with each other only, these components are stateless with respect to the rest of the network. Hence, the new version of such a service can be activated before the old one is finished without breaking consistency. Monitoring R_{old} for quiescence, for instance, can safely be carried out while new requests for reliable packet transmission are directed towards R_{new}.

2) No state transfer. In addition, NeCoMan does not apply this customization when the underlying nodes employ state transfer to finish the old service components. This is because if new service processes are operational, overwriting their execution state with older (and most likely outdated) state information compromises the service's correct functioning. Besides, transferring state to a component that is already taken into use is subject to critical section problems. The variables that store this state may be in use (by the component's processes) while being reinstated with new information.

3) Packet re-ordering tolerated. Finally, when the old service becomes finished while the new one is already brought into use, both services temporarily execute in parallel during reconfiguration. Consequently, packets processed by both versions most likely will be re-ordered. This customization, therefore, cannot be applied when replacing a service which ensures that packets are delivered in the same order they are sent, or when the network or its applications do not tolerate packets re-ordering.

Modifications to reconfiguration phases

This customization involves more than simply swapping the order in which NeCoMan executes its finishing and activation actions. To illustrate this, we first explain how NeCoMan implements the four reconfiguration phases when activating a new service before finishing the old one. This will be illustrated (again) with the replacement of the reliability service.

a) Installation phase. When an old network service is finished while the new one is already active, both versions will (temporarily) execute in parallel during reconfiguration. While this occurs, packets in transit may be processed by the old

or by the new service components. Support is needed, therefore, to distinguish between these packets. This includes support to

- mark transmitted packets to identify the service components they were processed by,
- de-multiplex these packets when they arrive at their destination, and
- delegate them to the appropriate service component.

Therefore, when switching the order of finishing and activation, NeCoMan extends the installation phase such that all affected nodes become instructed to integrate dedicated "marking" and "dispatching" components into their stack composition (in addition to loading and connecting the new service components). These marking components label all packets passing by, and thus have one inport and one outport[2]. Dispatching components, in contrast, de-multiplex incoming packets and delegate them to the appropriate (old or new) service component according to the packet's label. Hence, these components have one inport and two outports.

To be precise, NeCoMan instructs the affected nodes to only mark those packets that the *new* service components have processed. Marking components, therefore, only become connected to the service-internal outports of new service components, as illustrated in Figure 6.1(b). When replacing the reliability service, for instance, marking components (symbolized as MC) are connected to the data-outport of R_{new}, and to the ack-outport of A_{new}. This way, the new data and acknowledgement packets will be labelled when both new reliability components are brought into use. Likewise, dispatching components (symbolized as DC) delegate packets without a label to the old components' service-internal inports, while marked packets are delivered to the (service-internal inports of the) new reliability components. As illustrated in Figure 6.1(b), these dispatching components thus replace the connectors that mediate incoming data and acknowledgement packets to A_{old} and R_{old}, respectively.

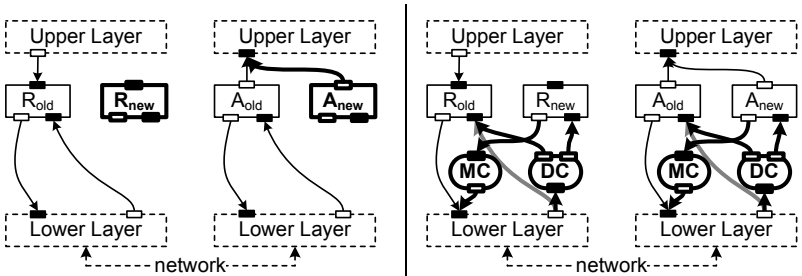
NeCoMan implements this customized installation phase by executing CC_{new}, LO^{ext}_{new}, APD^{mark}_{new}, and $APD^{disp}_{old-new}$, on every node where needed. With this

- APD^{mark}_{new} symbolizes a reconfiguration action responsible for integrating marking components into the node's protocol stack composition, and
- $APD^{disp}_{old-new}$ denotes a reconfiguration action for integrating dispatching components into the node's protocol stack composition.

Figure 6.1 illustrates this customized installation phase. Furthermore, we express the implementation of this customized installation phase as

$$\begin{aligned} IS_{new} \equiv \forall node_x :[&CC_{new}(node_x) \curlywedge LO^{ext}_{new}(node_x) \curlywedge \\ &APD^{mark}_{new}(node_x) \curlywedge APD^{disp}_{old-new}(node_x)] \end{aligned} \tag{P.5}$$

[2]Note that this labelling involves the use of header-bits that are reserved for marking packets during reconfiguration (for example by using IP-options).



(a) Installation of new reliability components

(b) Adding marking and dispatching support

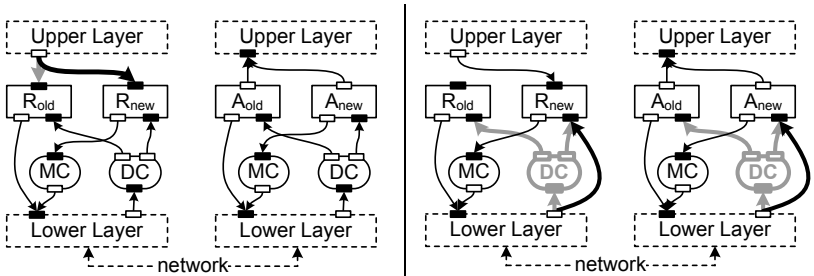
Figure 6.1: Activating new service before finishing old version: installing the new reliability components and the associated packet-distinguishing support.

Note that in contrast to the original installation phase expressed in (P.1), this customized installation phase does not involve the execution of LO^{int}_{new}. This is because service-internal communication ports are already bound by installing marking and dispatching components, which makes the execution of LO^{int}_{new} redundant. Furthermore, new service components are not yet activated in this installation phase. Attaching marking components to these components' service-internal outports, therefore, involves simply creating and connecting these marking components. The integration of dispatching components, however, requires a dynamic recomposition. NeCoMan uses its local reconfiguration algorithm for isolated services tailored with the "service addition" customizations to accomplish this.

Finally, because the new service components are not activated in this phase, the distributed execution of CC_{new}, LO^{ext}_{new}, and APD^{mark}_{new} must not be synchronized. Furthermore, since dispatching components deliver original packets to the old service components, the distributed execution of $APD^{disp}_{old-new}$ does not have to be synchronized either to preserve a correct service composition. The local execution of these reconfiguration actions, in contrast, must be initialized in a specific order. NeCoMan can only bind a component's service-external outports when the associated component has been created. The same pre-condition holds for adding marking and dispatching components. We express this local reconfiguration condition as

$$\begin{aligned} \forall node_x : [&LO^{ext}_{new}(node_x) \wedge APD^{mark}_{new}(node_x) \wedge APD^{disp}_{old-new}(node_x) \\ &\leftarrow CC_{new}(node_x)] \end{aligned} \tag{6.1}$$

b) Activation phase. When activating a new service before the old one is finished, the activation phase involves only the execution of $LI^{ext}_{old-new}$. This is because in contrast to the original activation phase expressed in (P.3), the execution



(a) Activating the new reliability service (b) Removing dispatching support

Figure 6.2: Activating new service before finishing old version: activation of the new reliability service and removal of dispatching support.

of $LI^{int}_{old-new}$ becomes redundant; the new components' service-internal inports are already bound by installing dispatching components. Besides, the execution of RP^{client}_{new} can be omitted as well since no packets are intercepted anymore during reconfiguration. So, this customized activation phase involves only

- binding the service-external inports of the affected components such that packets are delivered exclusively to the new client processes, and
- starting the active objects that these new components employ.

Figure 6.2(a) illustrates (part of) the activation of the new reliability service. Furthermore, we express the implementation of this customized activation phase as

$$AS_{new} \equiv \forall node_x : [LI^{ext}_{old-new}(node_x) \curlywedge AP^{client}_{new}(node_x) \curlywedge AP^{server}_{new}(node_x)] \tag{P.6}$$

NeCoMan must coordinate the order in which these reconfiguration actions are executed. To be precise, packets can only be (re)directed towards the service-external inports of new client processes once the active objects of both these client processes and their collaborating server processes have been initiated. Hence, when activating the new service before finishing the old one, reconfiguration condition (5.4) becomes replaced with

$$\begin{aligned} \forall node_x :[LI^{ext}_{old-new}(node_x) \leftarrow AP^{client}_{new}(node_x) \wedge \\ \forall node_y \neq node_x : AP^{server}_{new}(node_y)] \end{aligned} \tag{6.2}$$

c) Finishing phase. Finishing the old service after the new one is brought into use makes the execution of IP^{client}_{old} redundant. This is because finishing the old

service does not involve intercepting packets anymore. Once the new reliability components are brought into use, no more packets will be delivered to the service-external inports of the old retransmission process. The old client processes, therefore, do not have to be prevented anymore from accepting new packets to bring about a quiescent execution state. Hence, we express the implementation of this customized finishing phase as

$$FS_{old} \equiv \forall node_x : [ISS_{old}^{client}(node_x) \curlywedge ISS_{old}^{server}(node_x)] \tag{P.7}$$

Besides, note that by omitting the execution of IP_{old}^{client}, reconfiguration conditions (5.2) does not apply anymore.

d) Removal phase. Once every old component is finished, all packets in transit belong to the new service components. From this moment on, packet-distinguishing support thus becomes redundant and can safely be removed. So, when switching the order of finishing and activation, NeCoMan extends the removal phase to include the removal of marking and dispatching components as well (besides disconnecting and deleting the old service components). The resulting removal phase includes the execution of DC_{old}, UO_{old}^{ext}, UO_{old}^{int}, RPD_{new}^{mark}, and $RPD_{old-new}^{disp}$ on every node where needed. With this

- RPD_{new}^{mark} symbolizes a reconfiguration action responsible for removing the marking components from the node's protocol stack composition, and
- $RPD_{old-new}^{disp}$ denotes a reconfiguration action for removing the employed dispatching components.

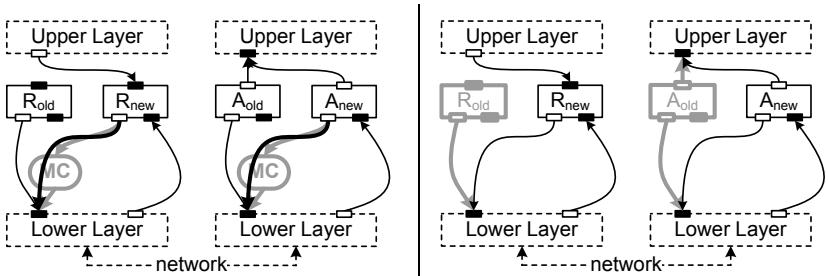
Note that the execution of RPD_{new}^{mark} and $RPD_{old-new}^{disp}$ requires dynamic recomposition. NeCoMan uses its local reconfiguration algorithm for isolated services tailored with the "service removal" and "activate before finishing" customization to accomplish this.

We express the implementation of this customized removal phase as

$$\begin{aligned} RS_{old} \equiv \forall node_x :[&DC_{old}(node_x) \curlywedge UO_{old}^{ext}(node_x) \curlywedge UO_{old}^{int}(node_x) \curlywedge \\ &RPD_{new}^{mark}(node_x) \curlywedge RPD_{old-new}^{disp}(node_x)] \end{aligned} \tag{P.8}$$

Furthermore, to illustrate this customized removal phase when replacing the reliability service, Figure 6.2(b) sketches the composition of both nodes after executing $RPD_{old-new}^{disp}$, while Figure 6.3(a) depicts the effect of executing RPD_{new}^{mark}. To complete this example, Figure 6.3(b) illustrates the execution of DC_{old}, UO_{old}^{ext}, and UO_{old}^{int} as well.

To conduct safe reconfigurations, NeCoMan must synchronize the distributed execution of RPD_{new}^{mark} and $RPD_{old-new}^{disp}$. The marking support on node x can only be removed safely when there is no more dispatching support left on every node y



(a) Removing marking support (b) Deleting old reliability components

Figure 6.3: Activating new service before finishing old version: removing marking support and deleting old reliability components.

where node x sends packets to. We express this additional reconfiguration condition as

$$\forall node_x : [RPD^{mark}_{new}(node_x) \leftarrow \forall node_y \neq node_x : RPD^{disp}_{old-new}(node_y)] \quad (6.3)$$

This reconfiguration condition results from marking (during reconfiguration) only packets that new service components transmit. Suppose this rule is ignored and the marking component MC, which is attached to R_{new}, becomes removed before the dispatching components DC, which directs packets towards A_{old} or A_{new}, has been taken out. When this occurs, DC will incorrectly deliver unmarked data packets that are processed by R_{new} towards A_{old} (which is quiescent). This breaks the correct functioning of the reliability service.

Besides coordinating the distributed execution of these reconfiguration actions, their local execution must be coordinated as well to correctly remove the old service components. To be precise, NeCoMan can only delete an old component once this component is disconnected – that is, after completing the execution of UO^{ext}_{old}, $LI^{ext}_{old-new}$, $RPD^{disp}_{old-new}$ and UO^{int}_{old}. Consequently, reconfiguration condition (5.5) becomes replaced with

$$\forall node_x : [DC_{old}(node_x) \leftarrow UO^{ext}_{old}(node_x) \wedge LI^{ext}_{old-new}(node_x) \wedge RPD^{disp}_{old-new}(node_x) \wedge UO^{int}_{old}(node_x)] \quad (6.4)$$

Modifications to reconfiguration conditions

Furthermore, activating the new service before the old one is finished affects many reconfiguration conditions, which in turn will modify the partial ordering of the reconfiguration actions that are involved. First, condition (H.2) becomes replaced

with (H.4)

$$FS_{old} \leftarrow AS_{new} \tag{H.4}$$

Besides, all reconfiguration conditions that NeCoMan must fulfill to correctly execute the original distributed reconfiguration algorithm become obsolete, except for (5.3), (5.7), (5.13) and (5.14).

To identify the new (additional) reconfiguration conditions that NeCoMan must fulfill when it applies this customization, we refine conditions (H.1), (H.4), and (H.3) in terms of the reconfiguration actions that implement the customized versions of IS_{new}, AS_{new}, FS_{old}, and RS_{old}. This involves first substituting these four high-level reconfiguration actions by expressions (P.5), (P.6), (P.7) and (P.8), respectively. After reducing the resulting conditions, (H.1) produces condition (6.5)

$$\begin{aligned}\forall node_x :[&LI^{ext}_{old-new}(node_x) \leftarrow APD^{mark}_{new}(node_x) \wedge APD^{disp}_{old-new}(node_x) \wedge \\ &\forall node_y \neq node_x : [LO^{ext}_{new}(node_y) \wedge APD^{mark}_{new}(node_y) \wedge \\ &APD^{disp}_{old-new}(node_y)]]\end{aligned} \tag{6.5}$$

which defines that the service-external inports of the new (reliability) components can only be bound once all new collaborating processes are prepared to process each others invocations.

Condition (H.4), in turn, produces reconfiguration condition (6.6)

$$\forall node_x : \quad [ISS^{client}_{old}(node_x) \leftarrow LI^{ext}_{old-new}(node_x)] \tag{6.6}$$

which impose to only monitor the old client processes for quiescence once these processes are prevented from accepting and processing new service requests.

Finally, refining condition (H.3) produces reconfiguration condition (6.7)

$$\begin{aligned}\forall node_x :[&RPD^{mark}_{new}(node_x) \wedge RPD^{disp}_{old-new}(node_x) \leftarrow \\ &ISS^{client}_{old}(node_x) \wedge ISS^{server}_{old}(node_x)]\end{aligned} \tag{6.7}$$

which dictates NeCoMan to only remove the employed marking and dispatching components on node x once the old processes on that node are quiescent.

Result

To define the partial ordering of reconfiguration actions that NeCoMan must fulfill when it applies this customization, Table 6.1 summarizes all associated reconfiguration conditions. The resulting ordering when recomposing a node x as part of a distributed reconfiguration is sketched in Figure 6.4. Furthermore, Figure 6.5 depicts the Petri net that models the (resulting) algorithm itself. Note that this algorithm uses three synchronization messages: **C**, **A**, and **E**. Message **C** serves

	reconfiguration condition	place
(5.3)	$\forall node_x : [ISS^{server}_{old}(node_x) \leftarrow \forall node_y \neq node_x : ISS^{client}_{old}(node_y)]$	p_9/p_{24}
(5.7)	$\forall node_x : [AP^{client}_{new}(node_x) \wedge AP^{server}_{new}(node_x) \leftarrow CC_{new}(node_x)]$	p_2/p_{17}
(5.13)	$\forall node_x : [UO^{int}_{old}(node_x) \leftarrow ISS^{client}_{old}(node_x) \wedge ISS^{server}_{old}(node_x)]$	p_9/p_{24}
(5.14)	$\forall node_x : [UO^{ext}_{old}(node_x) \leftarrow ISS^{server}_{old}(node_x)]$	p_{10}/p_{25}
(6.1)	$\forall node_x : [LO^{ext}_{new}(node_x) \wedge APD^{mark}_{new}(node_x) \wedge APD^{disp}_{old-new}(node_x) \leftarrow CC_{new}(node_x)]$	p_2/p_{17}
(6.2)	$\forall node_x : [LI^{ext}_{old-new}(node_x) \leftarrow AP^{client}_{new}(node_x) \wedge \forall node_y \neq node_x : AP^{server}_{new}(node_y)]$	p_7/p_{22}
(6.3)	$\forall node_x : [RPD^{mark}_{new}(node_x) \leftarrow \forall node_y \neq node_x : RPD^{disp}_{old-new}(node_y)]$	p_{11}/p_{26}
(6.4)	$\forall node_x : [DC_{old}(node_x) \leftarrow UO^{ext}_{old}(node_x) \wedge LI^{ext}_{old-new}(node_x) \wedge RPD^{disp}_{old-new}(node_x) \wedge UO^{int}_{old}(node_x)]$	p_{14}/p_{29}
(6.5)	$\forall node_x : [LI^{ext}_{old-new}(node_x) \leftarrow APD^{mark}_{new}(node_x) \wedge APD^{disp}_{old-new}(node_x) \wedge \forall node_y \neq node_x : [LO^{ext}_{new}(node_y) \wedge APD^{mark}_{new}(node_y) \wedge APD^{disp}_{old-new}(node_y)]]$	p_7/p_{22}
(6.6)	$\forall node_x : [ISS^{client}_{old}(node_x) \leftarrow LI^{ext}_{old-new}(node_x)]$	p_8/p_{23}
(6.7)	$\forall node_x : [RPD^{mark}_{new}(node_x) \wedge RPD^{disp}_{old-new}(node_x) \leftarrow ISS^{client}_{old}(node_x) \wedge ISS^{server}_{old}(node_x)]$	p_{10}/p_{25}

Table 6.1: Customization of NeCoMan's algorithm for conducting synchronized distributed reconfigurations: overview of all reconfiguration conditions that must be fulfilled when activating the new service before the old reaches quiescence. The right column lists the places as form which the associated pre-condition is fulfilled.

to satisfy reconfiguration conditions (6.2) and (6.5). Message **A**, in turn, is required to meet reconfiguration condition (5.3), while message **E** is needed to fulfill condition (6.3).

Finally, the right column of Table 6.1 specifies for each reconfiguration condition the place (of this Petri net model) as from which the associated pre-condition is fulfilled. This enables to verify that the algorithm modelled in Figure 6.5 meets all required reconfiguration conditions.

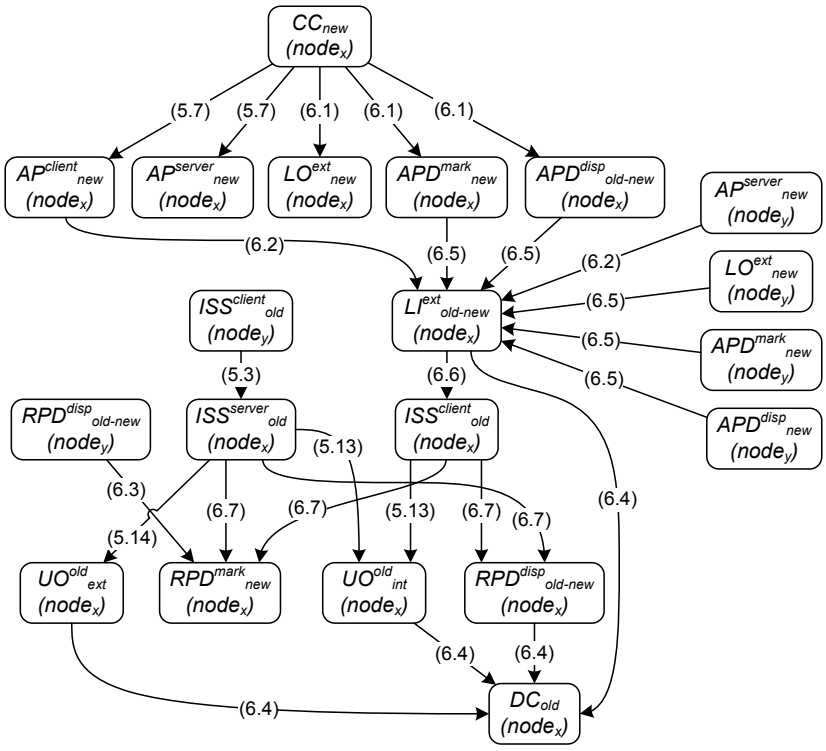


Figure 6.4: Customization of NeCoMan's algorithm for conducting synchronized distributed reconfigurations: overview of the partial ordering of reconfiguration actions that the reconfiguration conditions listed in Table 6.1 define.

6.3.2 Independent distributed reconfigurations

Activating new service components before the old ones are finished can be applied as well when NeCoMan recomposes all affected nodes independently. The preconditions to safely switch the order of activation and finishing actions are in this case identical as for local reconfigurations of distributed services. For the same reason the effect of this customization differs in nothing from the effect explained in Section 4.2.1.

Note, however, that NeCoMan only applies this customization to its independent reconfiguration algorithm if the latter results from customization "no distributed activation". When NeCoMan uses its independent distributed reconfiguration algorithm because the underlying nodes employ state transfer to finish the old service, in contrast, then the new service components cannot be activated before finishing the old one. Similar as for synchronized distributed reconfigurations, this would compromise the service's correct functioning.

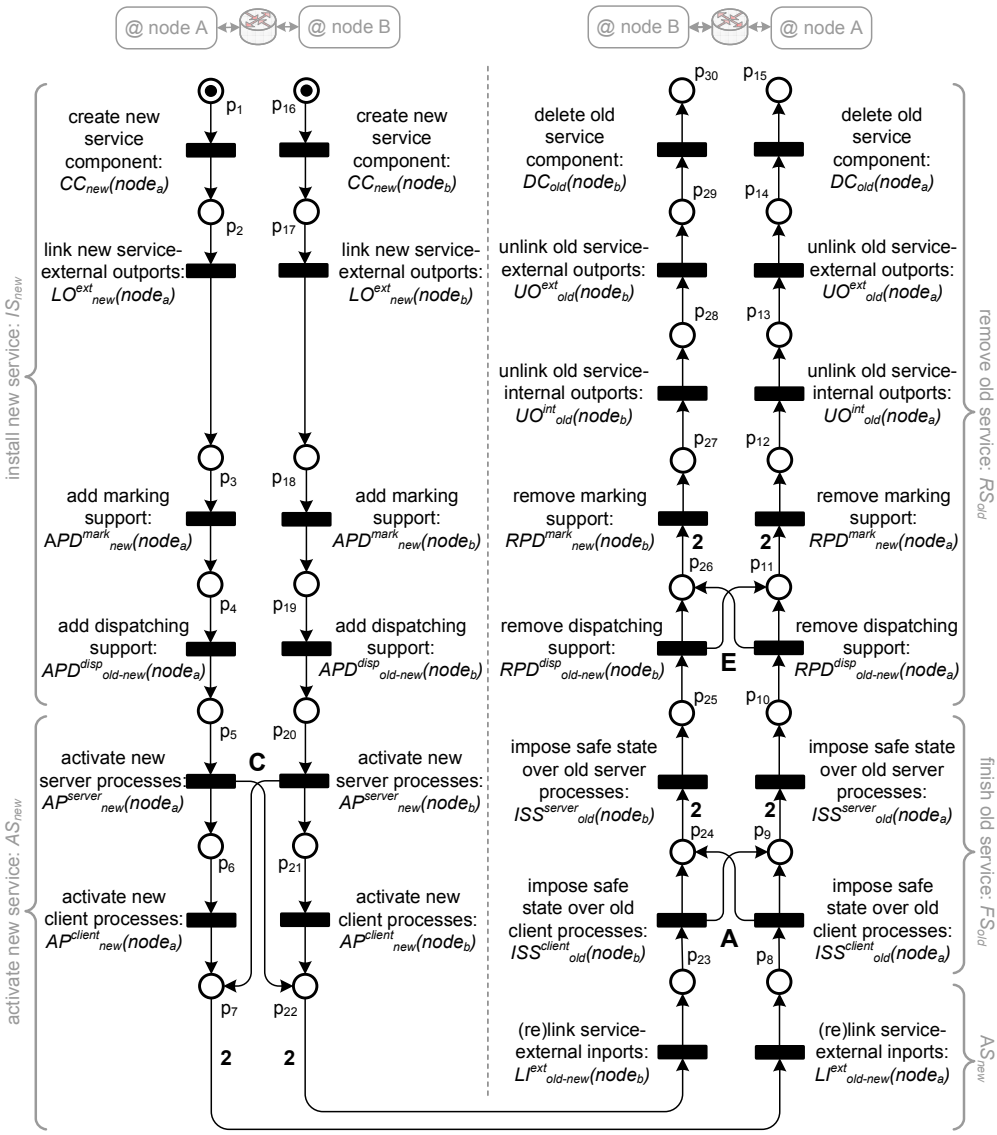


Figure 6.5: Customization of NeCoMan's algorithm for conducting synchronized distributed reconfigurations: Petri net representation of the resulting algorithm when activating the new components before finishing the old ones.

6.4 No finishing

A next customization involves omitting all finishing actions. When the new network service or the network itself can deal with inconsistencies that may occur during

reconfiguration, there is no need for the reconfiguration middleware to preserve consistency. In that case, NeCoMan can safely omit its finishing actions, thus reducing the overhead that a reconfiguration causes.

6.4.1 Synchronized distributed reconfigurations

Pre-conditions

The pre-conditions that must be fulfilled to safely apply this customization when conducting synchronized distributed reconfigurations are similar as when NeCoMan executes local reconfigurations of distributed services. First, the affected service components must operate in a best-effort network, such that packet loss caused by omitting all finishing actions will not compromise the correct functioning of the network. Second, the network or the new network service must be able to deal with all ongoing protocol-transactions. This is because when finishing actions are discarded, there is no knowledge about the status of these unfinished protocol-transactions at the moment when new service components are brought into use.

To illustrate this, consider the addition of a compression service on two nodes. When no finishing is involved, there may possibly be packets in transit that are not compressed when the new decompression process becomes activated. This causes no problem, however, when the new decompression process is able to service those uncompressed packets as well. As an additional example, consider again the removal of a compression service. When the compression service is not finished, there may still be compressed packets in transit when removing the affected decompression process. This can be solved by filtering out all compressed packets that reach the edge of the programmable network.

Finally, NeCoMan can only omit its finishing actions when this causes no inconsistent execution states or when the network tolerates or restores from inconsistent execution states. This is fulfilled, among others, when the affected service components do not share their execution state (that is, not with each other and not with other network service components).

Modifications

Discarding all finishing actions brings along the following changes. First, conditions (H.2) and (H.3) become redundant and are therefore omitted. Instead, we replaces these (high-level) conditions with (H.5)

$$RS_{old} \leftarrow AS_{new} \tag{H.5}$$

which dictates to only remove the old network service components once the new ones are active. Similar to local reconfigurations, this condition is not required to execute a correct reconfiguration, but is essential to minimize communication disruption. Second, omitting all finishing actions includes discarding the execution of IP_{old}^{client}, ISS_{old}^{client}, and ISS_{old}^{server}. Finally, since packets are not intercepted

anymore, the execution of RP_{old}^{client} becomes redundant as well. NeCoMan therefore customizes the implementation of its activation phase. We express this resulting implementation as

$$AS_{new} \equiv \forall node_x : [LI_{old-new}^{int}(node_x) \curlywedge LI_{old-new}^{ext}(node_x) \curlywedge AP_{new}^{client}(node_x) \curlywedge AP_{new}^{server}(node_x)] \tag{P.9}$$

Result

All these changes affect reconfiguration conditions (5.2), (5.3), (5.4), (5.8), (5.9), (5.10), (5.11), (5.12), (5.13), and (5.14). To be precise, these conditions become replaced with (6.8), (6.9), (6.10), and (6.11). Let us briefly explain these reconfiguration conditions.

Reconfiguration conditions (6.8) and (6.9). Since packets are not intercepted anymore during reconfiguration when all finishing actions are omitted, a component's service-internal inports can only be bound once this component is able to process invocations (which belong to ongoing protocol-transactions). This implies that its outports must be bound, and its active objects (if any) must be started. We express this reconfiguration condition as follows

$$\forall node_x : [LI_{old-new}^{int}(node_x) \leftarrow LO_{new}^{ext}(node_x) \wedge LO_{new}^{int}(node_x) \wedge AP_{new}^{client}(node_x) \wedge AP_{new}^{server}(node_x)] \tag{6.8}$$

Furthermore, a new component's service-external inports can only be bound once the execution of new protocol-transactions is enabled. This can be expressed as

$$\forall node_x : [LI_{old-new}^{ext}(node_x) \leftarrow LO_{new}^{int}(node_x) \wedge AP_{new}^{client}(node_x) \wedge LI_{old-new}^{int}(node_x) \wedge \forall node_y \neq node_x : LI_{old-new}^{int}(node_y)] \tag{6.9}$$

Reconfiguration conditions (6.10) and (6.11). Reconfiguration conditions (6.10) and (6.11), in turn, result from refining condition (H.5) in terms of the reconfiguration actions to implement AS_{new} and RS_{old}. This involves substituting AS_{new} and RS_{old} by expressions (P.9) and (P.4), respectively. Reducing the resulting condition then results in

$$\forall node_x : \quad [UO_{old}^{ext}(node_x) \leftarrow LI_{old-new}^{int}(node_x)] \tag{6.10}$$

$$\forall node_x : \quad [UO_{old}^{int}(node_x) \leftarrow LI_{old-new}^{ext}(node_x) \wedge LI_{old-new}^{int}(node_x)] \tag{6.11}$$

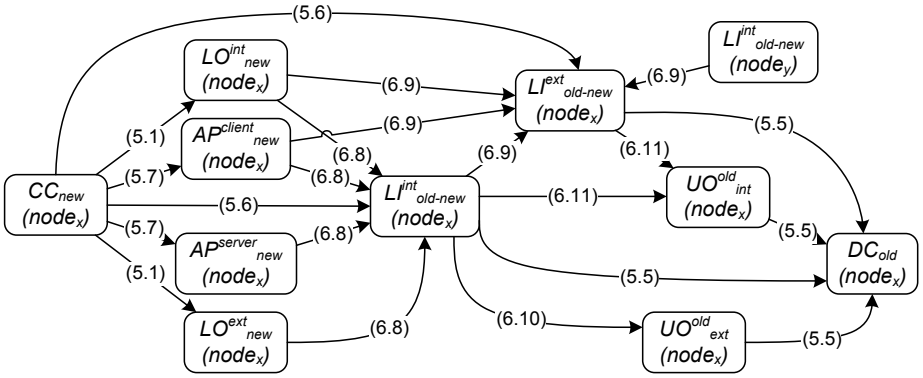


Figure 6.6: Customization of NeCoMan's algorithm for conducting synchronized distributed reconfigurations: overview of the partial ordering of reconfiguration actions that the reconfiguration conditions listed in Table 6.2 define.

Reconfiguration condition (6.10) imposes to only unlink a component's *service-external outports* when its service-internal inports are disconnected. This is because only server processes expose service-external outports. Besides, these server processes can only be invoked via the component's service-internal inports (see Figure 2.16). Reconfiguration condition (6.11), in contrast, instructs to only unlink a component's *service-internal outports* once both its service-external and service-internal inports have been disconnected. This is because service-internal outports may belong to client as well as to server processes (as illustrated in Figure 2.16). Besides, these client and server processes can be invoked via the component's service-external and service-internal inports.

Partial ordering and resulting algorithm. To conclude, Table 6.2 summarizes all reconfiguration conditions that NeCoMan must fulfill to correctly execute a coordinated distributed reconfiguration that involves no finishing. The resulting ordering of reconfiguration actions when recomposing a node x as part of a distributed reconfiguration is sketched in Figure 6.6. In addition, Figure 6.7 depicts the Petri net that models NeCoMan's coordinated distributed reconfiguration algorithm after applying this customization. Note that the resulting algorithm uses only one synchronization message. This message **F** serves to satisfy reconfiguration condition (6.9).

6.4.2 Independent distributed reconfigurations

The pre-conditions to omit finishing actions when conducting independent distributed reconfigurations differ in nothing from those listed in Section 6.4.1. Besides, the effect of this customization on independent distributed reconfigurations is iden-

	reconfiguration condition	place
(5.1)	$\forall node_x : [LO^{ext}_{new}(node_x) \wedge LO^{int}_{new}(node_x) \leftarrow CC_{new}(node_x)]$	p_2/p_{13}
(5.5)	$\forall node_x : [DC_{old}(node_x) \leftarrow UO^{ext}_{old}(node_x) \wedge UO^{int}_{old}(node_x) \wedge LI^{ext}_{old-new}(node_x) \wedge LI^{int}_{old-new}(node_x)]$	p_{10}/p_{21}
(5.6)	$\forall node_x : [LI^{int}_{old-new}(node_x) \wedge LI^{ext}_{old-new}(node_x) \leftarrow CC_{new}(node_x)]$	p_2/p_{13}
(5.7)	$\forall node_x : [AP^{client}_{new}(node_x) \wedge AP^{server}_{new}(node_x) \leftarrow CC_{new}(node_x)]$	p_2/p_{13}
(6.8)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow LO^{ext}_{new}(node_x) \wedge LO^{int}_{new}(node_x) \wedge AP^{client}_{new}(node_x) \wedge AP^{server}_{new}(node_x)]$	p_6/p_{17}
(6.9)	$\forall node_x : [LI^{ext}_{old-new}(node_x) \leftarrow LO^{int}_{new}(node_x) \wedge AP^{client}_{new}(node_x) \wedge LI^{int}_{old-new}(node_x) \wedge \forall node_y \neq node_x : LI^{int}_{old-new}(node_y)]$	p_7/p_{18}
(6.10)	$\forall node_x : [UO^{ext}_{old}(node_x) \leftarrow LI^{int}_{old-new}(node_x)]$	p_7/p_{18}
(6.11)	$\forall node_x : [UO^{int}_{old}(node_x) \leftarrow LI^{ext}_{old-new}(node_x) \wedge LI^{int}_{old-new}(node_x)]$	p_8/p_{19}

Table 6.2: Customization of NeCoMan's algorithm for conducting synchronized distributed reconfigurations: overview of all reconfiguration conditions that have to be fulfilled when finishing actions are discarded.

tical to when local reconfigurations of distributed services are involved. The latter has been presented in Section 4.3.1.

6.5 No finishing of server processes

As stated before, driving a distributed service to a quiescent execution state involves monitoring the affected client and server processes until all ongoing protocol-transactions have completed. For some services, however, reaching quiescence does not require to monitor the affected server processes, which makes the execution of ISS^{server}_{old} redundant. To illustrate this, consider (again) the replacement of a reliability service. Driving this service to a quiescent execution state requires to monitor only R_{old} (instead of A_{old} as well). Once R_{old}'s retransmission queue is empty, all transmitted data packets have arrived correctly, which indicates that both R_{old} and A_{old} are in a quiescent execution state. Consequently, NeCoMan can in this case safely omit the execution of ISS^{server}_{old}.

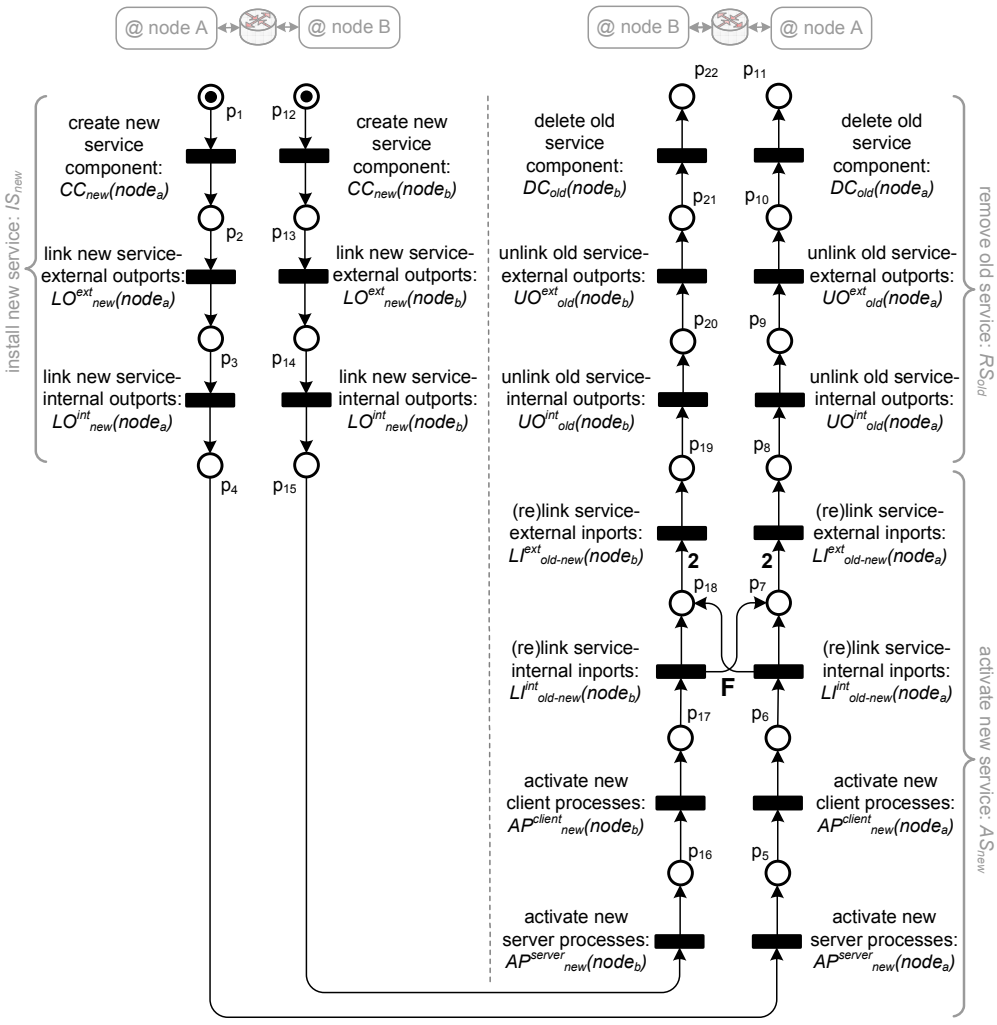
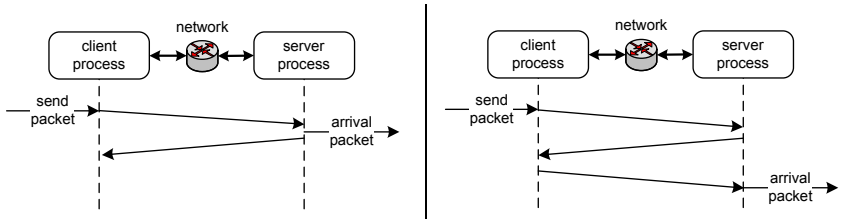


Figure 6.7: Customization of NeCoMan's algorithm for conducting synchronized distributed reconfigurations: Petri net representation of the resulting algorithm when all finishing actions are omitted.

Pre-conditions

A first pre-condition to apply this customization relates to the communication protocol that the old processes employ. NeCoMan can only safely omit the execution of ISS^{server}_{old} when these processes communicate by a protocol that terminates locally. If so, the last message of each protocol-transaction arrives to the client process



(a) Local protocol termination (b) Remote protocol termination

Figure 6.8: Communication characteristics: local versus remote protocol termination

that has initiated this protocol-transaction (see Figure 6.8(a)). Hence, verifying if all ongoing protocol-transactions have completed (and thus quiescence is reached) involves monitoring (only) the affected client processes. When the old service employs a protocol that terminates remotely (as illustrated in Figure 6.8(b)), however, checking if all ongoing protocol-transactions have completed requires to monitor the affected server processes. So, in this case ISS_{old}^{server} cannot be omitted.

An extra pre-condition to apply this customization includes that the old server processes do not encapsulate state which goes beyond the execution of ongoing protocol-transactions. Otherwise this state must be transferred from the old towards the new server processes to preserve consistency. This makes the execution of ISS_{old}^{server} necessary.

Furthermore, since this customization affects how to reach quiescence, it only applies to NeCoMan's synchronized distributed reconfiguration algorithm. Note also that this customization (obviously) cannot be combined with the previous customization which involves omitting all finishing actions. Finally, this customization cannot be applied either in case of service addition, as we further discuss later on[3].

Modifications

As stated before, NeCoMan omits the execution of ISS_{old}^{server} when applying this customization. Furthermore, recall that this customization assumes that the old server processes located on node x are quiescent once ISS_{old}^{client} is completed on every node $y \neq x$. Hence, for each reconfiguration condition that imposes to wait until ISS_{old}^{server} has completed on node x, $ISS_{old}^{server}(node_x)$ becomes replaced with $ISS_{old}^{client}(node_y)$, $\forall node_y \neq node_x$. To illustrate the impact of this, Table J.1 lists all affected reconfiguration conditions as well as the resulting ones. Besides, Figure G.2 models the algorithm that NeCoMan uses to replace the reliability service, which does not include the use of ISS_{old}^{server}.

[3] in Section 6.9, to be precise.

6.6 No active objects

The following customization involves the absence of active objects. Similar to local reconfigurations, NeCoMan can safely omit the execution of AP_{new}^{client} and/or AP_{new}^{server} when the new client and/or server processes do not employ active objects.

Pre-conditions

No additional pre-condition must be fulfilled to apply this customization – that is, besides the (obvious) requirement that the new component's client and/or server processes do not employ active objects. Note that this pre-condition is always fulfilled when a reconfiguration involves the removal of a distributed network service.

Modifications

When applying this customization, NeCoMan thus omits the execution of AP_{new}^{client} and/or AP_{new}^{server}. To illustrate the effect of this customization on NeCoMan's synchronized distributed reconfigurations, Table J.2 lists all reconfiguration conditions that are affected when client processes do not use active objects. For each of these conditions, the resulting reconfiguration conditions (if any) are presented in the right column of this table. Similarly, Table J.3 lists all reconfiguration conditions that are affected when new server processes do not employ active objects. Finally, Tables D.1 and D.2 illustrate the effect of omitting AP_{new}^{client} and/or AP_{new}^{server} when NeCoMan conducts independent distributed reconfigurations.

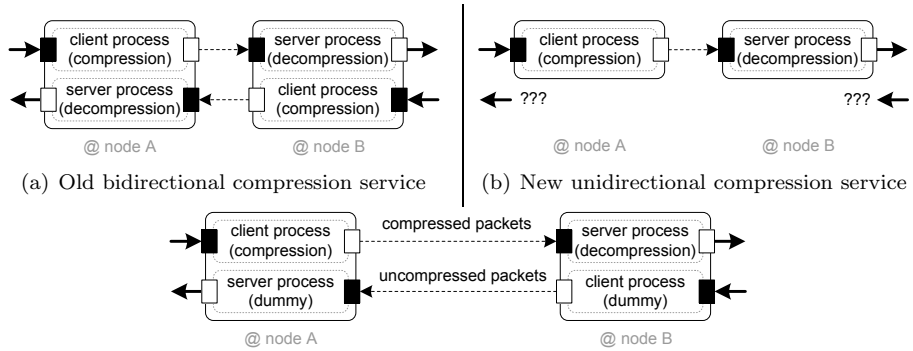
6.7 Only client or server processes instead of both

The next customization involves omitting the reconfiguration actions that become redundant when the affected service components encapsulate only client or server processes instead of both.

6.7.1 Synchronized distributed reconfigurations

Pre-conditions

No additional pre-condition must be fulfilled to apply this customization – that is, besides the requirement that the old and new service components encapsulate only client or server processes instead of both. Note that the old as well as the new component must fulfill this requirement, as otherwise structural integrity cannot be preserved. To illustrate this, consider the replacement of a bidirectional compression service (as illustrated in Figure 6.9(a)) by a unidirectional version (depicted in Figure 6.9(b)). In this case, the old service components encapsulate both a client and a server process, while the new ones contain only a client or a server process



(a) Old bidirectional compression service

(b) New unidirectional compression service

(c) New unidirectional compression service dealing with both up and downstream packet flows

Figure 6.9: Replacing a compression service with a new one.

instead of both. To preserve structural integrity, however, the new service components must provide (at least) every service-external communication port that the old service components offer. Replacing the service depicted in Figure 6.9(a) by the one modelled in Figure 6.9(b), therefore, cannot be accomplished as is. Instead, the components of the new unidirectional compression service must encapsulate dummy client and server processes to deal with upstream packets as well. This is illustrated in Figure 6.9(c).

Modifications

Depending on whether NeCoMan manages a node that accommodates only client or server processes, it omits different reconfiguration actions. When only client processes are involved when recomposing node x, the execution of ISS_{old}^{server} and AP_{new}^{server} become redundant for that node and therefore can safely be discarded. Besides, since only server processes expose service-external outports, NeCoMan also omits the execution of LO_{new}^{ext} and UO_{old}^{ext}. This is illustrated in Figure G.2, which models NeCoMan's algorithm for replacing the reliability service. To further illustrate the impact of this customization, Table J.4 lists all reconfiguration conditions that are affected when the node that NeCoMan manages accommodates only client processes.

When the node that NeCoMan manages hosts only server processes, then NeCoMan omits the execution of IP_{old}^{client}, ISS_{old}^{client}, AP_{new}^{client}, and RP_{new}^{client}. Besides, since only client processes expose service-external inports, NeCoMan also omits the execution of $LI_{old-new}^{ext}$ on that node. Table J.5 presents all reconfiguration conditions that become changed when applying this customization.

Note that this customization also affects how many synchronization messages must be exchanged during reconfiguration. For example, consider a node x with

only client processes and a node y with only server processes. Instead of sending two instances of message **A** (one from x to y, and another one in the opposite direction), NeCoMan sends this message only from node x towards node y in this case. This is because reconfiguration condition (5.3) does not apply anymore to node x. Likewise, in this case NeCoMan sends message **B** only from node y towards node x. Since node y accommodates only server processes, reconfiguration conditions (5.4) and (5.8) as well as the associated synchronization can be discarded on that node. To illustrate all this, we refer to Figure G.2, which models NeCoMan's algorithm for replacing the reliability service.

Similarly, combining this customization with the "activate before finishing" customization affects how many instances of synchronization message **C** that NeCoMan must exchange[4]. Consider again a node x and y with only client and server processes, respectively. In that case, NeCoMan must only send message **C** from node y towards node x. This is because reconfiguration conditions (6.2) and (6.5) do not apply anymore to node y, which makes the need to send message **C** from node x to y redundant. To illustrate all this, we refer to Figure H.2. This model depicts NeCoMan's algorithm for replacing the reliability service, which involves activating the new service before finishing the old one.

Finally, combining this customization with customization "no finishing" affects how many instances of synchronization message **F** must be sent. This is because reconfiguration condition (6.9) only relates to the nodes hosting client processes. So, when the affected components encapsulate only client or server processes, NeCoMan sends message **F** only from the nodes hosting the server processes towards those accommodating the client processes.

6.7.2 Independent distributed reconfigurations

The effect of this customization on independent distributed reconfigurations is identical to when local reconfigurations of distributed services are involved. The latter has been presented in Section 4.5.

6.8 Only service-internal inports or outports instead of both

The following customization is targeted at omitting reconfiguration actions that involve service-internal inports and outports. This customization, therefore, affects the execution of LO^{int}_{new}, UO^{int}_{old}, $LI^{int}_{old-new}$, APD^{mark}_{new}, $APD^{disp}_{old-new}$, RPD^{mark}_{new}, and $RPD^{disp}_{old-new}$.

[4]Messages **C** is depicted in Figure 6.5

6.8.1 Synchronized distributed reconfigurations

Pre-conditions

These reconfiguration actions can only be omitted if the **old and new** service components encapsulate only client or server processes instead of both. In addition, the **old and/or new** service components must communicate by a unidirectional communication protocol. If both conditions are fulfilled, then (some of) the affected components employ only service-internal inports or outports (instead of both). To illustrate this, we refer to Figure 6.9(b).

Note that this customization does not require for both the old as well as the new service components to communicate by a unidirectional communication protocol – that is, unlike when applying this customization to local reconfigurations. This is because a bidirectional service can safely be replaced with a unidirectional service (and vice versa) without compromising the network's structural integrity.

Finally, note that when the old service communicates by a unidirectional communication protocol, the execution of ISS_{old}^{server} cannot be omitted[5]. This is because a unidirectional communication protocol always ends at a remote node, while omitting ISS_{old}^{server} requires for the old service processes to communicate by a protocol that terminates locally.

Modifications

Because this customization is targeted at nodes with only client or server processes, its modifications include omitting the execution of

- ISS_{old}^{server}, AP_{new}^{server}, LO_{new}^{ext} and UO_{old}^{ext} on the nodes accommodating only client processes, and
- IP_{old}^{client}, ISS_{old}^{client}, AP_{new}^{client}, RP_{new}^{client} and $LI_{old-new}^{ext}$ on the nodes where only server processes are involved

as well as discarding the associated use of synchronization messages.

Furthermore, different reconfiguration actions are affected depending on whether the old, the new, or both the old and new service processes communicate by a unidirectional communication protocol. We therefore distinguish between

1. replacing a unidirectional with a unidirectional service (see Figure 6.10),
2. replacing a bidirectional with a unidirectional service (see Figure 6.11), and
3. replacing a unidirectional with a bidirectional service (see Figure 6.12).

For each of these reconfigurations, NeCoMan applies a slightly different customization to its algorithm for conducting synchronized distributed reconfiguration.

[5]This customization has been discussed in Section 6.5.

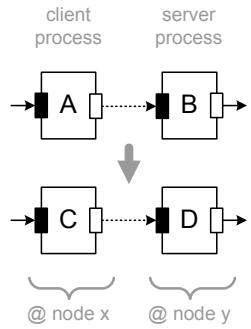


Figure 6.10: Replacing a unidirectional with a unidirectional service

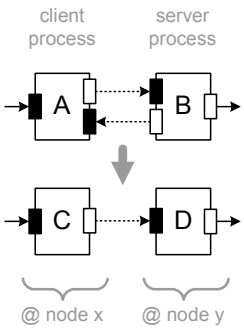


Figure 6.11: Replacing a bidirectional with a unidirectional service

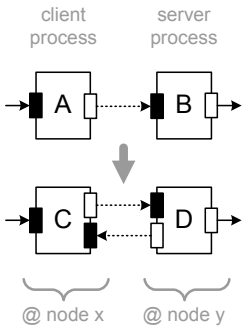


Figure 6.12: Replacing a unidirectional with a bidirectional service

1) Replacing a unidirectional by a unidirectional service. On a node x with only client processes, NeCoMan in this case omits the execution of $LI^{int}_{old-new}$, $APD^{disp}_{old-new}$, and $RPD^{disp}_{old-new}$. These omissions are allowed as none of the new client processes expose service-internal inports (as illustrated in Figure 6.10). To illustrate the impact of this customization, Table J.6 lists all changed reconfiguration conditions.

Likewise, on a node y that accommodates only server processes, NeCoMan discards the execution of LO^{int}_{new}, UO^{int}_{old}, APD^{mark}_{new}, and RPD^{mark}_{new}. These omissions are allowed as none of the new server processes expose service-internal outports (see Figure 6.10). Table J.7 lists all reconfiguration conditions that are changed when applying this customization.

Besides, note that this customization also affects the number of synchronization message **E** that NeCoMan must exchange[6] – that is, given that the new service becomes activated before finishing the old one. To illustrate this, consider the replacement depicted in Figure 6.10. In this case, marking components are only needed on node x, while dispatching components are only required on node y. Hence, NeCoMan must only transmit synchronization message **E** from node y towards node x, so as to notify node x that $RPD^{disp}_{old-new}$ has completed on node y. When a replacement includes bidirectional services, however, NeCoMan must send this message in the opposite direction as well.

2) Replacing a bidirectional by a unidirectional service. In this case, the new client processes do not expose service-internal inports. Likewise, the new server processes do not provide service-internal outports (see Figure 6.11). Hence, on a node with only client processes, NeCoMan (1) replaces $LI^{int}_{old-new}$ with UI^{int}_{old}, (2) omits the execution of $APD^{disp}_{old-new}$, and (3) replaces $RPD^{disp}_{old-new}$ with UI^{int}_{old}.

[6] Message **E** is depicted in Figure 6.5

To illustrate the impact of this customization, Table J.8 lists all reconfiguration conditions that this customization changes (as backing).

Let us explain this customization in short. Since the new client processes do not expose service-internal inports, $LI^{int}_{old-new}$ cannot be executed as is. This is because $LI^{int}_{old-new}$ involves both unbinding the old and binding the new component's service-internal inports. This reconfiguration action, therefore, must be reduced to UI^{int}_{old} – which is responsible (only) for unbinding the old component's service-internal inports[7].

Besides, when the new (unidirectional) service becomes activated before finishing the old (bidirectional) one, only packets that are sent from component C to D must be labelled (both components are depicted in Figure 6.11). Hence, NeCoMan can safely omit the execution of $APD^{disp}_{old-new}$ and $RPD^{disp}_{old-new}$ when recomposing node x. To be precise, $RPD^{disp}_{old-new}$ becomes replaced with UI^{int}_{old} to make sure that the service-internal inports of the old client processes are unlinked.

Furthermore, on a node y with only server processes, NeCoMan discards the execution of LO^{int}_{new} because the new server processes expose no service-internal outports. Besides, when the new service becomes activated before finishing the old one, marking components are not needed on node y for the same reason. Hence, NeCoMan omits the execution of APD^{mark}_{new} and RPD^{mark}_{new} when recomposing node y. To illustrate the impact of this customization, we refer to Table J.9.

Finally, note that synchronization message **E** (again) must only be sent from the nodes that host server processes towards the other ones (hosting the client processes). The reason for this is similar as when replacing a unidirectional service with a new unidirectional version. That is, marked packets are only transmitted in one direction (from the client towards the server processes), such that NeCoMan must only transmit message **E** in the opposite direction to fulfill reconfiguration condition (6.3).

3) Replacing a unidirectional by a bidirectional service. In this case, the old client processes do not expose service-internal inports and the old server processes do not expose service-internal outports. Hence, on a node x with only client processes, $LI^{int}_{old-new}$ cannot be executed as is. This is because $LI^{int}_{old-new}$ involves unlinking the old component's service-internal inports as well as connecting the service-internal inports of the new component. Since the client processes on node x do not expose service-internal inports, NeCoMan replaces $LI^{int}_{old-new}$ by LI^{int}_{new}. The latter is responsible (only) for binding the new component's service-internal inports. The effect of this customization is illustrated in Table J.10.

Besides, when activating the new (bidirectional) service before the old (unidirectional) one is finished, there is no need to employ marking and dispatching components on every affected node. To illustrate this, consider the reconfigura-

[7]Note that $LI^{int}_{old-new}$ encapsulates both UI^{int}_{old} and LI^{int}_{new}. To be precise, UI^{int}_{old} is responsible for unbinding the old component's service-internal inports, while LI^{int}_{new} represents binding the service-internal inports of a new component.

tion depicted in Figure 6.12. During this reconfiguration, only downstream packets (sent from component A to B, and from C to D) will get multiplexed. The packets that D returns to C, however, will not get mixed with other packets in transit (because there are none). So, when replacing a unidirectional with a bidirectional service, NeCoMan must only employ marking and dispatching support along the communication path that both services have in common. This implies that marking components are only needed on the nodes with client processes. Likewise, dispatching components must only be added to and removed from nodes accommodating server processes.

Therefore, when NeCoMan manages a node x that accommodates only client processes which are activated before finishing the old ones, it omits the execution of $APD^{disp}_{old-new}$ and $RPD^{disp}_{old-new}$. To be precise, $APD^{disp}_{old-new}$ becomes replaced with LI^{int}_{new}, so as to bind the service-internal inports of the new component. Table J.10 lists all reconfiguration conditions that are affected by this customization.

On a node y with only server processes, NeCoMan omits the execution of UO^{int}_{old} when replacing a unidirectional with a bidirectional service. This omission is allowed as the (old) server processes of a unidirectional service do not expose service-internal outports. Besides, when activating a new bidirectional service before finishing the old unidirectional one, NeCoMan omits the execution of APD^{mark}_{new} and RPD^{mark}_{new} on node y. Instead, APD^{mark}_{new} becomes replaced with LO^{int}_{new} to bind the service-internal outports of the new server processes. To illustrate the effect of this customization, Table J.11 lists all affected reconfiguration conditions.

Finally, also in this case synchronization message **E** must only be sent from the nodes with server processes towards those accommodating client processes.

6.8.2 Independent distributed reconfigurations

Pre-conditions

NeCoMan only conducts isolated distributed reconfigurations to replace a bidirectional service with a unidirectional one (and vice versa) if structural inconsistencies do not compromise the correct network functioning. To illustrate this, consider the independent recomposition of both nodes depicted in Figure 6.11. When component A becomes replaced with C while B is still in use, none of the messages that B sends out will arrive correctly. NeCoMan therefore only recomposes both these nodes independently when the affected services or the network can deal with such inconsistencies – that is, if customization "no finishing" is applicable as well.

Note that this pre-condition is not required when replacing a unidirectional service with another unidirectional one. In that case, the affected nodes can be recomposed independently if the old and new service components are inter-compatible or if the network can handle inconsistencies.

Modifications

1) Replacing a unidirectional by a unidirectional service. The effect of this customization on independent distributed reconfigurations is identical to when local reconfigurations of distributed services are involved. The latter has already been discussed in Section 4.6.

2) Replacing a bidirectional by a unidirectional service. On a node with only client processes, NeCoMan replaces action $LI^{int}_{old-new}$ with UI^{int}_{old} to conduct this reconfiguration. Table J.12 lists all reconfiguration conditions that have to be fulfilled to correctly conduct this reconfiguration. Likewise, on a node with only server processes, NeCoMan omits the execution of LO^{int}_{new}. To illustrate the effect of this customization, Table J.13 lists all reconfiguration conditions that have to be fulfilled in this case.

3) Replacing a unidirectional with a bidirectional service. On a node with only client processes, NeCoMan replaces $LI^{int}_{old-new}$ with LI^{int}_{new}. For an overview of all reconfiguration conditions that have to be fulfilled to carry out this recomposition, we refer to Table J.14. Besides, on a node with only server processes, NeCoMan omits the execution of UO^{int}_{old}. Table J.15 illustrates the effect of this customization.

6.9 Service addition or removal

The last customizations involve the reconfiguration type – that is, service addition, removal, or replacement. To be precise, NeCoMan applies a different customization in case of service addition and removal, respectively.

6.9.1 Synchronized distributed reconfigurations

Pre-conditions

No additional pre-conditions must be satisfied for NeCoMan to safely apply both customizations. Note, however, that customization "no finishing of server processes" cannot be applied in case of service addition. This is because ISS^{server}_{old} can only be discarded when the old service uses a communication protocol that terminates locally. Adding a new service, however, is similar to replacing a dummy service with this new version. As illustrated in Figure 6.13, such a dummy service contains a (dummy) client process that transmits (unprocessed) packets towards a (dummy) server processes. Hence, the protocol that this dummy service uses to communicate terminates on a remote node. ISS^{server}_{old} therefore cannot be discarded in case of service addition.

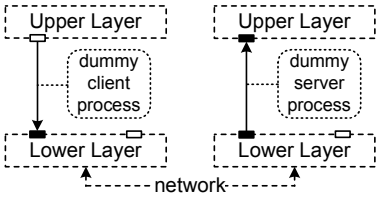


Figure 6.13: No reliability service deployed

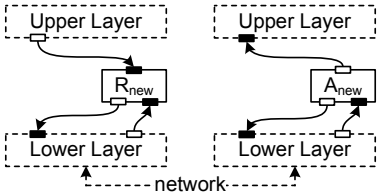


Figure 6.14: Nodes equipped with reliability service

Modifications

We illustrate both customizations with the addition and removal of a reliability service (in contrast to its replacement). With this in view, Figure 6.13 sketches the protocol stacks of two nodes when no reliability support is deployed. In addition, Figure 6.14 sketches both nodes equipped with a reliability service.

a) Service addition. When NeCoMan conducts a synchronized distributed reconfiguration to add a new service, it omits the execution of UO_{old}^{ext}, UO_{old}^{int} and DC_{old}. This is because no old service components must be removed.

Besides, NeCoMan also slightly customizes its finishing phase. To illustrate this, recall that adding a new network service is similar to replacing a dummy service with this new version. In this case quiescence comes about on the arrival of all "unprocessed" packets which are still in transit. Hence, reaching quiescence involves intercepting or redirecting packets on the nodes hosting the dummy client processes, as well as monitoring the affected dummy server processes. The execution of ISS_{old}^{client}, however, becomes redundant in this case.

To illustrate the effect of this customization, Table J.16 lists all reconfiguration conditions that are changed in case of service addition. Furthermore, we demonstrate in Appendix H in more detail how NeCoMan dynamically adds a reliability service on two programmable nodes. To be precise, Figures H.3 and H.5 sketch this reconfiguration when quiescence is reached before and after activating the new service, respectively. Besides, Figures H.4 and H.6 model the algorithms that NeCoMan uses to carry out both reconfiguration scenarios.

b) Service removal. When NeCoMan executes a service removal, it omits the execution of CC_{new}, LO_{new}^{ext}, LO_{new}^{int}, AP_{new}^{client}, and AP_{new}^{server} as there are no new components involved.

Besides, when customization "activate before finish" is applicable as well, NeCoMan performs two additional changes on its current reconfiguration algorithm. First, APD_{new}^{mark} cannot be used as is in this case. Because service removal involves no new service components, NeCoMan cannot attach (the inports of) marking components to the outports of these missing service components. Hence, NeCoMan

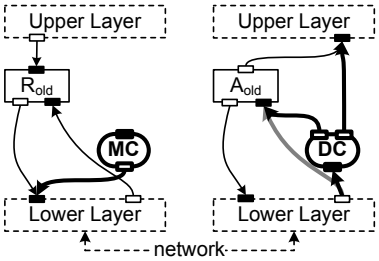


Figure 6.15: Adding marking and dispatching support when removing the old reliability service

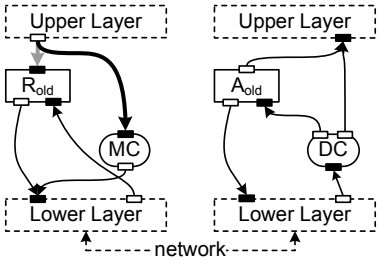


Figure 6.16: Activating new (dummy service) by redirecting packets towards the new marking component

instructs the affected nodes to (only) bind the outports of the marking components. This is illustrated in Figure 6.15. We denote this (slightly) different variant of APD^{mark}_{new} as $APD^{'mark}_{new}$.

Furthermore, NeCoMan also replaces $LI^{ext}_{old-new}$ by $LI^{'ext}_{old-new}$ in this case. $LI^{'ext}_{old-new}$ redirects packets from the old component's service-external inports towards the inports of the marking component – that is, instead of redirecting them towards the service-external inports of the new missing service component as $LI^{ext}_{old-new}$ would do. Figure 6.16 illustrates the effect of $LI^{'ext}_{old-new}$ when removing the reliability service.

To illustrate the effect of this customization, Table J.17 lists all reconfiguration conditions that are changed in case of service removal. Furthermore, we refer (again) to Appendix H for a more detailed overview of how NeCoMan dynamically removes the reliability service from two programmable nodes[8].

6.9.2 Independent distributed reconfigurations

Pre-conditions

NeCoMan only uses its isolated distributed reconfiguration algorithm to add or remove a service if this algorithm results from applying customization "no distributed activation"[9]. This implies that the network or the new network service must be able to handle structural inconsistencies. To dynamically add a compression and decompression component on two programmable nodes independently from each other, for instance, the new decompression component must be able to process compressed as well as uncompressed packets. Besides, when removing both components afterwards (again independently from each others), the network must be able to deal

[8] To be precise, see Figures H.7 to H.10

[9] When NeCoMan uses its independent distributed reconfiguration algorithm because the underlying nodes employ state transfer to finish the old service, in contrast, then only service replacement can be supported.

with packets reaching their destination in a compressed form.

Modifications

In case of service addition, NeCoMan omits the execution of DC_{old}, UO^{int}_{old} and UO^{ext}_{old} because no old service components must be removed. Besides, the execution of ISS^{client}_{old} and ISS^{server}_{old} will be discarded as well as the old (dummy) service does not have to be finished. Table J.18 lists all reconfiguration conditions that are affected by this customization.

When carrying out a service removal, NeCoMan omits the execution of CC_{new}, LO^{ext}_{new}, LO^{int}_{new}, AP^{client}_{new}, and AP^{server}_{new}. This is because the new (dummy) service does not contain new components. To illustrate the effect of this customization, Table J.19 lists all reconfiguration conditions that this customization changes.

6.10 Conclusion

We conclude this chapter by revisiting the four requirements that NeCoMan must satisfy to achieve its objectives.

6.10.1 Correct reconfigurations

Similar as for the customizations to NeCoMan's local reconfiguration algorithms, none of the customizations presented in this chapter compromise the reconfiguration correctness – that is, given that all associated pre-conditions are fulfilled. This is because the resulting algorithms meet all (adapted) reconfiguration conditions.

6.10.2 Limited reconfiguration overhead

All customizations presented in this chapter seek to optimize distributed reconfigurations. To illustrate this, Appendix K evaluates the effect on reconfiguration overhead for the first three customizations. All other customizations involve omitting redundant reconfiguration actions, and thus optimize the reconfiguration scenario as well.

Note, however, that the current version of NeCoMan cannot always select the *most optimal* algorithm for each distributed reconfiguration. For instance, when replacing a stateful service that takes a long time to finish, activating the new service before the old one reaches quiescence will reduce communication disruption. But, when the service reaches the finished state in a negligible period of time, NeCoMan's original synchronized reconfiguration algorithm might be a better choice. This solution needs less synchronization and lacks the potential performance overhead that the packet-distinguishing support causes. Similarly, customization "no finishing"

Question	Cust.
Are the old and new services compatible?	6.2
Are the affected service components stateless?	6.3, 6.4
Do the affected components share their execution state (if any) with their client applications?	6.3
Are the new service components able to process ongoing protocol-transactions?	6.4
Can the new service components recover from inconsistent execution states?	6.4
Do the old service components communicate by a protocol that terminates locally or remotely?	6.5
Do the old server processes encapsulate state that goes beyond the execution of a single protocol transaction?	6.5
Do the new components employ active objects or not?	6.6
Do the old and new components employ only client or server processes instead of both?	6.7
Do the old and/or new server processes communicate by a unidirectional or bidirectional communication protocol?	6.8

Table 6.3: Distributed reconfigurations: questions that the network administrator must answer to specify the service characteristics. The right column lists the related customizations.

only reduces the reconfiguration overhead when the effect of potential inconsistencies on the network performance is insignificant[10]. Hence, NeCoMan should also take into account this context specific information to decide whether or not to apply this customization.

6.10.3 Limited openness

To identify which customizations it can apply, NeCoMan checks for each customization if all associated pre-conditions are fulfilled. NeCoMan therefore requires from the network administrator to specify the service characteristics and reconfiguration semantics by answering the questions listed in Tables 6.3 and 6.4, respectively. Hence, NeCoMan restricts the contribution needed to conduct a correct and optimized reconfiguration.

6.10.4 Reusability

Most customizations presented in this chapter involve only re-ordering and omitting the (existing) reconfiguration actions that both basic algorithms incorporate. The

[10]This has already been explained in Section 4.8.2

Question	Cust.
Do the affected nodes impose a safe state by deactivating the old components immediately and transferring their execution state?	6.3
Does the network tolerate packet re-ordering?	6.3
Do the affected components operate in a best-effort network?	6.4
Does the network restore from or tolerate inconsistent execution states?	6.4
Can the network deal with incorrect service compositions?	6.2
Does the reconfiguration involves service addition, replacement or removal?	6.9

Table 6.4: Questions that the network administrator must answer to specify the reconfiguration semantics. The right column lists the related customizations.

customizations presented in Sections 6.3, 6.8, and 6.9, however, have also introduced new reconfiguration actions – APD_{new}^{mark}, $APD_{new}^{'mark}$, $APD_{old-new}^{disp}$, RPD_{new}^{mark}, $RPD_{old-new}^{disp}$, $LI_{old-new}^{'ext}$, LI_{new}^{int}, and UI_{old}^{int}, to be precise. These new reconfiguration actions invoke only the predefined set of operations that a node's reconfiguration support must provide. The customizations presented in this chapter, therefore, do not compromise NeCoMan's ability to be reused on top of other flow-oriented, component-based protocol stack architectures besides DiPS+.

> *We conclude that the basic distributed reconfiguration algorithms extended with the customizations presented in this chapter enable NeCoMan to achieve its objectives.*

Chapter 7

Design and reconfiguration overhead

In the previous chapters we presented NeCoMan's (basic) reconfiguration algorithms as well as the customizations to these algorithms that are included. This chapter changes focus slightly and discusses part of NeCoMan's design. Besides, we also evaluate the actual overhead that the NeCoMan middleware brings about.

7.1 Design

To promote reuse, NeCoMan decouples the logic to *define* customized reconfigurations from node-specific support to *execute* these reconfigurations. This separation of concerns enables to reuse the same reconfiguration logic for different programmable node architectures. This is an important advantage, as NeCoMan's reconfiguration logic is complex and error-prone to develop from scratch.

As illustrated in Figure 7.1, this separation of concerns has been achieved by splitting up NeCoMan's functionality into a script generator (which encapsulates the reconfiguration logic) and node-specific virtual machines (to execute these reconfiguration scripts). In short, the script generator (as its name suggests) creates tailored and optimized reconfiguration scripts. These scripts seek to reconfigure various node architectures (such as Click [75], Netkit [36] and DiPS+ [97, 99]), and so they do not include platform-specific expressions. The affected programmable nodes, therefore, must provide a node-specific virtual machine (VM) to execute these portable reconfiguration scripts. Hence, those VMs are responsible for carrying out the actual node reconfiguration.

An additional advantage of separating both concerns (besides promoting reusability) includes that the reconfiguration logic does not have to be executed on the network nodes themselves. A network administrator or adaptive network management

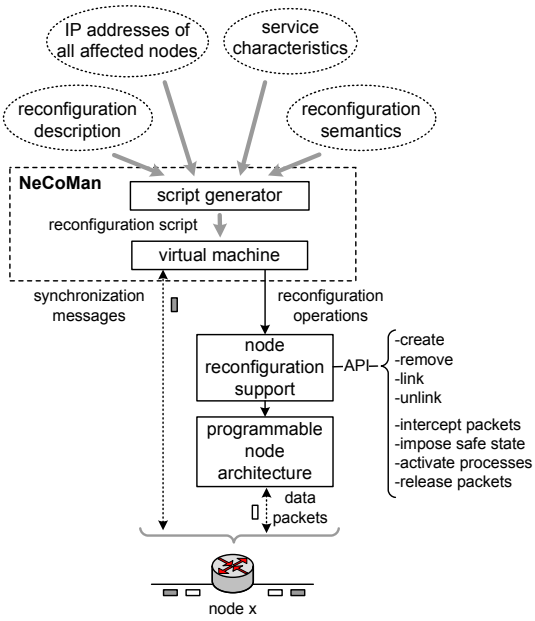


Figure 7.1: High-level overview of the NeCoMan architecture

software can generate reconfiguration scripts outside the network or on dedicated nodes, and upload them to the affected programmable nodes when reconfiguration is required. This way, composing a customized reconfiguration does not consume (the often limited) node resources.

7.1.1 NeCoMan reconfiguration script

So, to promote reusability, the script generator is responsible for creating tailored and portable reconfiguration scripts for all nodes participating in a reconfiguration. Listing 7.1 depicts such a script for replacing R_{old} with R_{new} on one specific node (that is, as part of replacing the complete reliability service). To be sufficiently portable, these reconfiguration scripts do not include platform-specific expressions. Instead, they specify (1) the *reconfiguration operations* that the affected node must execute and (2) the *synchronization operations* that are required to correctly synchronize distributed reconfigurations.

Reconfiguration operations. Recall that NeCoMan's reconfiguration actions do nothing more than coordinating the invocation of (some of) the eight reconfiguration operations that the underlying node must provide[1]. NeCoMan's script

[1] as explained in Section 3.5.2, page 67

generator, therefore, determines for each reconfiguration the operations that have to be invoked as well as their invocation sequence. To illustrate this, line 2 specifies the execution of CC_{new}, which includes invoking the node's reconfiguration support to execute the "create" operation. Next, lines 3 and 4 specify the execution of LO^{int}_{new}, which involves linking R_{new}'s data-outport to the inport of the lower layer.

Synchronization operations. Besides reconfiguration operations, a reconfiguration script can also include "synchronization operations". These synchronization operations specify the distributed coordination that is needed to correctly execute a reconfiguration. Line 9, for instance, expresses that message **A** must be sent from the node where this script is executed towards node `192.168.150.2`. In addition, line 21 specifies that message **B** sent from node `192.168.150.2` must have arrived before continuing the reconfiguration of the affected node.

```
1  # Installation phase
2  create(component_id="Rnew",class_name="NewRetrComp");
3  link(source_comp="Rnew",source_port="data-outport",
4      dest_comp="LowerLayer",dest_port="inport");

6  # Finishing phase
7  intercept_packets(component_id="Rold",process_id="retransmit");
8  impose_safe_state(component_id="Rold",process_id="retransmit");
9  sync_notify(message_string="A",dest_address="192.168.150.2");

11 # Activation phase
12 activate_processes(component_id="Rnew",process_id="retransmit");
13 unlink(source_comp="LowerLayer",source_port="outport",
14     dest_comp="Rold",dest_port="ack-inport");
15 link(source_comp="LowerLayer",source_port="outport",
16     dest_comp="Rnew",dest_port="ack-inport");
17 unlink(source_comp="UpperLayer",source_port="outport",
18     dest_comp="Rold",dest_port="data-inport");
19 link source_comp="UpperLayer",source_port="outport",
20     dest_comp="Rnew",dest_port="data-inport");
21 sync_wait(message_string="B",source_address="192.168.150.2");
22 release_packets(component_id="Rnew",process_id="retransmit");

24 # Removal phase
25 unlink(source_comp="Rold",source_port="data-outport",
26     dest_comp="LowerLayer",dest_port="inport");
27 remove(component_id="Rold");
```

Listing 7.1: Example of a reconfiguration script. This script specifies the replacement of R_{old} with R_{new} (as part of replacing the complete reliability service). Note that this reconfiguration involves finishing the old reliability service before activating the new one.

7.1.2 Creating customized reconfiguration scripts

As illustrated in Figure 7.1, NeCoMan's script generator creates these tailored reconfiguration scripts starting from (1) a declarative description of the recomposition that must be executed, (2) a specification of the service characteristics and the reconfiguration semantics, and (3) an overview of the IP-addresses of all nodes participating in this reconfiguration. The recomposition description must be expressed in terms of link, unlink, create, and remove primitives (as these are the primitives to define a composition). To illustrate this, Listing 7.2 sketches the description that was used to define the replacement of R_{old} with R_{new}. In addition, the specification of the service characteristics and reconfiguration semantics results form answering the questions listed in Tables 4.5, 4.6, 6.3, and 6.4[2]. To illustrate this, Table 7.1 lists the service characteristics and reconfiguration semantics that have resulted in the script that Listing 7.1 depicts.

```
create(component_id="Rnew",class_name="NewRetrComp");
unlink(source_comp="UpperLayer",source_port="outport",
    dest_comp="Rold",dest_port="data-inport");
unlink(source_comp="Rold",source_port="data-outport",
    dest_comp="LowerLayer",dest_port="inport");
unlink(source_comp="LowerLayer",source_port="outport",
    dest_comp="Rold",dest_port="ack-inport");
link source_comp="UpperLayer",source_port="outport",
    dest_comp="Rnew",dest_port="data-inport");
link(source_comp="Rnew",source_port="data-outport",
    dest_comp="LowerLayer",dest_port="inport");
link(source_comp="LowerLayer",source_port="outport",
    dest_comp="Rnew",dest_port="ack-inport");
remove(component_id="Rold");
```

Listing 7.2: A declarative reconfiguration description that specifies the replacement of R_{old} with R_{new} (as part of replacing the complete reliability service).

NeCoMan's script generator creates customized reconfiguration scripts in two steps. The script generator first composes a tailored reconfiguration algorithm starting from the specified service characteristics and reconfiguration semantics. This includes (1) selecting the proper basic reconfiguration algorithm to start from, and (2) identifying which customizations can be applied to this algorithm. Appendices F and L explain this customization procedure in more detail by presenting the order in which NeCoMan's script generator applies customizations to its local and distributed reconfiguration algorithms, respectively.

Once a tailored algorithm is composed, NeCoMan's script generator converts this algorithm into reconfiguration scripts for all nodes participating in the reconfiguration. For each reconfiguration action that the algorithm includes, the script generator determines the associated reconfiguration operations that must be invoked. This can be achieved by taking into account the declarative description of

[2] As explained in Sections 4.8.3 and 6.10.3

Service characteristics: question	Answer
Are the old and new services compatible?	No
Are the affected service components stateless?	No
Do the affected components share their execution state (if any) with their client applications?	No
Are the new service components able to process ongoing protocol-transactions?	No
Can the new service components recover from inconsistent execution states?	No
Do the old service components communicate by a protocol that terminates locally or remotely?	Yes
Do the old server processes encapsulate state that goes beyond the execution of a single protocol transaction?	No
Do the new components employ active objects or not?	Yes, their new client processes do
Do the old and new components employ only client or server processes instead of both?	Only client processes
Do the old and/or new server processes communicate by a unidirectional or bidirectional communication protocol?	Bidirectional protocol
Reconfiguration semantics: question	**Answer**
Do the affected nodes impose a safe state by deactivating the old components immediately and transferring their execution state?	No
Does the network tolerate packet re-ordering?	No
Do the affected components operate in a best-effort network?	No
Does the network restore from or tolerate inconsistent execution states?	No
Can the network deal with incorrect service compositions?	No
Does the reconfiguration involves service addition, replacement or removal?	Replacement

Table 7.1: Service characteristics and reconfiguration semantics that have resulted into the reconfiguration script depicted in Listing 7.1.

the recomposition that the network administrator must provide. The reconfiguration operations that the script in Listing 7.1 includes, for instance, are based on the reconfiguration description specified in Listing 7.2.

Besides the reconfiguration operations, the script generator also determines the synchronization operations that must be added to the reconfiguration scripts. To be precise, the script generator translates each synchronization point into a "sync-wait" instruction (see line 21 in Listing 7.1). When synchronization messages are to be sent, in contrast, the script generator adds "sync-notify" instructions to the re-

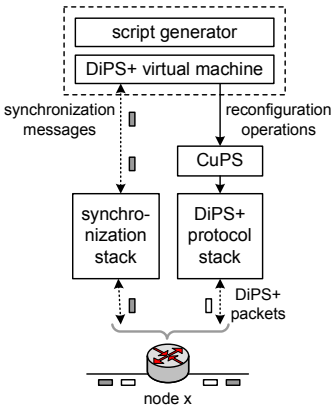


Figure 7.2: High-level overview of the NeCoMan architecture when employed to customize DiPS+ protocol stacks.

configuration scripts (see line 9 in Listing 7.1). Note that the network administrator must specify the associated IP-addresses[3].

7.1.3 Virtual machine to execute reconfiguration scripts

Each reconfigurable node must provide a node-specific virtual machine (VM) to interpret and execute these portable reconfiguration scripts. To illustrate that the functionality of this VM can be kept very lightweight, we (briefly) present the DiPS+ VM in this subsection. This is a proof-of-concept VM that has been developed to reconfigure DiPS+ networks. Note that we do not intend to give a full-featured overview of this VM, but instead briefly elaborate on how this VM deals with the reconfiguration and synchronization operations that a NeCoMan script specifies.

Reconfiguration operations. As discussed in Section 3.3, we developed the CuPS (Customizable Protocol Stacks) platform to assist NeCoMan in dynamically recomposing DiPS+ protocol stacks. The DiPS+ VM, therefore, does not execute the specified reconfiguration operations by itself, but instead invokes CuPS to accomplish this. This is illustrated in Figure 7.2.

Synchronization operations. Node reconfiguration support like CuPS, however, is not supposed to assist a reconfiguration middleware in synchronizing a distributed reconfiguration. The DiPS+ VM therefore deals with synchronization operations by itself. As losing these messages compromises the execution of a distributed reconfiguration, the current proof-of-concept implementation of the DiPS+

[3]as stated in the beginning of this subsection

VM uses a simple reliability protocol that acknowledges the correct arrival of each synchronization message. These synchronization and acknowledgement messages, however, cannot be transmitted by the protocol stack being recomposed, as this may cause deadlock situations. So, the DiPS+ VM uses a different stack to exchange synchronization messages (as illustrated in Figure 7.2).

Furthermore, the DiPS+ VM collects all incoming synchronization messages in a blackboard data-structure. When a "sync-wait" operation must be executed, the DiPS+ VM checks the blackboard for the arrival of the specified synchronization message. If this message has not yet been received, the DiPS+ VM delays the execution of the reconfiguration until the blackboard announces its arrival.

7.2 Reconfiguration overhead

Additionally, we (briefly) evaluate the reconfiguration overhead that the DiPS+ VM brings about[4]. To do so, we investigate the actual cost incurred by a number of reconfigurations. This cost will be evaluated in terms of

1. the *communication disruption* that these reconfigurations cause, and
2. the *reconfiguration time*.

The first metric quantifies the period of time in which the affected nodes will be unable to process packets. This period must be as small as possible, so as to minimize the impact of a dynamic reconfiguration on the network quality attributes (which include availability, response-time, and throughput [59]). The second metric the time that it takes to complete a reconfiguration.

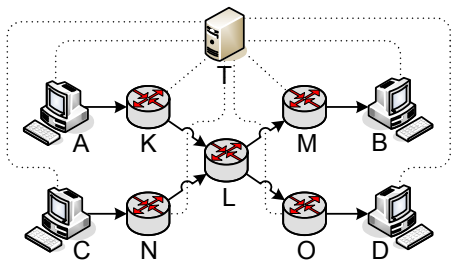
7.2.1 Test configuration

To evaluate the reconfiguration overhead of the DiPS+ VM, we measured the cost of adding, replacing, and removing a compression service and a reliability service on a DiPS+ network[5]. Note that these services have different characteristics, and thus represent different types of network services. The DiPS+ compression service, for instance, is a lightweight service that operates at the DiPS+ IP layer. The DiPS+ implementation of the reliability service, in contrast, is more cpu-intensive and operates at the transport layer of the OSI model. To be precise, this service has been developed to extend the DiPS+ UDP layer.

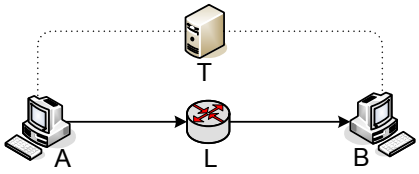
Figure 7.3(a) sketches the test setup that is used for measuring the cost of adding, replacing, and removing the compression service. In this configuration

[4]Appendices E and K provide an extra analysis of the effect on reconfiguration overhead that some of the customizations presented in Chapters 4 and 6 bring about.

[5]The DiPS+ network consists of a number of DiPS+ routers and end-nodes that each run on separate PCs. These PCs have 256 MB RAM, use Intel Pentium 2 400Mhz processors and run Linux 2.4.25. All software is written in Java, compiled using the Sun 1.4.2 JDK.



(a) Configuration to evaluate the cost of adding, replacing and removing the compression service



(b) Configuration to evaluate the cost of adding, replacing and removing the reliability service

Figure 7.3: Test setup to evaluate the overhead that DiPS+ VM causes.

nodes A and C represent source nodes transmitting test packets to nodes B and D. The compression service will be used on routers K, N, M, and O (which are all DiPS+ routers). Packets will be compressed on routers K and N, and are decompressed afterwards when passing through routers M and O. Finally, node T coordinates all test activities. To minimize the impact of the latter on regular network communication, node T communicates with the other nodes and routers in this setup through a separate network (symbolized in Figure 7.3(a) by dotted lines).

The test setup that is used for measuring the cost of adding, replacing, and removing the reliability service, however, is slightly different. This is because the reliability service (in contrast to the compression service) operates at the transport level – that is, between two end-nodes. The test configuration that is used to measure the overhead of these reconfigurations therefore includes only one source and one sink node communicating with each other over a single path. This setup is illustrated in Figure 7.3(b). Note that in this test configuration the retransmission and acknowledgement component will be added, replaced and removed from nodes A and B, respectively.

Finally, note that the addition, replacement and removal of both services has been executed both with and without applying the "activate before finishing" customization[6]. This allows us to compare the overhead of the (basic) algorithm presented in Section 5.4 (which includes reaching quiescence before activating the new service) with the overhead of the algorithm presented in Section 6.3 (which involves activating the new service before the old one is quiescent). In the next subsections we evaluate this overhead in terms of communication disruption and reconfiguration time[7].

[6]This customization is described in Section 6.3.
[7]as discussed in the beginning of this section

7.2.2 Communication disruption

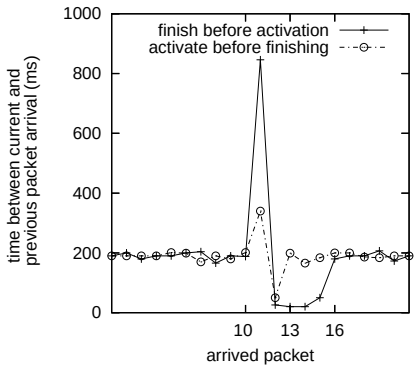
Reconfigurations that involve the compression service

To determine the communication disruption that the DiPS+ VM causes, we measured the effect of each reconfiguration on the packet arrival frequency. When the compression service is involved, senders A and C each send out a new test packet every 180 ms. Previous experiments have indicated that this is the highest sending frequency at which limited variation on the packet arrival frequency will be experienced when using compression on nodes K, M, N and O. By increasing this sending frequency, the variation on the time between successive packet arrivals increases as well, which in turn obfuscates the communication disruption that the reconfiguration causes.

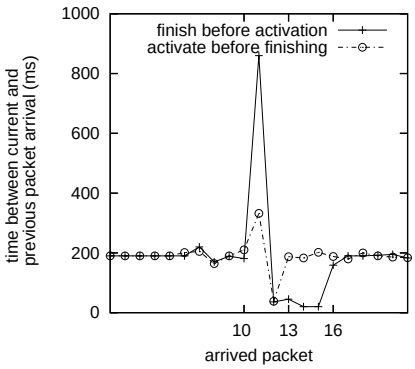
Figure 7.4 illustrates the resulting communication disruption measured at node B when adding, replacing or removing the compression service. We can clearly deduce from these graphs that (for this test setup) each reconfiguration causes less communication disruption if the new service becomes activated before finishing the old one. In that case, only 2 packets arrive at a moment in time different from what is expected (packets 11 and 12). When the old service becomes finished before activating the new one, however, 5 packets arrive at a moment in time different from what is expected (see packets 11 to 15). Furthermore, note that the "peak delay" is significantly larger when the old service becomes finished before activating the new one: packet 11 arrives approximately 680 ms later than expected. When activating the new service before finishing the old one, this delay is approximately 200 ms.

Besides, one can also observe from Figure 7.4 that packets 12 to 15 arrive very short after each other at their destination when the old service becomes finished before activating the new one. This is because the current implementation of CuPS releases intercepted packets almost immediately one after another. Besides, since (for these test reconfigurations) the time to process these intercepted packets is smaller than the frequency at which packets are transmitted from their source to their destination, new packets were not delayed by the processing of these intercepted packets. All this explains why the arrival frequency stabilizes rather abrupt (instead of gradually) for each of these reconfigurations.

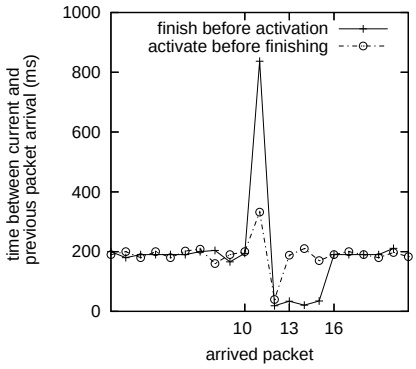
As a final remark to these test results, note that communication disruption is similar when adding, replacing or removing the compression service. The reason for this is twofold. First, we use the same technique to impose a safe state over the old compression service (in case of replacement or removal) as to finish the old "dummy" service (in case of addition). For each of these reconfigurations, the nodes hosting the server processes (that is, node M and O) wait for 500 ms to make sure that all packets have arrived, and thus quiescence is reached. Second, the time to initialize the functionality of the new compression service is very low. The first packet that uses this compression service thus will not suffer (significant) extra delay. All this explains why the three graphs in Figure 7.4 are similar.



(a) Adding compression service



(b) Replacing compression service



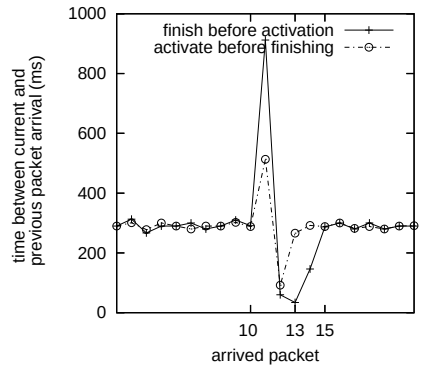
(c) Removing compression service

Figure 7.4: Communication disruption measured at node B when adding, replacing and removing the DiPS+ compression service. These graphs depict the variation on the interval between successive packet arrivals that was measured for each of these reconfigurations.

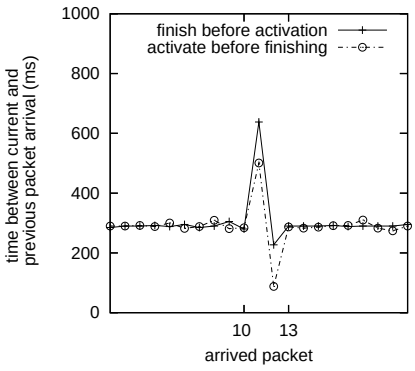
Reconfigurations that involve the reliability service

Besides the addition, replacement and removal of the compression service, we also measured the communication disruption that the DiPS+ VM causes when adding, replacing and removing the reliability service. To measure the effect of these reconfigurations on the frequency at which packets arrive at node B, sender A sends out a new test packet every 280 ms. Similar to the tests with the compression service, this frequency results from previous experiments to determine the highest sending frequency at which limited variation on the packet arrival frequency will be experienced when the reliability service is deployed.

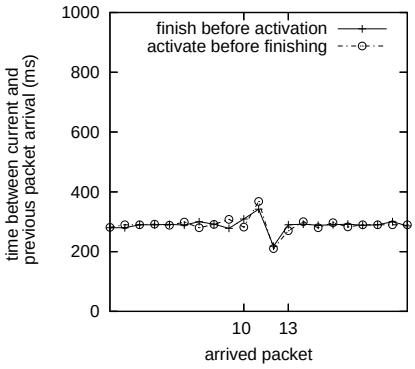
Figure 7.5(a) illustrates the communication disruption measured at node B when



(a) Adding reliability service



(b) Replacing reliability service



(c) Removing reliability service

Figure 7.5: Communication disruption measured at node B when adding, replacing and removing the DiPS+ reliability service.

adding the reliability service to nodes A and B[8]. Note that the peak delays that this reconfiguration causes are higher than when the compression service is involved. For every service addition, however, we used the same approach to reach quiescence (that is, waiting for 500 ms at the node hosting the server process). We therefore attribute this higher peak delay to the initialization cost of the (new) reliability components. The first packet that a new reliability component processes, initializes various reliability specific concerns (including session management, timer activation, etc.). Hence, this adds an extra delay to the processing of that packet, which

[8]Note that in contrast to the addition of the compression service, not 5 but 4 packets arrive at a moment in time different from what is expected (see packets 11 to 14). This difference results from the sending frequency, which is lower when the reliability service is involved than when adding the compression service.

explains the higher peak delays.

Figure 7.5(b) illustrates the communication disruption when replacing the reliability service with a new version. If the old service becomes finished before activating the new one, the replacement of this reliability service causes less communication disruption than its addition. This is because (in this test configuration) replacing the reliability service involves intercepting packets for approximately 340 ms, while in case of adding this service packets become intercepted for approximately 790 ms[9].

Finally, Figure 7.5(c) displays the communication disruption that the DiPS+ VM causes when removing the reliability service. In comparison to Figure 7.5(b), it catches the eye that the removal of the DiPS+ retransmission service causes less service disruption than its replacement. However, for both reconfigurations the same mechanism is used to detect quiescence (that is, monitoring R_{old} at node A until its retransmission queue is empty and no packets are currently being processed). We therefore attribute the extra disruption that a replacement causes again to the the initialization time of the (new) reliability components. As we already explained before, the first packet that these components process will initialize various reliability specific concerns, which in turn causes an extra delay to the processing of this packet.

Besides, note that the difference in service disruption for both graphs in Figure 7.5(c) is very small. Hence, (for this reconfiguration) the DiPS+ VM causes similar overhead whether or not it applies the "activate before finishing" customization. We (partially) attribute this to the inefficient implementation of the marking and dispatching components, which adds an extra delay to the packets that must be de-multiplexed during reconfiguration. Because this overhead was also generated when removing the compression service, one could question why Figure 7.4(c) does not depict a similar equality in service disruption for both reconfiguration scenarios – that is, "activate before finishing" and "finish before activation". This is because driving the compression service to quiescence caused (significantly) more service disruption than when the reliability service was involved. By reducing the time it takes for the compression service to reach quiescence, the benefit of applying the "activate before finishing" customization on the service disruption that this reconfiguration causes will decrease as well (similar as for removing the reliability service).

7.2.3 Reconfiguration time

Finally, Table 7.2 lists the time that it took for the DiPS+ VM to complete each reconfigurations discussed above. This overview illustrates a significant variation in reconfiguration time: the minimum time to complete a reconfiguration was 271 ms, while the maximum time was 1500 ms. It is clear that this variation results from (1)

[9]This results from the adopted approach to reach quiescence: the node hosting the server processes (that is, node B) waits for 500 ms to make sure that all packets have arrived.

reconfiguration	**reconfiguration time (ms)**	
	finish before activation	**activate before finishing**
add compression	979	1290
replace compression	850	1187
remove compression	779	1124
add reliability	1296	1500
replace reliability	818	1128
remove reliability	271	601

Table 7.2: The time that it takes for the DiPS+ VM to add, replace and remove the compression and reliability service.

the different amount of reconfiguration actions that the DiPS+ VM has to execute for each reconfiguration, and (2) the different amount of nodes participating in the reconfiguration, which affects the execution time of the "sync-wait" and "sync-notify" operations.

7.3 Conclusion

In this chapter we discussed the design of the NeCoMan middleware. To promote reuse, the functionality of this middleware has been split up into a script generator on the one hand (which generates tailored and portable reconfiguration scripts) and node-specific virtual machines on the other hand (to execute these reconfiguration scripts). In addition, we presented (part of) the DiPS+ VM to illustrate how a NeCoMan VM can deal with the reconfiguration and synchronization operations that a reconfiguration script includes. Finally, we evaluated the reconfiguration overhead that this DiPS+ VM causes.

Chapter 8

Related research

Chapter 2 has already (briefly) discussed related work in the fields of programmable networks, dynamic software reconfiguration, and dynamic change management. To complete this survey, we compare NeCoMan in this chapter in more detail with a number of research initiatives in the field of programmable networks that are closely related. We classify these research initiatives based on whether they support local or distributed reconfigurations.

8.1 Local reconfigurations

In [82], Lee and Chang present a framework to dynamically recompose component-based protocol stacks. The design of these protocol stacks is very similar to DiPS+ stacks. Besides, Lee and Chang's framework supports the addition, removal, and replacement of components to/from a running protocol stack. In short, the add and remove operations are similar to the *insmod* and *rmmod* utilities of the Linux module system, while the replace operation involves state transfer to preserve consistency. Performance measurements indicate that replacing a TCP component takes around 200 ms[1].

When comparing this architecture with NeCoMan/CuPS/DiPS+, we point out the following differences. First, Lee and Chang's framework embodies one single reconfiguration algorithm, which is comparable to NeCoMan's basic local reconfiguration algorithms. Second, this algorithm binds the new component's outports after interrupting the old component. As explained in Section 3.5.2, to limit the communication disruption that a reconfiguration causes, NeCoMan binds the new component's outports once this component is instantiated. Third, instead of intercepting packets, Lee and Chang's framework governs a "safe reconfiguration point" by coordinating the execution of all active threads. When the reconfiguration thread

[1]These tests are executed on a Celeron 1.13 GHz PC.

is modifying a composition, no other thread can invoke the affected components simultaneously. Finally, the employed reconfiguration support is tightly coupled to the protocol stack architecture that will be recomposed, thus limiting its reusability.

Another programmable network architecture that supports dynamic reconfiguration is the Click modular router. Click is a software architecture developed at MIT for building flexible and configurable routers [75]. Similar to DiPS+ protocol stacks, these routers are assembled from fine-grained components implementing simple router functions (like packet classification, queueing, scheduling, etc.). To build a router configuration, the user chooses a collection of these components and connects them into a pipeline.

Recomposing a click router, however, involves changing the complete router configuration (instead of adding, replacing or removing some of its components). To execute a reconfiguration, the user has to install a new reconfiguration file with a hot-swapping option. This new configuration will only be brought into use if it initializes correctly (otherwise Click continues using the current router configuration). If the new configuration is correct, Click atomically captures the old components' state information and reinstates this into the new composition. All enqueued packets, for instance, are moved into the new composition. So, in comparison to NeCoMan/CuPS/DiPS+, Click's hot-swapping support (1) replaces the complete router composition, (2) supports one single reconfiguration scenario, which involves state transfer, and (3) is tightly coupled to the Click architecture.

Also Netkit, developed at Lancaster University [35, 36], supports dynamic reconfiguration. Netkit provides network programming support that, among others, seeks to facilitate the management of node configuration and reconfiguration. To accomplish this, the Netkit architecture (1) applies the OpenCOM component model[2], (2) provides extensive support for structural and behavioral reflection, and (3) uses the concept of component frameworks (CFs) to maintain integrity during reconfiguration.

Although Netkit provides (generic and principled) support to implement dynamic reconfigurations, the latter seems to be the developer's responsibility. To illustrate this, consider the Netkit IP router presented in [36]. This router provides support to load and bind new components in an IXP1200 environment[3]. Netkit's extensive use of reflection also enables to equip the affected component-outports with locks if needed. Furthermore, extra functionality to reach a reconfiguration-safe state can be added to the router CF, which manages the (re-)composition of routing functionality[4]. Composing all these features to a correct and efficient reconfiguration process, however, appears to be the task of the developer. Since this

[2]In short, OpenCOM is a lightweight and reflective component model developed at Lancaster University that uses the core features of Microsoft COM to underpin its implementation [37]

[3]For more information about the Intel IXP1200 network processor, see [3].

[4]A (Netkit) component framework embodies "rules and interfaces" that make sense for a specific domain. The router CF presented in [37], for instance, includes knowledge about the composition and reconfiguration of 'plugged-in' packet-forwarding components.

can be complex and error-prone, extending Netkit with NeCoMan as an additional meta-layer may provide a more controlled way to conduct Netkit reconfigurations.

In [43], Feng et al. describe an architecture to conduct dynamic reconfiguration of network management software (such as SNMP). In comparison to NeCoMan/CuPS/DiPS+, the embodied hot-swapping functionality supports one single reconfiguration scenario that involves service replacement only. This is because Feng's architecture carries out a reconfiguration by replacing a component's content instead of recomposing the affected node. Furthermore, Feng et al. present some (initial) ideas on how to coordinate distributed reconfigurations, which involves the use of multicast synchronization messages and synchronized system clocks. These ideas, however, are not further explored in [43].

Other out-of-band active network implementations like VERA [71], the Bowman NodeOS [95], the PromethOS NP framework [117] and Washington University's pluggable router framework [38] apply a different approach to dynamically reconfigure a programmable node. These architectures do not support router recomposition, but instead enable to dynamically *extend* router functionality with new forwarding services that apply to specific packet flows. To support this extensibility, the architectures listed above all provide a "programmable classifier", which delegates incoming packets to the appropriate service. Adding a new service to these routers thus involves (1) loading the associated functionality and (2) extending the programmable classifier with an extra filter which identifies the packet flows that should be redirected to this new service.

Note that these reconfigurations are similar to recompositions that involve activation before finishing. After loading new service components, they become activated immediately (that is, without waiting until other services are finished) by modifying the programmable classifier. These new service components thus operate in parallel with other router forwarding services until the latter become removed.

Finally, recall that active network implementations such as ANTS [141], Dan [39], NetScript [144] and SwitchWare [9] (to only name a few) support in-band customizations at packet transport granularity. Similar to the out-of-band active networks discussed above, these in-band active networks do not support the recomposition of running nodes but instead allow to dynamically plug-in new services that active packets can invoke when passing by. So, the only pre-condition that must be fulfilled for these active packets to use a new service includes that the service is present on the affected node. The reconfiguration of these in-band active network architectures, therefore, requires less coordination than when recomposing out-of-band active networks.

8.2 Distributed reconfigurations

Chen et al. present in [30] a system to coordinate the dynamic adaptation of distributed services and communication protocols. Although this architecture and

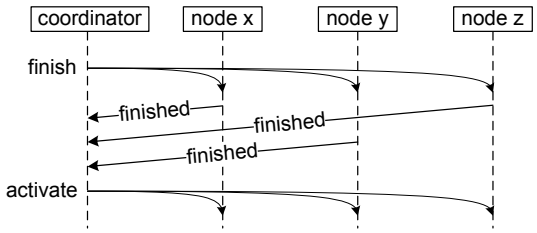


Figure 8.1: The protocol switch protocol in Ensemble

the NeCoMan middleware both coordinate distributed service reconfigurations, we point out three differences. First, Chen's adaptation algorithm always activates the new service before finishing the old one. When guaranteed packet ordering is required, the switch-over of incoming packets (arriving from the network) from the old to the new service components is deferred until the old components have processed all incoming packets. In addition, their adaptation algorithm only supports service replacement. Second, unlike NeCoMan, Chen's adaptation system is an open framework that requires the service components' developer to implement part of the adaptation algorithm. Examples include the activation of a new service component as well as some optimizations to the adaptation algorithm. Finally, Chen's adaptation system was not designed to be reused on top of various node architectures. The system is an extension of Cactus, a framework for constructing highly configurable middleware and protocol stacks [143]. The adaptation system is therefore tightly coupled to the Cactus syntax as well as to its architecture.

Robbert van Renesse et al. presented Ensemble, a hierarchical framework for constructing adaptive protocol stacks [133]. A distributed adaptation in Ensemble involves replacing every affected protocol stack – that is, instead of replacing the affected network-service components. These adaptations are coordinated by the Protocol Switch Protocol (PSP), which is a fault-tolerant protocol that synchronizes the participating stacks, assists them in finalizing their state, and performs the necessary agreement before resuming communication. Similar to the NeCoMan middleware, Ensemble seeks to facilitate the programming complexity of dynamic distributed adaptations.

The PSP's adaptation algorithm (depicted in Figure 8.1) is comparable to the basic distributed reconfiguration algorithm that NeCoMan includes, except that it conducts the replacement of entire protocol stacks rather than recomposing operating protocol stacks. Consequently, coordinating the sequence of activating the new protocol stacks is unnecessary, since a stack can be invoked only when it is fully operational. Besides, PSP elects one of the nodes that will be reconfigured to coordinate the distributed reconfiguration. As illustrated in Figure 8.1, this coordinator is responsible for managing a correct reconfiguration by collecting and broadcasting synchronization messages from and to other nodes participating in this reconfigura-

tion. In contrast to the NeCoMan middleware, the network load that such a single coordination unit causes scales better when many nodes are involved. At the same time, however, using a centralized coordination entity introduces a single point of failure.

Furthermore, note that also in-band active networks do not require external coordination to correctly activate a new service. To illustrate this, consider an active packet that contains a pointer to a specific router service. On every node that this packet visits, it activates the associated service. The new service thus becomes activated in the correct order without the need for external coordination. The only distributed coordination needed in this case involves the installation of service libraries on all affected nodes before the active packets arrive. Besides, note that active network architectures like ANTS [141] support automatic service installation on an as-needed basis. If the service libraries that an ANTS-packet requires are not present at a node, they are loaded dynamically using a lightweight distribution protocol. This on-demand scheme thus makes the need to install service libraries on a node before an active packet arrives redundant. This comes at the cost, however, of increased startup latency.

Finally, we briefly mention protocol boosters. In short, protocol boosters represent a design methodology for programming networks [42, 88]. This methodology imposes a number of design restrictions such that a booster (encapsulating protocol functions) can operate correctly (albeit with lower performance) in the absence of any booster-aware code [88]. These restrictions include that a booster can add, delete, or delay messages of an existing protocol, but may not originate or terminate that protocol. Besides, a booster element may not change the syntax of the packets that it processes. These booster elements therefore do not jeopardize the correct functioning of a network when inconsistent execution states or structural inconsistencies come about[5]. Hence, this methodology makes the need to coordinate distributed reconfigurations redundant[6].

[5] in contrast, for instance, to a compression service

[6] In this case, NeCoMan applies both the "no coordinated activation" and the "no finishing" customization to its first basic distributed reconfiguration algorithm

Chapter 9

Conclusions and future research

In this final chapter, we first recapitulate the main contribution of this work in Section 9.1. Next, we present in Section 9.2 a critical reflection and some open issues that are subject for future research. Finally, Section 9.3 presents future research tracks that involve investigating the contribution of customizable change management support like NeCoMan to research areas different from programmable networking.

9.1 Contributions

As a summary, we list the main contributions of this work in the research areas discussed in Chapter 2: programmable networking, dynamic software reconfiguration and dynamic change management.

Programmable networking

In the context of programmable networking, we have presented a middleware to dynamically reconfigure out-of-band active networks, which has been published in [65, 66, 70]. This middleware coordinates local and distributed node recompositions. Besides, this middleware fulfills the following requirements to meet its objectives:

- **Correct reconfigurations.** The algorithms that NeCoMan encapsulates do not compromise the correct network functioning in the course of a reconfiguration (except when the network tolerates inconsistencies to occur). In Chapters 3 and 5, we indicated that NeCoMan's basic reconfiguration algorithms conform to all reconfiguration conditions, and thus conduct correct

reconfigurations. Besides, the customizations to these basic reconfiguration algorithms presented in Chapters 4 and 6 do not compromise the correct network functioning either, as long as all associated pre-conditions are fulfilled.

- **Limited reconfiguration overhead.** Because NeCoMan's basic reconfiguration algorithms do not guarantee limited reconfiguration overhead for every reconfiguration, NeCoMan includes an extensive set of customizations to these algorithms. These customizations optimize and tailor the employed reconfiguration algorithm by switching the order in which some reconfiguration actions are executed, and by discarding those actions that are redundant for a particular reconfiguration. Besides, part of the reconfiguration overhead that NeCoMan brings about is caused by the VM as well as by the node architecture carrying out the reconfiguration operations that the VM invokes. Chapter 7 evaluated this overhead for the DiPS+ VM, which is a proof-of-concept VM that has been developed to reconfigure DiPS+ networks.

- **Limited openness.** To carry out a customized reconfiguration, NeCoMan only requires the network administrator to provide (1) a declarative description of the reconfiguration that must be executed, (2) a declarative description of the service characteristics and reconfiguration semantics. NeCoMan then creates a customized reconfiguration algorithm based on these properties. Consequently, the "openness" of this reconfiguration middleware to conduct a correct and optimized reconfiguration is restricted.

- **Reusability.** To promote reuse, NeCoMan separates the logic to define a customized reconfiguration (encapsulated by the script generator) from node-specific support to execute this reconfiguration (encapsulated by the virtual machine). As discussed in Chapter 7, separating these concerns enables to reuse NeCoMan's reconfiguration logic for other out-of-band active node architectures as well – that is, besides DiPS+/CuPS. This is an important advantage, as NeCoMan's reconfiguration logic is complex and error-prone to develop from scratch.

We validated this middleware by carrying out various reconfigurations. These include the runtime addition, replacement and removal of a compression service, a fragmentation service, a reliability service and the (DiPS+ implementation of) the TCP-booster developed at the University of Ghent [60].

Dynamic software reconfiguration

In the context of dynamic software reconfiguration, this dissertation has presented an extensive analysis on how to coordinate both local and distributed out-of-band compositional adaptations. Part of this analysis included defining a set of reconfiguration conditions that must be fulfilled to conduct correct and efficient reconfigurations. Because these reconfiguration conditions make explicit in what order

reconfiguration actions must be executed, they (1) provide some guidance for the development of future reconfiguration support, and (2) allow to reason about the reconfiguration process.

Dynamic change management

Finally, in the context of dynamic change management, this dissertation has presented *customizable* change management support. As we explained in Chapter 2, existing change management systems typically conform to the black-box philosophy by encapsulating a single and fixed reconfiguration algorithm. Because these systems do not allow developers to adapt the employed reconfiguration algorithm, Hillman and Warren propose to open up change management support such that the reconfiguration process can be customized to perform better in a specific context [59]. We argue, however, that opening up dynamic change management support increases again the costs and risks that dynamic software reconfiguration introduces.

NeCoMan, therefore, has been developed as customizable change management support. In contrast to black-box change management systems, NeCoMan allows a network administrator to customize the reconfiguration process by specifying the service characteristics and reconfiguration semantics. In contrast to open change management support, NeCoMan still protects the network administrator from the complexity of composing a correct and efficient reconfiguration algorithm. We believe that this provides a good balance between controlling the costs and risks of dynamic reconfiguration on the one hand, and supporting sufficient flexibility on the other hand.

9.2 Critical reflection and open issues

The work presented in this dissertation still has room for enhancement and further research.

Optimizing customization process

NeCoMan's current version cannot always select the *most optimal* algorithm for each reconfiguration. As discussed in Chapter 6, when replacing a stateful distributed service that takes a long time to finish, for instance, activating the new service before the old one reaches quiescence reduces communication disruption. But, when the service reaches the finished state in a negligible period of time, the first basic distributed reconfiguration algorithm might be a better choice. This solution needs less synchronization and lacks the potential performance overhead that the packet-distinguishing support causes. Similarly, omitting all finishing actions only reduces the reconfiguration overhead when the effect of potential inconsistencies on the network performance is insignificant. Future research, therefore, includes investigating what extra context specific information NeCoMan must take into account

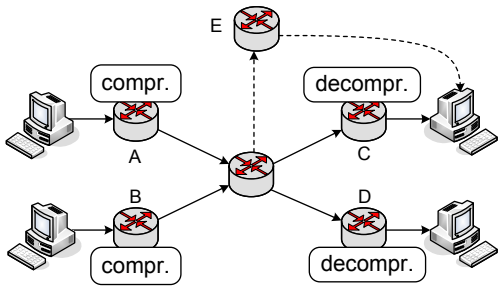


Figure 9.1: Compressed packets may leave the sub-net via node E without being restored in their original form.

to optimize its customization process.

Failure recovery support

The current version of NeCoMan assumes that the execution of a reconfiguration action never fails. When this occurs anyway, NeCoMan stalls its execution, thus leaving the network in an inconsistent state. To support atomic reconfigurations in this case, NeCoMan must be extended with (local and distributed) failure recovery support. Future research, therefore, includes investigating the impact of such failure recovery support on the current reconfiguration algorithms.

Additional validation

Although NeCoMan is prepared to be reused on top of various component-based node architectures, extra validation is needed to confirm its portability. Future research, therefore, includes extending node architectures like Netkit [35, 36] and Click [75] with a NeCoMan VM. This allows to validate NeCoMan's reconfiguration algorithms in different out-of-band active network setups besides DiPS+/CuPS networks.

Network-wide consistency preservation in case of dynamic network topologies

NeCoMan (only) preserves consistency in the course of a reconfiguration. When the topology of the network being reconfigured changes frequently, however, also after completing the reconfiguration a correct network service composition must be preserved. To illustrate this, consider the network setup depicted in Figure 9.1. When node E connects to this sub-net after NeCoMan has added a compression service to nodes A, B, C and D, then the network's structural integrity gets broken. This is because compressed packets may leave this sub-net without being restored in their original form. Future research, therefore, involves exploring how to preserve

network-wide consistency in a network characterized by a (highly) dynamic topology. Possible solutions to be investigated include (1) changing the routing scheme, and (2) managing topological changes such that new nodes cannot connect to a network without being equipped with the required network services.

9.3 Additional future research tracks

Other future research tracks involve investigating the contribution of customizable change management support like NeCoMan to research areas different from programmable networking. These include (1) reflective middleware, (2) middleware for transparent application reconfiguration, and (3) dynamic distributed aspect weaving.

Reflective middleware

Because the traditional role of middleware is to hide resource distribution and platform heterogeneity from the application logic, it is a logical place to define dynamic reconfiguration support related to various concerns like quality-of-service (QoS), energy management, fault tolerance, and security [91]. The dynamicTAO ORB [77, 78] and OpenORB [20, 77] (developed at the University of Illinois and Lancaster University, respectively) achieve dynamic reconfigurability of these concerns with the extensive use of reflection. This way, both middleware architectures are opened up to support developers in implementing dynamic out-of-band reconfigurations.

It is our believe, however, that reconfiguration-specific openness should be constrained as the implementation of correct and efficient reconfiguration scenarios can be complex and error-prone (hence compromising the benefit of dynamic reconfiguration). Extending middleware like dynamicTAO and OpenORB with NeCoMan-like support (targeted at ORB reconfigurations instead of recompositions of programmable nodes) as an additional meta-layer may conceal this reconfiguration complexity. Future research, therefore, includes investigating the benefits that can be gained by extending reflective middleware with customizable change management support.

Middleware for transparent application reconfiguration

Other middleware initiatives are targeted at supporting the reconfiguration of a running system with maximum transparency for the application developers. The work of Bidan et al. [19], for instance, seeks to accomplish this by extending CORBA with a dynamic reconfiguration service. This service supports the recomposition of CORBA applications by coordinating the addition, removal, and replacement of single application components. In addition, Almeida et al. presented a dynamic reconfiguration service for CORBA applications that (in contrast to Bidan's work) conducts the atomic replacement of multiple application components.

Both reconfiguration services encapsulate only a single reconfiguration algorithm, and therefore lack the opportunity to optimize recompositions if possible. Future research, therefore, includes investigating the benefit of using customizable change management support (like NeCoMan) to transparently reconfigure running applications.

Dynamic distributed aspect weaving

A final research track that spin-offs from the work presented in this dissertation relates to dynamic distributed aspect weaving. As Truyen and Joosen define in [128, 129], the latter involves the ability to dynamically weave and unweave distributed aspects on different computer nodes. These distributed aspects encapsulate services that are tightly coupled, including compression/decompression and encryption/decryption.

To preserve system consistency during dynamic aspect weaving (and unweaving), Truyen and Joosen present a model and an architecture for middleware, called Lasagne, that supports run-time weaving of distributed aspects in an atomic way [128]. This model supports in-band reconfigurations. Once an invocation becomes tagged with a so called "aspect identifier", the latter propagates with the message flow of the entire collaboration. On the nodes where needed, Lasagne activates the aspect that the identifier specifies. This way, Lasagne preserves system-wide execution consistency when weaving/unweaving distributed aspects.

A drawback of such in-band dynamic aspect weaving, however, is that it increases the processing time for each invocation. Future research, therefore, is targeted at investigating the benefits of out-of-band aspect weaving (including to what extent out-of-band aspect weaving may perform better). For that reason, we plan to validate NeCoMan's reconfiguration support in the field of dynamic distributed aspect weaving as well.

Bibliography

[1] http://www.mobileinfo.com.

[2] http://www.rfc-editor.org.

[3] http://www.intel.com/design/network/products/npfamily/ixp1200.htm.

[4] Intelligent Networking - A Foundation for Application and Service Optimization. White Paper Cisco Systems, March 2004.

[5] Stateful Inspection Technology. White Paper Check Point Software Technologies, August 2004.

[6] Check Point Application Intelligence. White Paper Check Point Software Technologies, May 2005.

[7] Sameer Ajmani. *Automatic Software Upgrades for Distributed Systems*. PhD thesis, MIT, September 2004.

[8] Mehmet Aksit and Zièd Choukair. Dynamic, adaptive and reconfigurable systems overview and prospective vision. In *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW '03)*, pages 84–89, Washington, DC, USA, 2003. IEEE Computer Society.

[9] D. Scott Alexander, William A. Arbaugh, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. The Switchware Active Network Architecture. *IEEE Network*, 12(3):29–36, May/June 1998.

[10] D. Scott Alexander, Bob Braden, Carl A. Gunter, Alden W. Jackson, Angelos D. Keromytis, Gary J. Minden, and David Wetherall. Active Network Encapsulation Protocol (ANEP). http://www.cis.upenn.edu/ dsl/switchware/ANEP/docs/ANEP.txt, July 1997.

[11] João Paulo A. Almeida, Maarten Wegdam, Marten van Sinderen, and Lambert J. M. Nieuwenhuis. Transparent Dynamic Reconfiguration for CORBA. In *Proceedings of the 3rd International Symposium on Distributed Objects and*

Applications (DOA 2001), pages 197–207, Rome, Italy, September 2001. IEEE Computer Society.

[12] Elan Amir, Steven McCanne, and Randy Katz. An active service framework and its application to real-time multimedia transcoding. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 178–189, Vancouver, British Columbia, Canada, 1998. ACM Press.

[13] Gregory R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):49–90, 1991.

[14] Oguz Angin, Andrew T. Campbell, Michael E. Kounavis, and Raymond R.-F. Liao. The mobiware toolkit: Programmable support for adaptive mobile networking. *IEEE Personal Communications*, 5(4):32–43, August 1998.

[15] Jason Baker and Wilson Hsieh. Runtime aspect weaving through metaprogramming. In *Proceedings of the 1st international conference on Aspect-oriented software development (AOSD '02)*, pages 86–95, New York, NY, USA, 2002. ACM Press.

[16] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, and Randy H. Katz. A comparison of mechanisms for improving tcp performance over wireless links. *IEEE/ACM Transactions on Networking*, 5(6):756–769, 1997.

[17] David Ball. A Smarter Way to Network: How an Intelligent, Systems-Based Approach Reduces Complexity While Increasing Functionality. White Paper Cisco Systems, October 2004.

[18] Samrat Bhattacharjee, Kenneth L. Calvert, and Ellen W. Zegura. An architecture for active networking. In *Proceedings of the IFIP TC6 seventh international conference on High performance netwoking VII (HPN '97)*, pages 265–279, London, UK, UK, 1997. Chapman & Hall, Ltd.

[19] Christophe Bidan, Valérie Issarny, Titos Saridakis, and Apostolos Zarras. A Dynamic Reconfiguration Service for CORBA. In *Proceedings of the International Conference on Configurable Distributed Systems (CDS 1998)*, pages 35–42, Washington, DC, USA, 1998. IEEE Computer Society.

[20] Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fabio Costa, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas, and Katia Saikoski. The Design and Implementation of Open ORB 2. *IEEE Distributed Systems Online*, 2(6), 2001.

[21] Toby Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, Massachusetts Institute of Technology, 1983. Also as MIT LCS Tech. Report 303.

[22] Toby Bloom and Mark Day. Reconfiguration in Argus. In *Proceedings of the International Workshop on Configurable Distributed Systems*, pages 176–187, London, England, March 1992.

[23] Matthias Bossardt, Takashi Egawa, Hideki Otsuki, and Bernhard Plattner. Integrated service deployment for active networks. In *Proceedings of the IFIP-TC6 4th International Working Conference on Active Networks (IWAN '02)*, pages 74–86, London, UK, 2002. Springer-Verlag.

[24] Bob Braden, Lixia Zhang, Steve Berson, Shai Herzog, and Sugih Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification, September 1997. RFC 2205.

[25] Kenneth L. Calvert, Samrat Bhattacharjee, Ellen Zegura, and James Sterbenz. Directions in Active Networks. *IEEE Communications Magazine, Special Issue on Programmable Networks*, 36(10):72–78, October 1998.

[26] Andrew Campbell, Stephen Chou, Michael Kounavis, Vassilis Stachtos, and John Vicente. NetBind: A Binding Tool for Constructing Data Paths in Network Processor-based Routers. In *Proceedings of the 5th conference on Open Architectures and Network Programming (OpenArch 2002)*, pages 49–60, New York City, NY, June 2002.

[27] Andrew T. Campbell, Michael E. Kounavis, Daniel A. Villela, John B. Vicente, Hermaan G. De Meer, Kazuho Miki, and Kalai S. Kalaichelvan. Spawning Networks. *IEEE Network*, 13(4):16–29, July/August 1999.

[28] Andrew T. Campbell, Herman G. De Meer, Michael E. Kounavis, Kazuho Miki, John B. Vicente, and Daniel Villela. A Survey of Programmable Networks. *ACM SIGCOMM Computer Communications Review*, 29(2):7–23, April 1999.

[29] Prashant Chandra, Yang-Hua Chu, Allan Fisher, Jun Gao, Corey Kosak, T.S. Eugene Ng, Peter Steenkiste, Eduardo Takahashi, and Hui Zhang. Darwin: Customizable Resource Management for Value-Added Network services. *IEEE Network*, 15(1):22–35, 2001.

[30] Wen-Ke Chen, M.A. Hiltunen, and R.D. Schlichting. Constructing Adaptive Software in Distributed Systems. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS'01)*, pages 635–643. IEEE Computer Society, 2001.

[31] Xuejun Chen. Extending RMI to Support Dynamic Reconfiguration of Distributed Systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, Vienna, Austria, 2002. IEEE Computer Society.

[32] Jonathan E. Cook and Jeffery A. Dage. Highly Reliable Upgrading of Components. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, pages 203–212, Los Angeles, California, United States, 1999. IEEE Computer Society Press.

[33] Pascal Costanza. Dynamic Object Replacement and Implementation-Only Classes. In *6th International Workshop on Component-Oriented Programming (WCOP 2001), ECOOP related Workshop*, Budapest, Hungary, June 2001.

[34] Pascal Costanza. Dynamic Replacement of Active Objects in the Gilgul Programming Language. In *Proceedings of the IFIP/ACM Working Conference on Component Deployment (CD '02)*, pages 125–140, London, UK, 2002. Springer-Verlag.

[35] Geoff Coulson, Gordon Blair, Antônio Tadeu Gomes, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Irvin Ye. A Reflective Middleware-based Approach to Programmable Networking. In *Proceedings of the 2nd workshop on Adaptive and Reflective Middleware*. ACM Press, 2003.

[36] Geoff Coulson, Gordon Blair, David Hutchison, Ackbar Joolia, Kevin Lee, Jo Ueyama, Antônio Gomes, and Yimin Ye. Netkit: a software component-based approach to programmable networking. *ACM SIGCOMM Computer Communication Review*, 33(5):55–66, 2003.

[37] Geoff Coulson, Gordon S. Blair, Michael Clarke, and Nikos Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126, 2002.

[38] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner. Router plugins: a software architecture for next generation routers. *ACM SIGCOMM Computer Communication Review*, 28(4):229–240, 1998.

[39] Dan Decasper and Bernhard Plattner. DAN: Distributed Code Caching for Active Networks. In *Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 1998)*, volume 2, pages 609–616, San Francisco, CA, USA, April 1998.

[40] Brian Ensink and Vikram Adve. Coordinating Adaptations in Distributed Systems. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 446–455. IEEE Computer Society, 2004.

[41] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of the 2nd international conference on Software engineering (ICSE '76)*, pages 470–476, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[42] David C. Feldmeier, Anthony J. McAuley, Jonathan M. Smith, Deborah S. Bakin, William S. Marcus, and Thomas M. Raleigh. Protocol boosters. *IEEE Journal on Selected Areas in Communications – Special Issue on Protocol Architectures for 21st Century Applications*, 16(3):437–444, April 1998.

[43] N. Feng, A. Gang, T. White, and B Pagurek. Dynamic Evolution of Network Management Software by Software Hot-Swapping. In *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management (IM '01)*, pages 63–76, Seattle, May 2001.

[44] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000. Chair-Richard N. Taylor.

[45] Enrica Filippi. Introduction to volume 2, issue 1. *ST Journal of Research – Networked Multimedia*, 2(1), November 2005.

[46] Richard Fox. TCP big window and NAK options, June 1989. RFC 1106.

[47] Ophir Frieder and Mark E. Segal. On dynamically updating a computer program: from concept to prototype. *Journal of Systems and Software*, 14(2):111–128, 1991.

[48] Xiadong Fu, Weisong Shi, and Vijay Karamcheti. Automatic deployment of transcoding components for ubiquitous. Technical Report TR2001-814, Computer Science Department, New York University, NY, USA, March 2001.

[49] Xiaodong Fu, Weisong Shi, Anatoly Akkerman, and Vijay Karamcheti. CANS: Composable, Adaptive Network Services Infrastructure. In *3rd USENIX Symposium on Internet Technologies and Systems (USITS 2001)*, pages 135–146, San Francisco, California, USA, March 2001.

[50] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[51] Claude Girault and Rüdiger Valk. *Petri Nets for Systems Engineering - A Guide to Modeling, Verification, and Applications*. Springer, Berlin, 2003.

[52] Steven Glassman. A caching relay for the World Wide Web. *Computer Networks and ISDN Systems*, 27(2):165–173, 1994.

[53] Bob Gleichauf. Core Elements of the Cisco Self-Defending Network Strategy. White Paper Cisco Systems, May 2005.

[54] Deepak Gupta and Pankaj Jalote. On-line software version change using state transfer between processes. *Software - Practice and Experience*, 23(9):949–964, 1993.

[55] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks. *ACM SIGPLAN Notices*, 34(1):86–93, 1999.

[56] Michael Hicks, Jonathan T. Moore, D. Scott Alexander, Carl A. Gunter, and Scott M. Nettles. PLANet: An active internetwork. In *Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 1999)*, volume 3, pages 1124–1133, New York, NY, USA, March 1999.

[57] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming Language Design and Implementation (PLDI '01)*, pages 13–23, New York, NY, USA, June 2001. ACM Press.

[58] Michael Hicks and Scott Nettles. Active networking means evolution (or enhanced extensibility required). In Hiroshi Yashuda, editor, *Proceedings of the Second International Working Conference on Active Networks (IWAN 2000)*, volume 1942 of *Lecture Notes in Computer Science*, pages 16–32. Springer-Verlag, October 2000.

[59] Jamie Hillman and Ian Warren. An open framework for dynamic reconfiguration. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 594–603, Washington, DC, USA, 2004. IEEE Computer Society.

[60] Jeroen Hoebeke, Tom Van Leeuwen, Liesbeth Peters, Koen Cooreman, Ingrid Moerman, Bart Dhoedt, and Piet Demeester. Development of a TCP protocol booster over a wireless link. In *Proceedings of the 9th Symposium on Communications and Vehicular Technology in the Benelux (SCVT 2002)*, Louvain la Neuve, oct 2002.

[61] Christine R. Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. PhD thesis, Computer Science Department, University of Maryland, College Park, 1993.

[62] Christine R. Hofmeister and James M. Purtilo. A Framework for Dynamic Reconfiguration of Distributed Programs. Technical Report CS-TR-3119, Computer Science Department, University of Maryland, College Park, 1993.

[63] Syed Asad Hussain. An active scheduling paradigm for open adaptive network environments. *International Journal of Communication Systems*, 17(5):491–506, 2004.

[64] Van Jacobson, Bob Braden, and Dave Borman. TCP Extensions for High Performance, May 1992. RFC 1323.

[65] Nico Janssens, Lieven Desmet, Sam Michiels, and Pierre Verbaeten. NeCoMan: Middleware for Safe Distributed Service Deployment in Programmable Networks. In *Proceedings of the 3rd workshop on Adaptive and Reflective Middleware*, pages 256–261, Toronto, Ontario, Canada, 2004. ACM Press.

[66] Nico Janssens, Wouter Joosen, and Pierre Verbaeten. NeCoMan: Middleware for Safe Distributed-Service Adaptation in Programmable Networks. *IEEE Distributed Systems Online*, 6(7), July 2005.

[67] Nico Janssens, Sam Michiels, Tom Holvoet, and Pierre Verbaeten. A Modular Approach Enforcing Safe Reconfiguration of Producer-Consumer Applications. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pages 274–283. IEEE, IEEE Computer Society, 2004.

[68] Nico Janssens, Sam Michiels, Tom Mahieu, and Pierre Verbaeten. Towards Hot-Swappable System Software: The DiPS/CuPS Component Framework. In *7th International Workshop on Component-Oriented Programming (WCOP '02), ECOOP related Workshop*, Malaga, Spain, June 2002.

[69] Nico Janssens, Sam Michiels, Tom Mahieu, and Pierre Verbaeten. Towards Transparent Hot-Swapping Support for Producer-Consumer Components. In *Proceedings of the 2nd International Workshop on Unanticipated Software Evolution (USE '03)*, pages 9–16, 2003.

[70] Nico Janssens, Elke Steegmans, Tom Holvoet, and Pierre Verbaeten. An Agent Design Method Promoting Separation Between Computation and Coordination. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC'04)*, pages 456–461, Nicosia, Cyprus, 2004. ACM Press. Special Track on Coordination Models, Languages and Applications.

[71] Scott Karlin and Larry Peterson. VERA: An Extensible Router Architecture. *Computer Networks*, 38(3):277–293, February 2002.

[72] Eric P. Kasten and Philip K. McKinley. Perimorph: Run-Time Composition and State Management for Adaptive Systems. In *24th International Conference on Distributed Computing Systems Workshops (ICDCS '04 Workshops)*, pages 332–337, Hachioji, Tokyo, Japan, March 2004. IEEE Computer Society.

[73] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[74] Tim Kindberg. Reconfiguring Client-server systems. Technical Report 630, Dept. of Computer Science, Queen Mary and Westfield College, University of London, Mile End Road, E1 4NS, London, UK, 1992.

[75] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[76] Fabio Kon, Roy H. Campbell, M. Dennis Mickunas, Klara Nahrstedt, and Francisco J. Ballesteros. 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. In *Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC '00)*, pages 201–210, Pittsburgh, Pennsylvania, USA, August 2000.

[77] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, 2002.

[78] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Claudio Magalhães, and Roy H. Campbell. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In *IFIP/ACM International Conference on Distributed systems platforms (Middleware '00)*, pages 121–143, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[79] Jeff Kramer. Configuration Programming - A Framework for the Development of Distributable Systems. In *Proceedings of IEEE International Conference on Computer Systems and Software Engineering (CompEuro 90)*, pages 374–384, Tel-Aviv, Israel, May 1990.

[80] Jeff Kramer and Jeff Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, 1990.

[81] Jeff Kramer and Jeff Magee. Analysing dynamic change in distributed software architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998.

[82] Yueh-Feng Lee and Ruei-Chuan Chang. Developing dynamic-reconfigurable communication protocol stacks using Java. *Softw. Pract. Exper.*, 35(6):601–620, 2005.

[83] Ulana Legedza, David Wetherall, and John V. Guttag. Improving the Performance of Distributed Applications Using Active Networks. In *Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 1998)*, volume 2, pages 590–599, San Francisco, CA, USA, April 1998.

[84] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[85] Janghoon Lyu, Youngjin Kim, Yongsub Kim, and Inhwan Lee. A procedure-based dynamic software update. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN 2001)*, pages 271–280, Göteborg, Sweden, June 2001.

[86] Pattie Maes. Concepts and experiments in computational reflection. In *Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA '87)*, pages 147–155, Orlando, Florida, United States, 1987. ACM Press.

[87] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime Support for Type-Safe Dynamic Java Classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, pages 337–361, June 2000.

[88] William S. Marcus, Ilija Hadzic, Antony J. McAuley, and Jonathan M. Smith. Protocol boosters: Applying programmability to network infrastructures. *IEEE Communications Magazine*, 36(10):79–83, October 1998.

[89] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. TCP Selective Acknowledgement Options, October 1996. RFC 2018.

[90] Frank Matthijs. *Component Framework Technology for Protocol Stacks*. PhD thesis, K.U. Leuven, December 1999.

[91] Philip. K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. A Taxonomy of Compositional Adaptation. Technical Report MSU-CSE-04-17, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, 2004.

[92] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing Adaptive Software. *IEEE Computer*, 37(7):56–64, July 2004.

[93] Dylan McNamee, Jonathan Walpole, Calton Pu, Crispin Cowan, Charles Krasic, Ashvin Goel, Perry Wagle, Charles Consel, Gilles Muller, and Renauld Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems*, 19(2):217–251, 2001.

[94] Tom Mens, Jim Buckley, Matthias Zenger, and Awais Rashid. Towards a Taxonomy of Software Evolution. In *Proceedings of the 2nd International Workshop on Unanticipated Software Evolution (USE '03)*, pages 78–94, 2003.

[95] Shashidhar Merugu, Samrat Bhattacharjee, Ellen W. Zegura, and Kenneth L. Calvert. Bowman: A Node OS for Active Networks. In *Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2000)*, volume 3, pages 1127–1136, Tel Aviv, Israel, March 2000.

[96] Scott Michel, Khoi Nguyen, Adam Rosenstein, Lixia Zhang, Sally Floyd, and Van Jacobson. Adaptive web caching: towards a new global caching architecture. *Computer Networks and ISDN Systems*, 30(22-23):2169–2177, 1998.

[97] Sam Michiels. *Component Framework Technology for Adaptable and Manageable Protocol Stacks*. PhD thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, November 2003.

[98] Sam Michiels, Lieven Desmet, Nico Janssens, Tom Mahieu, and Pierre Verbaeten. Self-Adapting Concurrency: the DMonA architecture. In D. Garlan, J. Kramer, and A. Wolf, editors, *Proceedings of the first workshop on Self-Healing Systems (WOSS'02)*, pages 43–48, Charleston, South Carolina, USA, 2002.

[99] Sam Michiels, Nico Janssens, Lieven Desmet, Tom Mahieu, Wouter Joosen, and Pierre Verbaeten. Connecting Embedded Devices Using a Component Platform for Adaptable Protocol Stacks. In *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends*, volume 3778/2005 of *Lecture Notes in Computer Science*, pages 185–208. Springer-Verlag, 2005.

[100] Kaveh Moazami-Goudarzi. *Consistency preserving dynamic reconfiguration of distributed systems*. PhD thesis, Imperial College, London, March 1999.

[101] Kaveh Moazami-Goudarzi and Jeff Kramer. Maintaining node consistency in the face of dynamic change. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems (ICCDS '96)*, page 62, Washington, DC, USA, 1996. IEEE Computer Society.

[102] Jonathan T. Moore, Michael W. Hicks, and Scott Nettles. Practical Programmable Packets. In *Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*, volume 1, pages 41–50, Anchorage, Alaska, USA, April 2001.

[103] Akihiro Nakao, Larry Peterson, and Andy Bavier. Constructing end-to-end paths for playing media objects. *Computer Networks (Amsterdam, Netherlands: 1999)*, 38(3):373–389, 2002.

[104] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE '98)*, pages 177–186, Kyoto, Japan, 1998. IEEE Computer Society.

[105] Harold Ossher and Peri Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, 2001.

[106] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[107] Charles E. Perkins. IP Mobility Support for IPv4, August 2002. RFC 3344.

[108] Larry Peterson, Yitzchak Gottlieb, Mike Hibler, Patrick Tullmann, Jay Lepreau, Steve Schwab, Hrishikesh Dandelkar, Andrew Purtell, and John Hartman. An OS Interface for Active Routers. *IEEE Journal on Selected Areas in Communications*, 19(3):473–487, March 2001.

[109] F. Plásil, D. Bálek, and R. Janecek. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. In *Proceedings of the International Conference on Configurable Distributed Systems (CDS '98)*, page 43, Washington, DC, USA, 1998. IEEE Computer Society.

[110] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development (AOSD '02)*, pages 141–147, New York, NY, USA, 2002. ACM Press.

[111] Jon Postel. Transmission Control Protocol specification (TCP), September 1981. RFC 793.

[112] Konstantinos Psounis. Active Networks: Applications, Security, Safety, and Architectures. *IEEE Communications Surveys*, pages 1–16, First Quarter 1999.

[113] James M. Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.

[114] Tobias Ritzau and Jesper Andersson. Dynamic Deployment of Java Applications. In *Java for Embedded Systems Workshop*, London, May 2000.

[115] Jose German Rivera, Alejandro Andres Danylyszyn, Charles B. Weinstock, Lui R. Sha, and Michael J. Gagliardi. An Architectural Description of the Simplex Architecture. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1996.

[116] Paul D. Robertson, Matt Curtin, and Marcus J. Ranum. Internet Firewalls: Frequently Asked Questions, July 2004. Available at http://www.interhack.net/pubs/fwfaq/.

[117] Lukas Ruf, Ralph Keller, and Bernhard Plattner. A Scalable High-performance Router Platform Supporting Dynamic Service Extensibility On Network and Host Processors. In *Proceedings of 2004 ACS/IEEE International Conference on Pervasive Services (ICPS)*, Beirut, Lebanon, July 2004.

[118] Seyed Masoud Sadjadi and Philip K. McKinley. ACT: An Adaptive CORBA Template to Support Unanticipated Adaptation. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS'04)*, Tokyo, Japan, March 2004.

[119] Matt Sanders, Mark Keaton, Samrat Bhattacharjee, Ken Calvert, Stephen Zabele, and Ellen Zegura. Active reliable multicast on CANEs: A case study. In *Proceedings of the 4th conference on Open Architectures and Network Programming (OpenArch 2001)*, pages 49–60, Anchorage, Alaska, april 2001.

[120] Beverly Schwartz, Wenyi Zhou, Alden W. Jackson, W. Timothy Strayer, Dennis Rockwell, and Craig Partridge. Smart packets for active networks. In *Proceedings of the 2nd conference on Open Architectures and Network Programming (OpenArch 1999)*, New York, NY, USA, March 1999.

[121] Mark E. Segal and Ophir Frieder. On-the-Fly Program Modification: Systems for Dynamic Updating. *IEEE Software*, 10(2):53–65, 1993.

[122] Lui Sha, R. Rajkumar, and M. Gagliardi. Evolving Dependable Real-Time Systems. In *Proceedings of the 1996 IEEE Aerospace Applications Conference*, pages 335–46, Aspen, CO, 3–10 1996. IEEE New York, NY, USA.

[123] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[124] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Publishing Co., 1998. ISBN 0-201-17888-5.

[125] Seng Kee Tan, Yu Ge, Kean Soon Tan, Chee Wei Ang, and Nirmalya Ghosh. Dynamically Loadable Protocol Stacks: A Message Parser–Generator Implementation. *IEEE Internet Computing*, 8(2):19–25, 2004.

[126] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, 1997.

[127] Eddy Truyen. *Dynamic and Context-Sensitive Composition in Distributed Systems*. PhD thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, November 2004.

[128] Eddy Truyen and Wouter Joosen. Run-time and atomic weaving of distributed aspects. *Transactions on Aspect-Oriented Software Development*, 2006. to appear.

[129] Eddy Truyen, Wouter Joosen, and Pierre Verbaeten. Run-time Support for Aspects in Distributed System Infrastructure. In *Proceedings of the First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS '02)*, Enschede, Netherlands, 2002.

[130] Eddy Truyen, Bart Vanhaute, Wouter Joosen, Pierre Verbaeten, and Bo Nørregaard Jørgensen. Dynamic and Selective Combination of Extensions in Component-Based Applications. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE'01)*, pages 233–242. IEEE Computer Society, May 2001.

[131] Christian Tschudin. The Messenger Environment M0 - A Condensed Description. In *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 149–156. Springer-Verlag, April 1997.

[132] Jacobus E. van der Merwe, Sean Rooney, Ian Leslie, and Simon Crosby. The tempest–A Practical Framework for Network Programmability. *IEEE Network*, 12(3):20–28, May/June 1998.

[133] Robbert van Renesse, Ken Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building Adaptive Systems Using Ensemble. *Software – Practice & Experience*, 28(9):963–979, 1998.

[134] Yves Vandewoude and Yolande Berbers. Supporting runtime evolution in SEESCOA. *Journal of Integrated Design & Process Science: Transactions of the SDPS*, 8(1):77–89, March 2004.

[135] Yves Vandewoude and Yolande Berbers. DeepCompare: Static Analysis for Runtime Software Evolution. Technical Report CW405, KULeuven, Belgium, Februari 2005.

[136] Yves Vandewoude and Yolande Berbers. Fresco: Flexible and Reliable evolution system for components. *Electronic Notes in Theoretical Computer Science*, 127(3):197–205, April 2005.

[137] Alex Villazón. A reflective active network node. In *Proceedings of the Second International Working Conference on Active Networks (IWAN '00)*, pages 87–101, London, UK, 2000. Springer-Verlag.

[138] Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Craig Rodrigues, Balachandran Natarajan, Joseph P. Loyall, Richard E. Schantz, and Christopher D. Gill. *Middleware for Communications*, chapter QoS-enabled Middleware. Wiley and Sons, 2003.

[139] Ian Warren and Ian Sommerville. Dynamic configuration abstraction. In W. Schäfer and P. Botella, editors, *Proceedings of the Fifth European Software Engineering Conference*, pages 173–190. Springer-Verlag, 1995.

[140] Ian Welch and Robert J. Stroud. Kava - A Reflective Java Based on Bytecode Rewriting. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering*, pages 155–167, London, UK, 2000. Springer-Verlag.

[141] David Wetherall, Ulana Legedza, and John Guttag. Introducing New Internet Services: Why and How. *IEEE Network Magazine, special issue on Active and Programmable Networks*, 12(3):12–19, May/June 1998.

[142] David J. Wetherall and David L. Tennenhouse. The ACTIVE IP Option. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, 1996.

[143] Gary T. Wong, Matti A. Hiltunen, and Richard D. Schlichting. A Configurable and Extensible Transport Protocol. In *Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*, volume 1, pages 319–328, Anchorage, Alaska, USA, April 2001.

[144] Yechiam Yemini and Sushil daSilva. Towards programmable networks. In *IFIP/IEEE International Workshop on Distributed Systems:Operations and Management*, Oct. 1996.

[145] Yasuhiko Yokote. The apertos reflective operating system: The concept and its implementation. In Andreas Paepcke, editor, *Proceedings of the Seventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '92)*, pages 414–434. ACM Press, 1992.

Appendix A

Overview of all reconfiguration actions

This appendix lists all reconfiguration actions that NeCoMan uses (as backing). Note that these reconfiguration actions have been chosen thoroughly. On the one hand, these actions should not be defined at a too low abstraction level, since this has a negative effect on error-proneness and comprehensibility. At the same time, however, their abstraction level should not be raised too far either, since this compromises the flexibility of the employed reconfiguration algorithms.

primitive	**description**
AC_{new}	Activate the new component
APD^{mark}_{new}	Add marking support to the service-internal outports of the affected component (APD stands for "**a**dd **p**acket-**d**istinguishing support")
APD'^{mark}_{new}	Add marking support without binding the inports of the affected marking component
$APD^{disp}_{old-new}$	Add dispatching support to de-multiplex incoming packets and delegate them towards the service-internal inports of the associated component (which can be the old or the new component)
AP_{new}	Start the active objects that the new component employs (AP stands for "**a**ctivate **p**rocesses")
AP^{client}_{new}	Start the active objects that the new client processes employ
AP^{server}_{new}	Start the active objects that the new server processes employ
CC_{new}	Create the new component
DC_{old}	Delete the old component
FC_{old}	Finish the old component
IC_{new}	Install the new component

Continued on next page

primitive	description
IP_{old}	Intercept packets directed to all inports of the affected component
IP_{old}^{client}	Intercept packets directed to every old client process that the affected component encapsulates
IP_{old}^{server}	Intercept packets directed to every old server process that the affected component encapsulates
ISS_{old}	Impose a safe state over the old component
ISS_{old}^{client}	Impose a safe state over all client processes that the affected component encapsulates
ISS_{old}^{server}	Impose a safe state over all server processes that the affected component encapsulates
LI_{new}^{int}	Bind the new component's service-internal inports (LI stands for "**l**ink **i**nports")
$LI_{old-new}$	Unbind the old component's inports and simultaneously bind those of the new component
$LI_{old-new}^{ext}$	Unbind the old component's service-external inports and simultaneously bind those of the new component
$LI_{old-new}^{'ext}$	Unbind the old component's service-external inports and simultaneously bind those of the new marking components
$LI_{old-new}^{int}$	Unbind the old component's service-internal inports and simultaneously bind those of the new component
LO_{new}	Bind all outports of the new component (LO stands for "**l**ink **o**utports")
LO_{new}^{ext}	Bind the new component's service-external outports
LO_{new}^{int}	Bind the new component's service-internal outports
RC_{old}	Remove the old component
RP_{new}	Release packets directed to the new component
RP_{new}^{client}	Release packets directed to the new client processes
RP_{new}^{server}	Release packets directed to the new server processes
RPD_{new}^{mark}	Remove the employed marking support (RPD stands for "**r**emove **p**acket-**d**istinguishing support")
$RPD_{old-new}^{disp}$	Remove the employed dispatching support
UI_{old}^{int}	Unbind the old component's service-internal inports (UI stands for "**u**nlink **i**nports")
UO_{old}	Unbind all outports of the old component (UO stands for "**u**nlink **o**utports")
UO_{old}^{ext}	Unbind the old component's service-external outports
UO_{old}^{int}	Unbind the old component's service-internal outports

Table A.1: An overview of all NeCoMan's reconfiguration actions.

Appendix B

Replacement of retransmission component

In this appendix, we exemplify NeCoMan's reconfiguration algorithm for carrying out local reconfigurations of distributed services with the dynamic replacement of R_{old} by R_{new}. Figure 3.2 sketches the protocol stack of the affected node before and after this replacement has occurred. In addition, the Petri net in Figure B.1 models how NeCoMan coordinates this reconfiguration. Note that this model originates from NeCoMan's local reconfiguration algorithm, and is tailored to the characteristics of the retransmission component. As we further explain in the remainder of this section, this tailoring involves discarding a number of transitions that become redundant when replacing R_{old}.

Besides illustrating this reconfiguration algorithm with a concrete example, we also demonstrate that NeCoMan implements each reconfiguration action by only invoking the eight node operations listed in Section 3.3.1. This indicates that NeCoMan is prepared to coordinate the recomposition of various component-based, flow-oriented protocol stack architectures, given that these architectures provide the reconfiguration support needed to assist NeCoMan in carrying out reconfigurations.

Finally, once again we use black bold components and connections to graphically symbolize components and connections that are created after completing a reconfiguration action. Grey bold components and connections, in contrast, represent components and connections that are removed.

B.1 Installing new retransmission component

The replacement begins by installing the new retransmission component on the affected node. This requires first loading R_{new} into the node's reconfiguration support. Next, NeCoMan instructs the affected node to connect the new retransmission

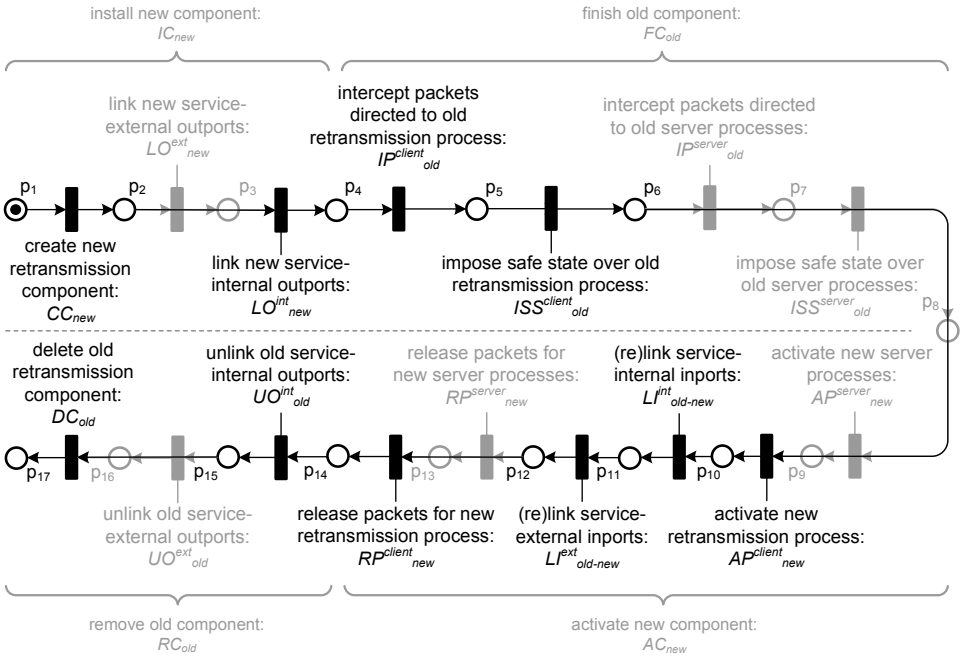


Figure B.1: Implementation of the reconfiguration algorithm that NeCoMan uses to replace R_{old} with R_{new}

component into its protocol stack composition. Because the new retransmission component should not be activated in this phase, the latter is limited to connecting R_{new}'s data-outport with the inport of the lower layer. Figure B.2 depicts the composition of the affected node after installing R_{new}. Besides, note that because R_{new} contains only a client process (the retransmission processes), there is no need to execute LO^{ext}_{new} (see Figure B.1).

NeCoMan implements the installation of R_{new} by invoking the "create" and "link" operations that the underlying node's reconfiguration support provides. To illustrate this, Listing B.1 depicts how NeCoMan instructs the underlying node to load and bind R_{new}.

```
create(component_id="Rnew",class_name="NewRetrComp");
link(source_comp="Rnew",source_port="data-outport",
     dest_comp="LowerLayer",dest_port="inport");
```

Listing B.1: NeCoMan instructing the node's reconfiguration support to install R_{new}

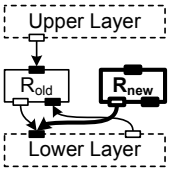


Figure B.2: Installation of R_{new}

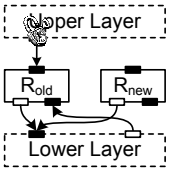


Figure B.3: Finishing of R_{old}

B.2 Finishing old retransmission component

Next, R_{old} becomes finished. NeCoMan first instructs the affected node to intercept packets that invoke R_{old}'s retransmission process. As a result, this node's reconfiguration support intercepts all packets directed to R_{old}'s data-inport (as illustrated in Figure B.3). Once this is completed, NeCoMan instructs the affected node to impose a reconfiguration-safe state over R_{old}'s retransmission process. Presume that this involves first monitoring R_{old}'s retransmission queue until it is empty. After that, the node's reconfiguration support stops R_{old}'s retransmission timer, captures the last sequence number that R_{old} has attached to an outgoing packet, and reinstates this information in R_{new}. Listing B.2 illustrates how NeCoMan instructs the node's reconfiguration support to accomplish this. Besides, note that because R_{old} encapsulates only a client process, reconfiguration actions IP_{old}^{server} and ISS_{old}^{server} become redundant and therefore can be discarded.

```
intercept_packets(component_id="Rold",process_id="retransmit");
impose_safe_state(component_id="Rold",process_id="retransmit");
```

Listing B.2: NeCoMan instructing the node's reconfiguration support to finish R_{old}

B.3 Activating new retransmission component

Next, R_{new} can safely be activated. To achieve this, NeCoMan first instructs the reconfiguration support of the affected node to relink all service-internal and service-external inports. As illustrated in Figure B.4, this involves disconnecting R_{old}'s ack-inport and data-inport from the outports of the upper layer and of the lower layer, respectively. Next, these outports become reconnected to the ack-inport and data-inport of R_{new}. Once this is accomplished, NeCoMan instructs the affected node to activate the new retransmission process. This results in starting this process' retransmission timer. After that, R_{new} is prepared to be brought in use. NeCoMan then completes the activation phase by invoking the affected node to release all intercepted packets (see Figure B.5). Listing B.3 illustrates the instructions that NeCoMan uses to activate R_{new}. In addition, note that because R_{new} does not

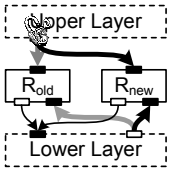


Figure B.4: Binding inports of R_{new}

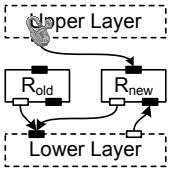


Figure B.5: Releasing intercepted packets

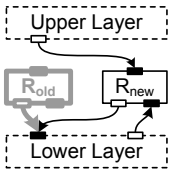


Figure B.6: Removal of R_{old}

contain server processes either, reconfiguration actions AP_{new}^{server} and RP_{new}^{server} can be discarded (as illustrated in Figure B.1).

```
activate_processes(component_id="Rnew",process_id="retransmit");
unlink(source_comp="LowerLayer",source_port="outport",
     dest_comp="Rold",dest_port="ack-inport");
link(source_comp="LowerLayer",source_port="outport",
     dest_comp="Rnew",dest_port="ack-inport");
unlink(source_comp="UpperLayer",source_port="outport",
     dest_comp="Rold",dest_port="data-inport");
link(source_comp="UpperLayer",source_port="outport",
     dest_comp="Rnew",dest_port="data-inport");
release_packets(component_id="Rnew",process_id="retransmit");
```

Listing B.3: NeCoMan instructing the node's reconfiguration support to activate R_{new}

B.4 Removing old retransmission component

Finally, the old retransmission component becomes removed from the affected node, as illustrated in Figure B.6. This involves first disconnecting R_{old} from the node's protocol stack composition by unlinking the outports of this component. Next, NeCoMan instructs the node's reconfiguration support to delete R_{old}. Because R_{old} has already been finished, the removal of this component will not compromise the correct network operation.

To remove R_{old}, NeCoMan invokes the underlying node's reconfiguration support to disconnect and delete this component by using the "unlink" and "remove" primitives, respectively. More in detail, Listing B.4 sketches how NeCoMan instructs the local removal of R_{old}. Finally, note that because R_{old} contains no server processes, there is no need to execute UO_{new}^{ext}.

```
unlink(source_comp="Rold",source_port="data-outport",
       dest_comp="LowerLayer",dest_port="inport");
remove(component_id="Rold");
```

Listing B.4: NeCoMan instructing the node's reconfiguration support to remove R_{old}

Appendix C

Correctness of NeCoMan's algorithm for local reconfiguration of distributed services

This appendix demonstrates that NeCoMan's algorithm for local reconfiguration of distributed services meets all required reconfiguration conditions. These conditions are listed in Table 3.3. Related to the algorithm's Petri net model, each one of these reconfiguration condition formalizes a pre-condition to fire a transition. Therefore, to demonstrate that the presented algorithm conducts correct reconfigurations, we check for every transition modelled in Figure 3.18 if all pre-conditions (that the associated reconfiguration conditions impose) are met.

1. **Create new service component**. From Table 3.3 we conclude that no pre-conditions must be fulfilled to safely execute CC_{new}. Consequently, there is no need to coordinate firing the transition that models the execution of CC_{new}.

2. **Link new service-external and service-internal outports**. The execution of LO^{ext}_{new} and LO^{int}_{new} can only be initiated when CC_{new} is finished, as dictated by reconfiguration condition (3.1). We conclude from Table 3.4 that this condition is fulfilled as from reaching place p_2. Because p_2 is the input place of LO^{ext}_{new} and the ancestor place of LO^{int}_{new}, the algorithm meets this reconfiguration condition.

3. **Intercept packets directed to old client processes**. Similar to the execution of CC_{new}, we conclude from Table 3.3 that no pre-condition must be

satisfied to correctly execute IP_{old}^{client}.

4. **Impose safe state over old client processes**. As defined by reconfiguration condition (3.2), NeCoMan can only instruct a node to carry out ISS_{old}^{client} when IP_{old}^{client} is completed. Table 3.4 indicates that this condition is met as from reaching place p_5. Because p_5 is the input place of ISS_{old}^{client}, the algorithm meets this reconfiguration condition.

5. **Intercept packets directed to old server processes**. Similar to the execution of IP_{old}^{client}, no pre-conditions must be satisfied either to correctly execute IP_{old}^{server}.

6. **Impose safe state over old server processes**. As reconfiguration condition (3.3) dictates, the execution of ISS_{old}^{server} can only be started when IP_{old}^{server} has completed. This condition is met as from reaching place p_7, which is the input place of the transition that defines the execution of ISS_{old}^{server}. The algorithm thus meets this reconfiguration condition as well.

7. **Activate new server processes**. As defined by reconfiguration conditions (3.9) and (3.12), NeCoMan can only instruct the execution of AP_{new}^{server} when LO_{new}^{ext}, LO_{new}^{int}, and ISS_{old}^{server} have completed. This is accomplished as from reaching place p_8, which is the input place of the transition that models the execution of AP_{new}^{server}.

8. **Activate new client processes**. According to reconfiguration conditions (3.8) and (3.11), the execution of AP_{new}^{client} can only be initiated safely when both LO_{new}^{int} and ISS_{old}^{client} are completed. This pre-condition is met as from reaching place p_6, which is an ancestor place of the transition that models AP_{new}^{client}.

9. **(Re)link service-internal inports**. The execution of $LI_{old-new}^{int}$ can only be initiated safely when both CC_{new}, ISS_{old}^{client}, and ISS_{old}^{server} are completed, as dictated by reconfiguration conditions (3.7) and (3.10). As we can deduce from Table 3.4, this condition is met as from reaching place p_8. Because p_8 is an ancestor place of the transition that models $LI_{old-new}^{int}$, the algorithm does not violate this pre-condition.

10. **(Re)link service-external inports**. According to reconfiguration conditions (3.7) and (3.11), the execution of $LI_{old-new}^{ext}$ can only be initiated safely when both CC_{new} and ISS_{old}^{client} are completed. As we can deduce from Table 3.4, this condition is fulfilled as from reaching place p_6. Because this is the ancestor place of $LI_{old-new}^{ext}$, the algorithm meets this pre-condition as well.

11. **Release packets for new server processes**. Reconfiguration conditions (3.5) and (3.9) dictate that a node's reconfiguration support can only be invoked to initiate RP_{new}^{server} once AP_{new}^{server}, $LI_{old-new}^{int}$, LO_{new}^{ext}, and LO_{new}^{int}

are completed on that node. This is met as from reaching place p_{11}, which is an ancestor place of the transition that models RP^{server}_{new}. The algorithm thus meets this requirement.

12. **Release packets for new client processes**. According to reconfiguration conditions (3.4) and (3.8), the execution of RP^{client}_{new} can only be started when AP^{client}_{new}, $LI^{int}_{old-new}$, $LI^{ext}_{old-new}$, and LO^{int}_{new} has completed. This pre-condition is met as from reaching place p_{12}, which is an ancestor place of the transition that models RP^{client}_{new}.

13. **Unlink old service-internal outports**. As defined by reconfiguration condition (3.13), the execution of UO^{old}_{int} should only be fired when both ISS^{client}_{old} and ISS^{server}_{old} are completed. This pre-condition is met as from reaching place p_8, which is an ancestor place of the transition that models UO^{old}_{int}. The algorithm thus fulfills this pre-condition.

14. **Unlink old service-external outports**. Reconfiguration condition (3.14) dictates that UO^{old}_{ext} can only be initiated once ISS^{server}_{old} has completed. The latter is accomplished as from reaching place p_8. Because p_8 is an ancestor place of the transition that models the execution of UO^{old}_{ext}, the algorithm complies with this reconfiguration condition as well.

15. **Delete old service component**. Finally, reconfiguration condition (3.6) dictates that a node's reconfiguration support can only be invoked to initiate DC_{old} once UO^{ext}_{old}, UO^{int}_{old}, $LI^{ext}_{old-new}$, and $LI^{int}_{old-new}$ are completed on that node. This pre-condition is met as from reaching place p_{16}. Because this is the input place of the transition that models the execution of DC_{old}, the algorithm fulfills this requirement as well.

So, we conclude that the algorithm depicted in Figure 3.18 fulfills all required reconfiguration conditions, and therefore yields correct reconfigurations.

Appendix D

Affected reconfiguration conditions when customizing NeCoMan's algorithms for local reconfigurations

As explained in detail in Chapter 4, NeCoMan incorporates an extensive set of customizations that apply to its local reconfiguration algorithms. These customizations seek to optimize the reconfiguration scenario by exploiting both the characteristics of the affected services and the reconfiguration semantics. Hence, each customization affects some of the reconfiguration conditions that NeCoMan must fulfill to conduct a correct reconfiguration. This appendix lists the reconfiguration conditions that are changed when applying customizations 4.4 to 4.7. Note that the effect of customizations 4.2 and 4.3 has already been discussed in detail in Chapter 4.

We first focus on customization 4.4, which involves the use of active objects. To be precise, Tables D.1 and D.2 list all reconfiguration conditions that are affected when the client and server processes of the new component do not use active objects. In addition, Table D.3 lists all reconfiguration conditions that are changed when isolated network-service components are involved and the new service component (again) does no employ active objects.

Next, we concentrate on customization 4.5. This customization involves omitting some reconfiguration actions when the old and new components encapsulate only client or server processes, instead of both. Table D.4 lists all reconfiguration conditions that are changed when the affected components encapsulate only client processes. Similarly, Table D.5 lists all reconfiguration conditions that are changed when the affected components encapsulate only server processes.

After that, we focus on customization 4.6. This customization is targeted at

affected reconfiguration condition	**resulting safety condition**
(3.4)	$RP^{client}_{new} \leftarrow LI^{int}_{old-new} \wedge LI^{ext}_{old-new}$
(3.8)	$RP^{client}_{new} \leftarrow LO^{int}_{new}$
(3.11)	$LI^{ext}_{old-new} \leftarrow ISS^{client}_{old}$
(4.1)	$AP^{server}_{new} \leftarrow CC_{new}$
(4.2)	$LI^{int}_{old-new} \leftarrow AP^{server}_{new} \wedge LO^{int}_{new} \wedge LO^{ext}_{new}$
(4.3)	$LI^{ext}_{old-new} \leftarrow LO^{int}_{new} \wedge LI^{int}_{old-new}$

Table D.1: Customization of NeCoMan's algorithm for local reconfigurations that involve distributed services: overview of all reconfiguration conditions that become changed when the new component's client processes do not use active objects.

affected reconfiguration condition	**resulting safety condition**
(3.5)	$RP^{server}_{new} \leftarrow LI^{int}_{old-new}$
(3.9)	$RP^{server}_{new} \leftarrow LO^{ext}_{new} \wedge LO^{int}_{new}$
(3.12)	N/A
(4.1)	$AP^{client}_{new} \leftarrow CC_{new}$
(4.2)	$LI^{int}_{old-new} \leftarrow AP^{client}_{new} \wedge LO^{int}_{new} \wedge LO^{ext}_{new}$

Table D.2: Customization of NeCoMan's algorithm for local reconfigurations that involve distributed services: overview of all reconfiguration conditions that become changed when the new component's server processes do not use active objects.

omitting reconfiguration actions that involve service-internal inports and outports. Hence, Table D.6 lists all reconfiguration conditions that are changed when NeCoMan manages a node with only client processes, and these processes communicate by a unidirectional communication protocol. Besides, Table D.7 lists all reconfiguration conditions that become affected when NeCoMan manages a node accommodating only server processes, which communicate (also) by a unidirectional communication protocol.

Finally, we concentrate on customization 4.7. This customization seeks to tailor NeCoMan's second local reconfiguration algorithm when services are to be added or removed instead of being replaced. Table D.8 lists all reconfiguration conditions that are affected in case of service addition. Similarly, Table D.9 lists all reconfiguration conditions that are changed when a reconfiguration involves service removal.

affected reconfig-uration condition	**resulting safety condition**
(3.17)	$RP_{new} \leftarrow LI_{old-new}$
(3.20)	$RP_{new} \leftarrow LO_{new}$
(3.22)	$RP_{new} \leftarrow ISS_{old}$
(4.6)	N/A
(4.7)	$LI_{old-new} \leftarrow LO_{new}$

Table D.3: Customization of NeCoMan's algorithm for local reconfigurations that involve isolated services: overview of the reconfiguration conditions that become changed when the new component's processes do not use active objects.

affected reconfiguration condition	**resulting safety condition**
(3.1)	$LO^{int}_{new} \leftarrow CC_{new}$
(3.3)	N/A
(3.5)	N/A
(3.6)	$DC_{old} \leftarrow UO^{int}_{old} \wedge$ $LI^{ext}_{old-new} \wedge LI^{int}_{old-new}$
(3.9)	N/A
(3.10)	$LI^{int}_{old-new} \leftarrow ISS^{client}_{old}$
(3.12)	N/A
(3.13)	$UO^{int}_{old} \leftarrow ISS^{client}_{old}$
(3.14)	N/A
(4.1)	$AP^{client}_{new} \leftarrow CC_{new}$
(4.2)	$LI^{int}_{old-new} \leftarrow$ $AP^{client}_{new} \wedge LO^{int}_{new}$
(4.5)	N/A
(4.10)	N/A

Table D.4: Customization of NeCoMan's algorithm for local reconfigurations that involve distributed services: overview of all reconfiguration conditions that become adapted when the old and new component encapsulate only client processes.

affected reconfiguration condition	**resulting safety condition**
(3.2)	N/A
(3.4)	N/A
(3.6)	$DC_{old} \leftarrow UO^{int}_{old} \wedge$ $UO^{ext}_{old} \wedge LI^{int}_{old-new}$
(3.7)	$LI^{int}_{old-new} \leftarrow CC_{new}$
(3.8)	N/A
(3.10)	$LI^{int}_{old-new} \leftarrow ISS^{server}_{old}$
(3.11)	N/A
(3.13)	$UO^{int}_{old} \leftarrow ISS^{server}_{old}$
(4.1)	$AP^{server}_{new} \leftarrow CC_{new}$
(4.2)	$LI^{int}_{old-new} \leftarrow AP^{server}_{new}$ $\wedge LO^{ext}_{new} \wedge LO^{int}_{new}$
(4.3)	N/A
(4.4)	N/A
(4.9)	$UO^{int}_{old} \leftarrow LI^{int}_{old-new}$

Table D.5: Customization of NeCoMan's algorithm for local reconfigurations that involve distributed services: overview of all reconfiguration conditions that become adapted when the old and new component encapsulate only server processes.

affected reconfiguration condition	**resulting safety condition**
(3.1)	$LO^{int}_{new} \leftarrow CC_{new}$
(3.3)	N/A
(3.5)	N/A
(3.6)	$DC_{old} \leftarrow UO^{int}_{old} \wedge LI^{ext}_{old-new}$
(3.9)	N/A
(3.10)	N/A
(3.12)	N/A
(3.13)	$UO^{int}_{old} \leftarrow ISS^{client}_{old}$
(3.14)	N/A
(4.1)	$AP^{client}_{new} \leftarrow CC_{new}$
(4.2)	N/A
(4.3)	$LI^{ext}_{old-new} \leftarrow AP^{client}_{new} \wedge LO^{int}_{new}$
(4.5)	N/A
(4.10)	N/A

Table D.6: Customization of NeCoMan's algorithm for local reconfigurations that involve distributed services: overview of all reconfiguration conditions that become adapted when the old and new component encapsulate only client processes and when these processes employ a unidirectional communication protocol

affected reconfiguration condition	**resulting safety condition**
(3.2)	N/A
(3.4)	N/A
(3.6)	$DC_{old} \leftarrow UO^{ext}_{old} \wedge LI^{int}_{old-new}$
(3.7)	$LI^{int}_{old-new} \leftarrow CC_{new}$
(3.8)	N/A
(3.10)	$LI^{int}_{old-new} \leftarrow ISS^{server}_{old}$
(3.11)	N/A
(3.13)	N/A
(4.1)	$AP^{server}_{new} \leftarrow CC_{new}$
(4.2)	$LI^{int}_{old-new} \leftarrow AP^{server}_{new} \wedge LO^{ext}_{new}$
(4.3)	N/A
(4.4)	N/A
(4.9)	N/A

Table D.7: Customization of NeCoMan's algorithm for local reconfigurations that involve distributed services: overview of all reconfiguration conditions that become adapted when the old and new component encapsulate only server processes and when these processes employ a unidirectional communication protocol

affected reconfiguration condition	**resulting safety condition**
(3.16)	N/A
(3.17)	N/A
(3.18)	N/A
(3.20)	$AP_{new} \leftarrow CC_{new}$
(3.21)	N/A
(3.22)	N/A
(3.23)	N/A
(4.8)	N/A
(4.11)	N/A

Table D.8: Customization of NeCoMan's algorithm for local reconfigurations that involve isolated services: overview of all reconfiguration conditions that become changed in case of service addition.

affected reconfiguration condition	**resulting safety condition**
(3.15)	N/A
(3.17)	$RP_{new} \leftarrow LI_{old-new}$
(3.19)	N/A
(3.20)	N/A
(3.22)	$RP_{new} \leftarrow ISS_{old}$
(4.6)	N/A
(4.7)	N/A

Table D.9: Customization of NeCoMan's algorithm for local reconfigurations that involve isolated services: overview of all reconfiguration conditions that become changed in case of service removal.

Appendix E

Effect of local customizations on reconfiguration overhead

This appendix analyzes the effect on reconfiguration overhead that the first two customizations presented in Chapter 4 bring about. Recall that these customizations seek to optimize NeCoMan's local reconfiguration algorithms by activating the new service before the old one is finished, and by discarding the finishing actions. The other customizations presented in Chapter 4 seek to tailor the implementation of a reconfiguration algorithm by discarding those actions that are redundant for a particular reconfiguration. The effect of these customizations on the overhead that a reconfiguration causes, therefore, is rather straight forward. Hence, this appendix focusses only on the effect of the first two customizations.

To evaluate the effect of these customizations, we compare the cost incurred by NeCoMan's local algorithms (depicted in Figures 3.18 and 3.22) with the cost incurred when these customizations are applied. We evaluate this cost in terms of (1) the *communication disruption* that these algorithms cause, as well as (2) the *reconfiguration time*. The first metric quantifies the period of time in which the affected node will be unable to process packets. This period must be as small as possible, so as to minimize the impact of a dynamic reconfiguration on the network's quality attributes (which include availability, response-time, and throughput [59]). The second metric quantifies the time that it takes to complete a reconfiguration.

Finally, note that the same reconfiguration is involved when comparing two different algorithms. If this is not the case, a comparison of the cost incurred by these two algorithms obviously is not representative.

E.1 Activate before finishing

As discussed in Section 4.2, a first customization that applies to NeCoMan's local reconfiguration algorithms involves activating a new component before the old one is finished. This customization seeks to reduce the communication disruption that a reconfiguration causes.

E.1.1 Local reconfigurations of distributed services

To evaluate the effect of this customization one on the reconfiguration overhead that NeCoMan's first local reconfiguration algorithm causes, we compare the cost incurred by the algorithm modelled in Figure 3.18 with the cost incurred by using the algorithm depicted in Figure 4.2.

Communication disruption

When NeCoMan employs its first local reconfiguration algorithm, then communication continuity will be disrupted as from the moment that packets are intercepted until these packets are released again. So, during reconfiguration a component's client processes are disrupted from the moment that NeCoMan executes IP_{old}^{client} until RP_{new}^{client} is completed. We can infer from the model of NeCoMan's first reconfiguration algorithm (depicted in Figure 3.18) that this period of communication disruption equals

$$\begin{aligned}\Delta_{disrupt}^{basic1}(client) &= t(IP_{old}^{client}) + t(IP_{old}^{server}) + t(ISS_{old}^{client}) + t(ISS_{old}^{server}) + \\ &\quad t(AP_{new}^{client}) + t(AP_{new}^{server}) + t(LI_{old-new}^{ext}) + t(LI_{old-new}^{int}) + \\ &\quad t(RP_{new}^{client}) + t(RP_{new}^{server})\end{aligned}$$

where *t(X)* denotes the time that it takes to complete the execution of reconfiguration action *X*.

Besides, a component's server processes are disrupted as from the moment that NeCoMan executes IP_{old}^{server} until RP_{new}^{server} is completed. Again we can infer from the model depicted in Figure 3.18 that this period of communication disruption equals

$$\begin{aligned}\Delta_{disrupt}^{basic1}(server) &= t(IP_{old}^{server}) + t(ISS_{old}^{server}) + t(AP_{new}^{client}) + t(AP_{new}^{server}) + \\ &\quad t(LI_{old-new}^{ext}) + t(LI_{old-new}^{int}) + t(RP_{new}^{server})\end{aligned}$$

When NeCoMan activates the new service component before finishing the old one, however, IP_{old}^{client}, IP_{old}^{server}, RP_{new}^{client} and RP_{new}^{server} become discarded. Besides, the execution of ISS_{old}^{client} and ISS_{old}^{server} will not affect service continuity because the new service component is already taken into use. Since AP_{new}^{client} and AP_{new}^{server} do not cause communication disruption either, only the execution of $LI_{old-new}^{ext}$ and $LI_{old-new}^{int}$ will affect service continuity. Hence, the period in which a client

process becomes disrupted when NeCoMan activates the new network service before finishing the old one equals

$$\Delta_{disrupt}^{basic1+cust1}(client) \quad = \quad t(LI_{old-new}^{ext}) + t(LI_{old-new}^{int})$$

Because activating the new server processes does not involve the execution of $LI_{old-new}^{ext}$, the period in which a server process becomes disrupted when the new component is activated before finishing the old one equals

$$\Delta_{disrupt}^{basic1+cust1}(server) \quad = \quad t(LI_{old-new}^{int})$$

So, the difference in communication disruption that both scenarios cause on the affected client processes equals

$$\begin{aligned}\Delta_{disrupt}^{basic1}(client) \quad - \quad & \Delta_{disrupt}^{basic1+cust1}(client) = t(IP_{old}^{client}) + t(IP_{old}^{server}) + \\ & t(ISS_{old}^{client}) + t(ISS_{old}^{server}) + t(AP_{new}^{client}) + t(AP_{new}^{server}) + \\ & t(RP_{new}^{client}) + t(RP_{new}^{server})\end{aligned}$$

Similarly, the difference in communication disruption that both scenarios cause on the affected server processes equals

$$\begin{aligned}\Delta_{disrupt}^{basic1}(server) \quad - \quad & \Delta_{disrupt}^{basic1+cust1}(server) = t(IP_{old}^{server}) + t(ISS_{old}^{server}) + \\ & t(AP_{new}^{client}) + t(AP_{new}^{server}) + t(LI_{old-new}^{ext}) + t(RP_{new}^{server})\end{aligned}$$

We conclude from these equations that the first local reconfiguration algorithm indeed causes more communication disruption than when the new component is activated before finishing the old one.

Reconfiguration time

From the model of NeCoMan's first local reconfiguration algorithm (depicted in Figure 3.18), we can infer that the time it takes to complete a basic reconfiguration equals

$$\begin{aligned}\Delta_{reconf}^{basic1} \quad = \quad & t(CC_{new}) + t(LO_{new}^{ext}) + t(LO_{new}^{int}) + \\ & t(IP_{old}^{client}) + t(IP_{old}^{server}) + t(ISS_{old}^{client}) + t(ISS_{old}^{server}) + \\ & t(AP_{new}^{client}) + t(AP_{new}^{server}) + t(LI_{old-new}^{ext}) + t(LI_{old-new}^{int}) + \\ & t(RP_{new}^{server}) + t(RP_{new}^{client}) + t(UO_{old}^{ext}) + t(UO_{old}^{int}) + t(DC_{old})\end{aligned}$$

Besides, the time that it takes to complete the algorithm modelled in Figure 4.2 equals

$$\begin{aligned}\Delta_{reconf}^{basic1+cust1} \quad = \quad & t(CC_{new}) + t(LO_{new}^{ext}) + t(LO_{new}^{int}) + \\ & t(ISS_{old}^{client}) + t(ISS_{old}^{server}) + t(AP_{new}^{client}) + t(AP_{new}^{server}) + \\ & t(LI_{old-new}^{ext}) + t(LI_{old-new}^{int}) + t(UO_{old}^{ext}) + t(UO_{old}^{int}) + t(DC_{old})\end{aligned}$$

Comparing Δ_{reconf}^{basic1} with $\Delta_{reconf}^{basic1+cust1}$ then results in

$$\Delta_{reconf}^{basic1} - \Delta_{reconf}^{basic1+cust1} = t(IP_{old}^{client}) + t(IP_{old}^{server}) + t(RP_{new}^{server}) + t(RP_{new}^{client})$$

From this, we conclude that NeCoMan takes more time to complete a reconfiguration when using its first local reconfiguration algorithm than when the new component is activated before the old one is finished.

E.1.2 Local reconfigurations of isolated services

To evaluate the effect of this customization on the reconfiguration overhead that NeCoMan's second local reconfiguration algorithm causes, we compare the cost incurred by the algorithm modelled in Figure 3.22 with the cost incurred by using the algorithm depicted in Figure 4.4.

Communication disruption

Similar as for NeCoMan's first local reconfiguration algorithm, all affected processes are disrupted as from the moment that NeCoMan executes IP_{old} until RP_{new} is completed. From the model of NeCoMan's second reconfiguration algorithm (depicted in Figure 3.22), we can infer that this period of communication disruption equals

$$\Delta_{disrupt}^{basic2} = t(IP_{old}) + t(ISS_{old}) + t(AP_{new}) + t(LI_{old-new}) + t(RP_{new})$$

When the new composition is activated before finishing the old one, IP_{old} and RP_{new} become discarded. Besides, ISS_{old} will be executed while the new composition is already activated, and AP_{new} does not affect service continuity. So, the period in which processes become disrupted when NeCoMan activates the new network service before finishing the old one equals

$$\Delta_{disrupt}^{basic2+cust1} = t(LI_{old-new})$$

So, the difference in communication disruption that both algorithms cause on the affected processes equals

$$\Delta_{disrupt}^{basic2} - \Delta_{disrupt}^{basic2+cust1} = t(IP_{old}) + t(ISS_{old}) + t(AP_{new}) + t(RP_{new})$$

From this, we conclude that a reconfiguration causes less communication disruption when a new component is already activated before finishing the old one.

Reconfiguration time

From the model of NeCoMan's second local reconfiguration algorithm (depicted in Figure 3.22), we can infer that the time it takes to complete this reconfiguration equals

$$\begin{aligned}\Delta_{reconf}^{basic2} = {} & t(CC_{new}) + t(LO_{new}) + t(IP_{old}) + t(ISS_{old}) + t(AP_{new}) + \\ & t(LI_{old-new}) + t(RP_{new}) + t(UO_{old}) + t(DC_{old})\end{aligned}$$

Besides, the time it takes to complete the algorithm modelled in Figure 4.4 equals

$$\begin{aligned}\Delta_{reconf}^{basic2+cust1} &= t(CC_{new}) + t(LO_{new}) + t(ISS_{old}) + t(AP_{new}) + \\ &\quad t(LI_{old-new}) + t(UO_{old}) + t(DC_{old})\end{aligned}$$

Comparing both algorithms results in

$$\Delta_{reconf}^{basic2} - \Delta_{reconf}^{basic2+cust1} = t(IP_{old}) + t(RP_{new})$$

From this, we can conclude that it takes less time to complete a reconfiguration when the new component is activated before finishing the old one.

E.2 No finishing

A second customization to optimize NeCoMan's local reconfiguration algorithms involves omitting all finishing actions. This customization has been discussed in detail in Section 4.3.

E.2.1 Local reconfigurations of distributed services

To evaluate the effect of this customization on the reconfiguration overhead that NeCoMan's first local reconfiguration algorithm causes, we compare the cost incurred by the algorithm modelled in Figure 3.18 with the cost incurred by using the algorithm depicted in Figure 4.6.

Communication disruption

When comparing NeCoMan's first local reconfiguration algorithm (depicted in Figure 3.18) with the model illustrated in Figure 4.6, we can infer that

$$\begin{aligned}\Delta_{disrupt}^{basic1}(client) - \Delta_{disrupt}^{basic1+cust2}(client) &= t(IP_{old}^{client}) + t(IP_{old}^{server}) + \\ &\quad t(ISS_{old}^{client}) + t(ISS_{old}^{server}) + t(AP_{new}^{client}) + t(AP_{new}^{server}) + \\ &\quad t(RP_{new}^{client}) + t(RP_{new}^{server})\end{aligned}$$

Similarly, the difference in communication disruption that both algorithms cause on the affected server processes equals

$$\begin{aligned}\Delta_{disrupt}^{basic1}(server) - \Delta_{disrupt}^{basic1+cust2}(server) &= t(IP_{old}^{server}) + t(ISS_{old}^{server}) + \\ &\quad t(AP_{new}^{client}) + t(AP_{new}^{server}) + t(LI_{old-new}^{ext}) + t(RP_{new}^{server})\end{aligned}$$

So, we can conclude that a reconfiguration causes less communication disruption when all finishing actions are omitted. Note, however, that this comparison does

not take into account the effect of inconsistencies on the network performance in general. When a recomposition involves components that belong to a multi-element protocol booster, for instance, the network will operate with lower performance until these inconsistencies are handled. In addition, when various packets get lost during a reconfiguration, TCP reduces its congestion window. Again, this affects the network's performance. It is clear that these performance penalties must also be taken into account when evaluating the benefit associated with omitting all finishing actions.

Reconfiguration time

The time that it takes to complete the algorithm depicted in Figure 4.6 equals

$$\begin{aligned}\Delta_{reconf}^{basic1+cust2} &= t(CC_{new}) + t(LO_{new}^{ext}) + t(LO_{new}^{int}) + \\ &\quad t(AP_{new}^{client}) + t(AP_{new}^{server}) + t(LI_{old-new}^{ext}) + t(LI_{old-new}^{int}) + \\ &\quad t(UO_{old}^{ext}) + t(UO_{old}^{int}) + t(DC_{old})\end{aligned}$$

Compare this with the time that it takes to complete a basic reconfiguration results in

$$\begin{aligned}\Delta_{reconf}^{basic1} - \Delta_{reconf}^{basic1+cust2} &= t(IP_{old}^{client}) + t(IP_{old}^{server}) + t(ISS_{old}^{client}) + \\ &\quad t(ISS_{old}^{server}) + t(RP_{new}^{server}) + t(RP_{new}^{client})\end{aligned}$$

From this, we can conclude that a reconfiguration takes less time to complete when all finishing actions are omitted.

E.2.2 Local reconfigurations of isolated services

To evaluate the effect of this customization on the reconfiguration overhead that NeCoMan's second local reconfiguration algorithm causes, we compare the cost incurred by the algorithm modelled in Figure 3.22 with the cost incurred by using the algorithm depicted in Figure 4.8.

Communication disruption

If we compare the communication disruption that NeCoMan's second local reconfiguration algorithm causes with the communication disruption that the algorithm depicted in Figure 4.8 brings about, then we can infer that

$$\Delta_{disrupt}^{basic2} - \Delta_{disrupt}^{basic2+cust2} = t(IP_{old}) + t(ISS_{old}) + t(AP_{new}) + t(RP_{new})$$

From this, we can conclude that also for NeCoMan's second reconfiguration algorithm communication disruption becomes reduced when all finishing actions are omitted. Again, this evaluation does not take into account the effect of packet loss on the network performance in general.

Reconfiguration time

The time it takes for the algorithm depicted in Figure 4.8 to complete a reconfiguration equals

$$\begin{aligned}\Delta_{reconf}^{basic2+cust2} &= t(CC_{new}) + t(LO_{new}) + t(AP_{new}) + \\ &\quad t(LI_{old-new}) + t(UO_{old}) + t(DC_{old})\end{aligned}$$

Comparing this with the time that it takes for NeCoMan's second local algorithm to complete results in

$$\Delta_{reconf}^{basic2} - \Delta_{reconf}^{basic2+cust2} = t(IP_{old}) + t(ISS_{old}) + t(RP_{new})$$

From this, we can conclude that also for the seconding local algorithm it takes less time to complete a reconfiguration when all finishing actions are omitted.

Appendix F

Customization procedure for local reconfigurations

This appendix briefly sketches in what order NeCoMan applies the customizations presented in Chapter 4 to its local reconfiguration algorithms. The flowchart depicted in Figure F.1 models how NeCoMan tailors its local reconfiguration algorithm for distributed services. Next, Figure F.2 illustrates this customization procedure when replacing R_{old} with R_{new}. Finally, Figure F.3 models in what order NeCoMan customizes its local reconfiguration algorithm for isolated services. Note that the decision symbols represent checking the service characteristics and reconfiguration semantics that must be fulfilled to safely apply a specific customization. These service characteristics and reconfiguration semantics are listed in Table F.1.

	service characteristics
A	the affected components belong to a distributed network service
B	the affected components encapsulate an isolated network service
C	the old components are stateless
D	the new components restore from or tolerate inconsistent execution states
E	the new components are able to continue processing all ongoing protocol-transactions
F	the new client processes do not employ active objects
G	the new server processes do not employ active objects
H	the processes of the new isolated component do not employ active objects
I	the old and new components encapsulate only client processes
J	the old and new components encapsulate only server processes
K	the old and new components encapsulate both client and server processes
L	the old and new components communicate by a unidirectional communication protocol
	reconfiguration semantics
M	the network tolerates packet re-ordering
N	the affected components operate in a best-effort network
O	the network restores from or tolerates inconsistent execution states
P	service addition
Q	service replacement
R	service removal

Table F.1: The service characteristics and reconfiguration semantics that NeCoMan uses to identify which customizations it can apply to its local reconfiguration algorithms. These properties result from the network administrator answering the questions listed in Tables 4.5 and 4.6.

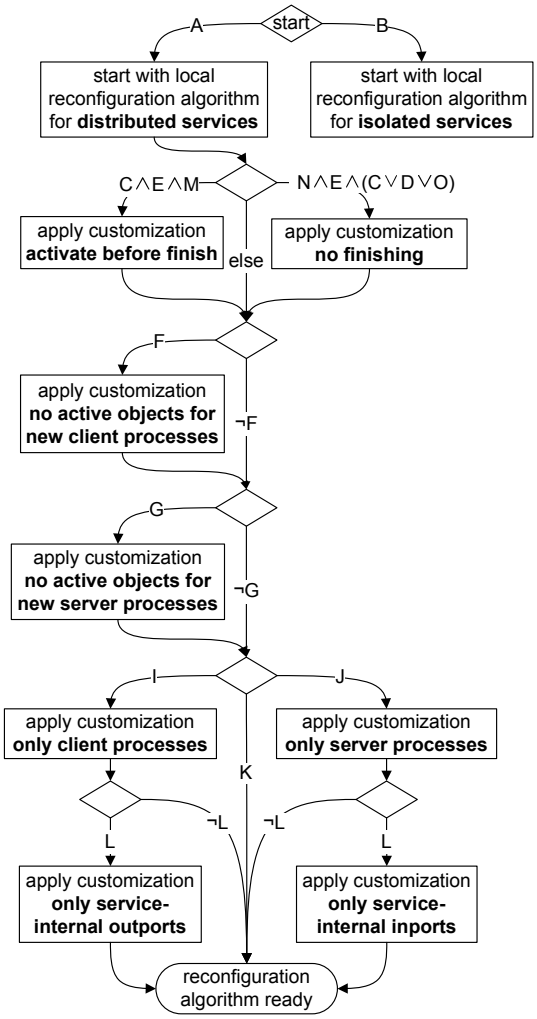


Figure F.1: Procedure to customize the first local reconfiguration algorithm based on the service characteristics and reconfiguration semantics listed in Table F.1.

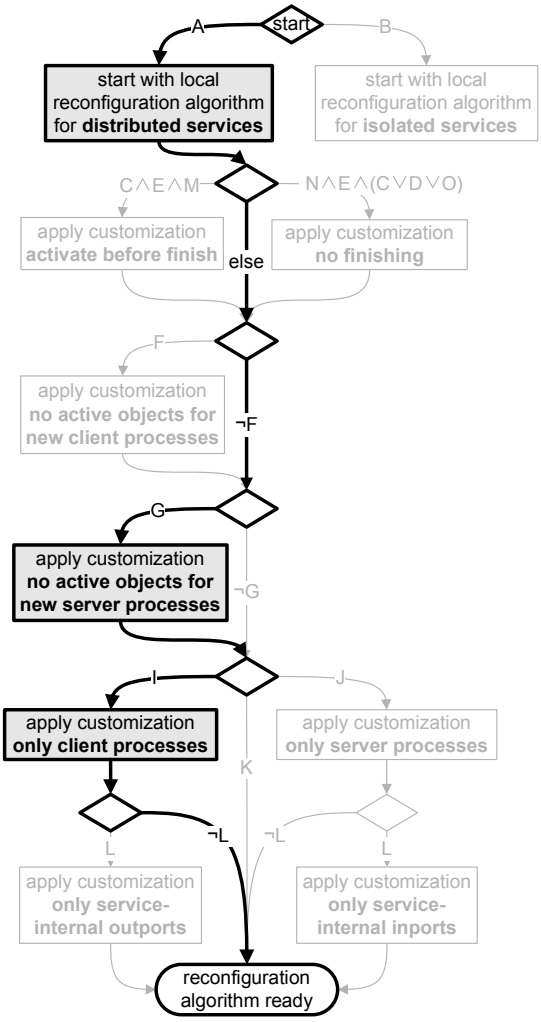


Figure F.2: Customization procedure when replacing R_{old} with R_{new}.

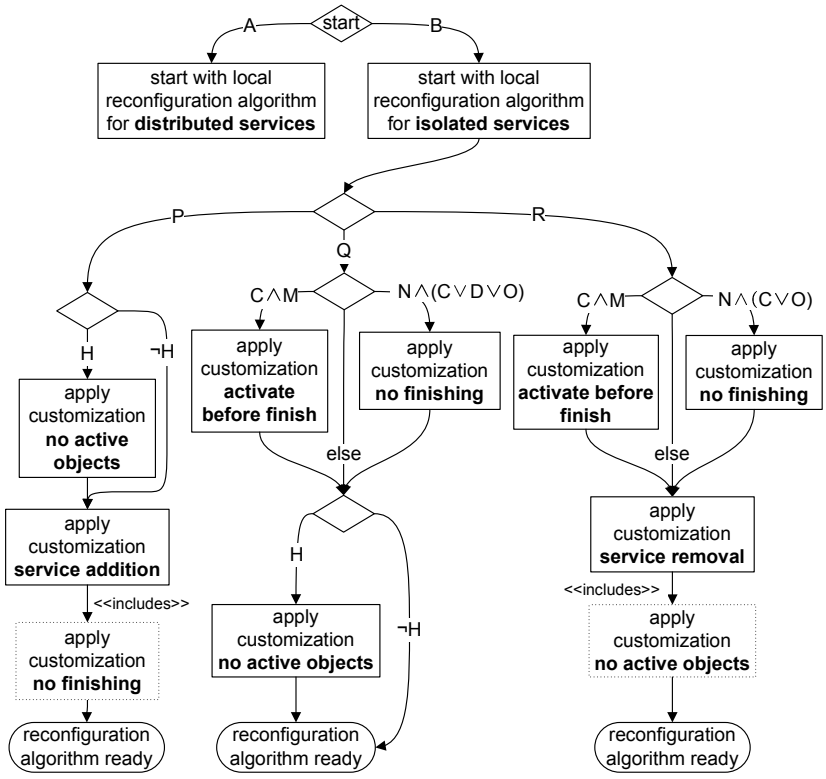


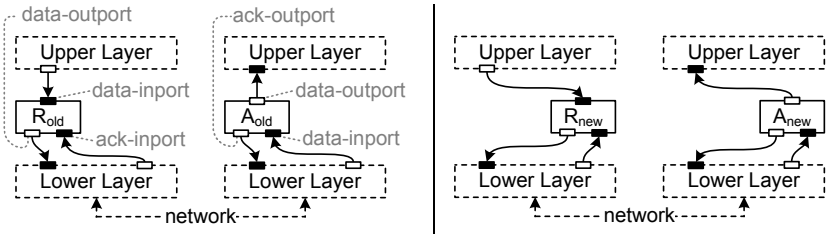
Figure F.3: Procedure to customize the second local reconfiguration algorithm based on the service characteristics and reconfiguration semantics listed in Table F.1.

Appendix G

Replacement of reliability service

In this appendix, we exemplify NeCoMan's algorithm for synchronized distributed reconfiguration with the dynamic replacement of a reliability service by a new version. Figure G.1 sketches the protocol stack composition of two nodes hosting the original reliability service (Figure G.1(a)), and the composition of these nodes after completing the replacement (Figure G.1(b)). R_{old} and R_{new} again represent the old and new retransmission components, while A_{old} and A_{new} correspond to the old and new acknowledgement components.

The Petri net in Figure G.2 models how NeCoMan coordinates the replacement of this reliability service. This model originates from NeCoMan's synchronized distributed reconfiguration algorithm, and is tailored to the characteristics of the reliability service. As we further explain in the remainder of this section, this involves discarding a number of transitions that become redundant when replacing



(a) Original composition (b) Replacement completed

Figure G.1: Synchronized distributed reconfiguration: protocol stack composition of two nodes hosting a reliability service, both before and after conducting the reconfiguration scenario.

a reliability service.

G.1 Installing new reliability service

The replacement begins by installing R_{new} and A_{new} on the sending and receiving node, respectively. NeCoMan first instructs these nodes' reconfiguration support to load R_{new} and A_{new}. Next, NeCoMan invokes both nodes to connect R_{new} and A_{new} into their protocol stack composition. Because the new reliability service should not be activated in this phase, the latter is limited to binding R_{new}'s data-outport, A_{new}'s data-outport, and A_{new}'s ack-outport. We illustrate the composition of both nodes after installing the new components in Figure G.3.

Besides, note that because R_{new} provides only service-internal outport, the execution of LO^{ext}_{new} on the sending node becomes redundant and can be discarded (as illustrated in Figure G.2). Component A_{new}, in contrast, exposes service-external as well as service-internal outports. Hence, on the receiving node both LO^{ext}_{new} and LO^{int}_{new} must be executed.

G.2 Finishing old reliability service

Next, the old reliability service becomes finished. This involves driving only R_{old} to a quiescent state. Once this is accomplished, A_{old} will have reached a reconfiguration-safe state as well without additional intervention. To demonstrate this, notice that

- only packets directed to the old retransmission process (being the client process of the old reliability service) must be intercepted for both R_{old} and A_{old} to reach a safe state, and that

- both the old retransmission and acknowledgment process are in a mutually consistent execution state once R_{old}'s retransmission queue is empty, which indicates that all transmitted data packets have arrived correctly.

So, to finish the old reliability service, NeCoMan first instructs the sending node to intercept packets that invoke R_{old}'s retransmission process. As a result, this node's reconfiguration support holds up all packets that are directed to R_{old}'s data-inport (as illustrated in Figure G.4). Once this is completed, NeCoMan instructs the sending node to impose a safe state over R_{old}'s retransmission process. Similar to a local reconfiguration, the associated reconfiguration support then monitors R_{old}'s retransmission queue until it is empty – that is, to observe when all ongoing protocol-transactions complete. Next, R_{old}'s retransmission timer is stopped as well. After that, both R_{old} and A_{old} are both quiescent and thus can safely be removed.

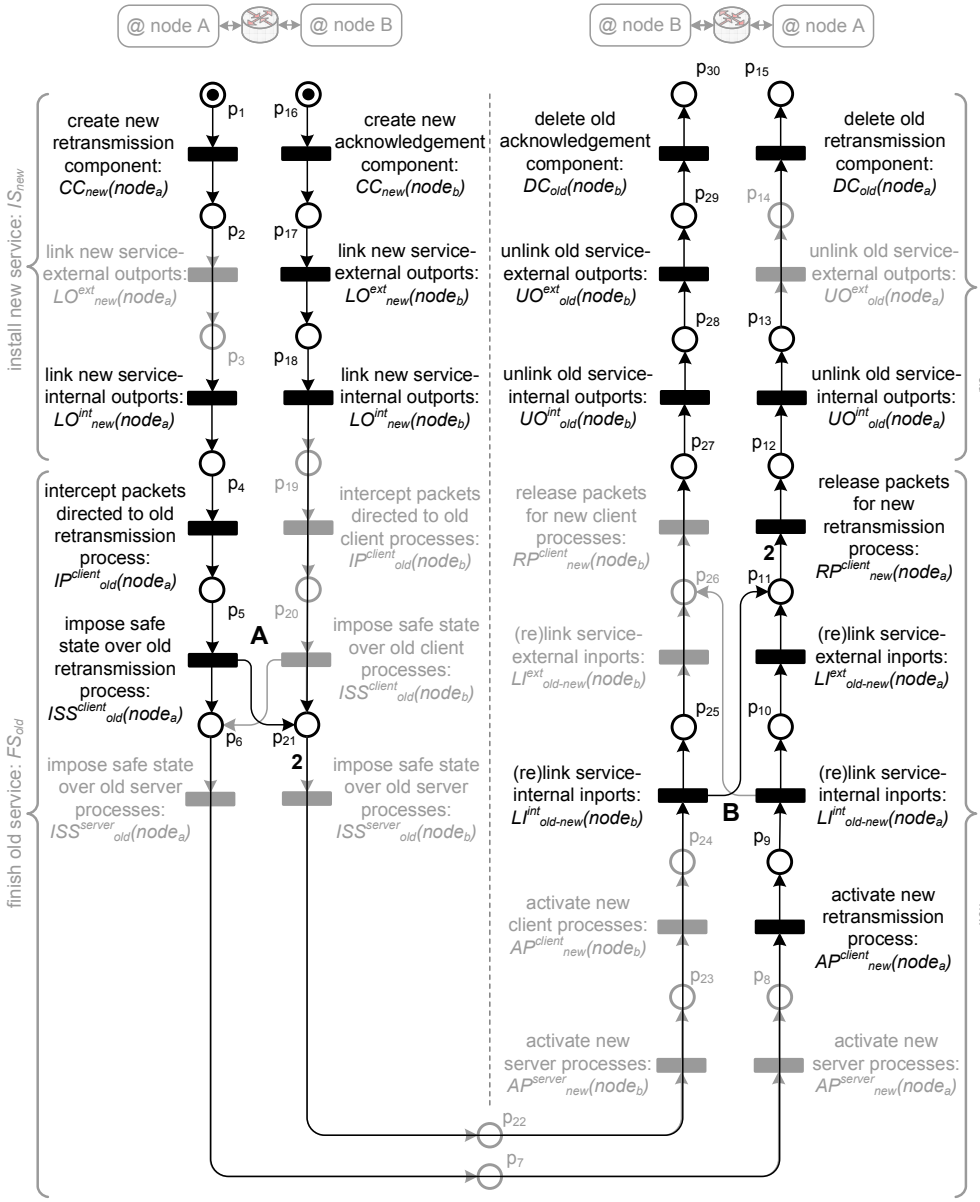


Figure G.2: Implementation of the distributed reconfiguration algorithm that NeCoMan uses to replace a reliability service

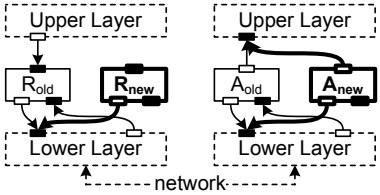


Figure G.3: Installation of new reliability components

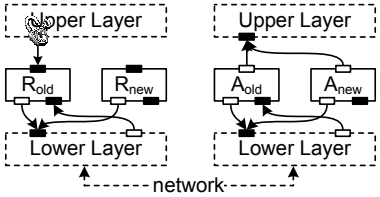


Figure G.4: Finishing old reliability components

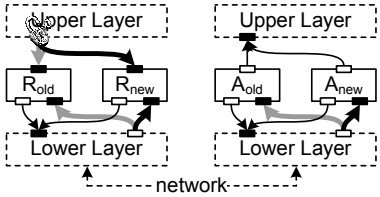


Figure G.5: Binding inports of new reliability components

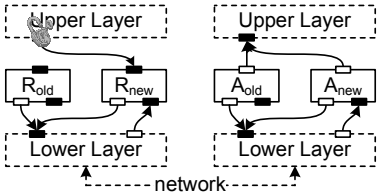


Figure G.6: Releasing intercepted packets

Note that because R_{old} encapsulates only a client process, reconfigurations actions IP_{old}^{server} and ISS_{old}^{server} become redundant and can be discarded. Besides, because A_{old} is in a reconfiguration-safe state once R_{old} is quiescent, on the receiving node no finishing operations must be executed either (as illustrated in Figure G.2)

G.3 Activating new reliability service

As soon as R_{old} is finished, NeCoMan activates the new reliability service. This involves instructing the affected nodes to bind the inports of R_{new} and A_{new}, to start R_{new}'s timer, and to resume all intercepted packets

Binding the inports of R_{new} and A_{new} is similar to a local reconfiguration. For the sending node, this involves first disconnecting R_{old}'s ack-inport and data-inport from the outports of the upper and lower layer, and then reconnecting these outports to the ack-inport and data-inport of R_{new}. On the receiving node, the outport of the lower layer becomes disconnected from A_{old}'s data-inport, and reconnected to the data-inport of A_{new}. This is illustrated in Figure G.5.

Once the inport of A_{new} is bound, NeCoMan sends a synchronization message from the receiving node towards the sending node. Once (1) this message has arrived, (2) the inports of R_{new} are bound, and (3) R_{new}'s retransmission timer is started, then NeCoMan instructs the sending node to release all intercepted packets. The latter is illustrated in Figure G.6. At this point in the reconfiguration scenario, the new reliability service processes all packets in transit.

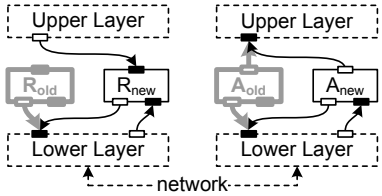


Figure G.7: Removal of old reliability components

Again some actions of NeCoMan's synchronized distributed reconfiguration algorithm can be discarded when replacing the reliability service. On the sending node, for instance, the execution of AP_{new}^{server} is redundant and thus must not be executed. On the receiving node, activating A_{new} involves only binding this component's inports. Hence, all other reconfiguration actions to activate a new service component can be discarded as well (as illustrated in Figure G.2).

G.4 Removing old reliability service

Finally, NeCoMan removes R_{old} and A_{old} from both nodes (as illustrated in Figure G.7). This involves first disconnecting R_{old} and A_{old} from other stack components by unlinking their outports. Next, NeCoMan instructs both nodes to delete R_{old} and A_{old}. Since the old reliability service has already been finished, the removal of these components will not compromise the correct network operation. Besides, note that because R_{old} provides only service-internal outport, the execution of UO_{new}^{ext} on the sending node becomes redundant and can be discarded (as illustrated in Figure G.2).

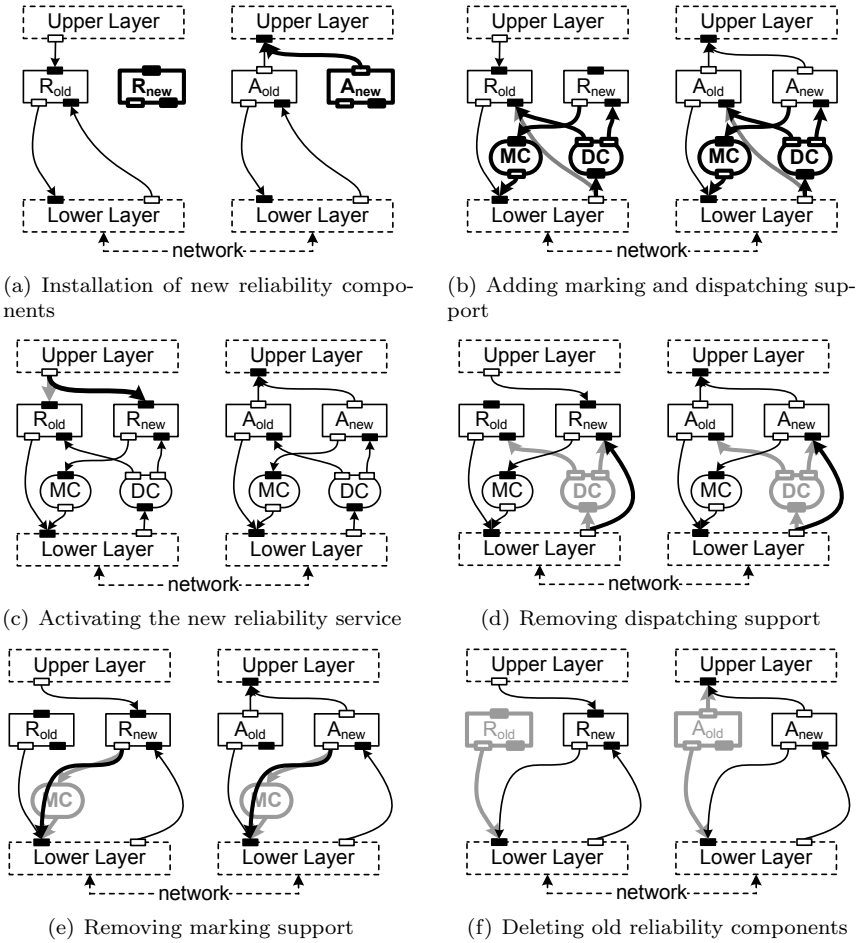
Appendix H

Additional distributed reconfigurations that involve the reliability service

In addition to Appendix G, we sketch a number of distributed reconfigurations that involve the reliability service. A first reconfiguration includes replacing the reliability service, in which the new service becomes activated before driving the old one to a quiescent execution state. Figure H.1 sketches this reconfiguration scenario. Besides, Figure H.2 depicts the Petri net model of the customized algorithm that NeCoMan uses to accomplish this replacement.

After that, we illustrate how NeCoMan adds a reliability service to two (operating) programmable nodes. Figure H.3 first sketches this reconfiguration scenario when quiescence must be reached before activating the new reliability service. In addition, Figure H.4 depicts the customized algorithm that NeCoMan uses to execute this dynamic addition. Next, Figure H.5 sketches all stages that NeCoMan passes through when the new reliability service can be activated before quiescence is reached. Furthermore, Figure H.6 depicts the Petri net model of the customized algorithm that NeCoMan uses to accomplish this addition.

Finally, we also illustrate how NeCoMan dynamically removes a reliability service from two (operating) programmable nodes. We first focus on the scenario in which the old reliability service becomes finished before being bypassed. This scenario and the algorithm that NeCoMan uses to carry out this reconfiguration are illustrated in Figures H.7 and H.8, respectively. After that, Figure H.9 sketches all stages that NeCoMan passes through when the old service becomes bypassed before quiescence is reached. The customized algorithm that NeCoMan uses to conduct this removal is depicted in Figure H.10.



(a) Installation of new reliability components

(b) Adding marking and dispatching support

(c) Activating the new reliability service

(d) Removing dispatching support

(e) Removing marking support

(f) Deleting old reliability components

Figure H.1: Replacement of old reliability service with new one: scenario which involves activation before finishing

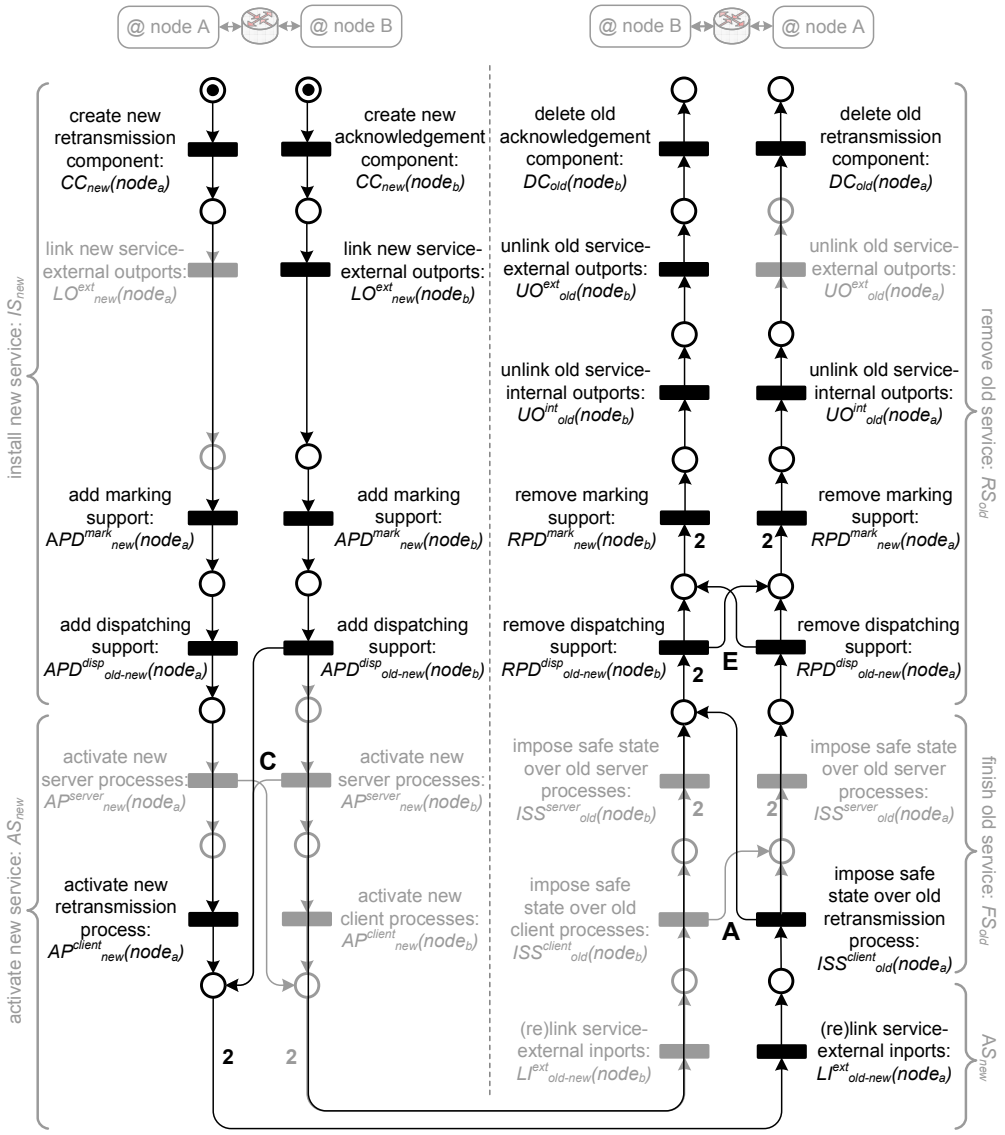
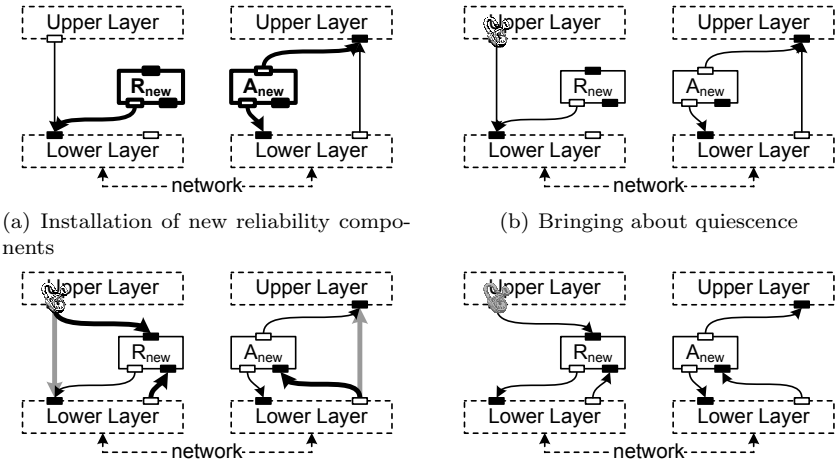


Figure H.2: Replacement of old reliability service with new one: customized reconfiguration algorithm which involves activation before finishing



(a) Installation of new reliability components

(b) Bringing about quiescence

(c) Binding inports

(d) Releasing intercepted packets

Figure H.3: Addition of new reliability service: scenario which involves reaching quiescence before activation

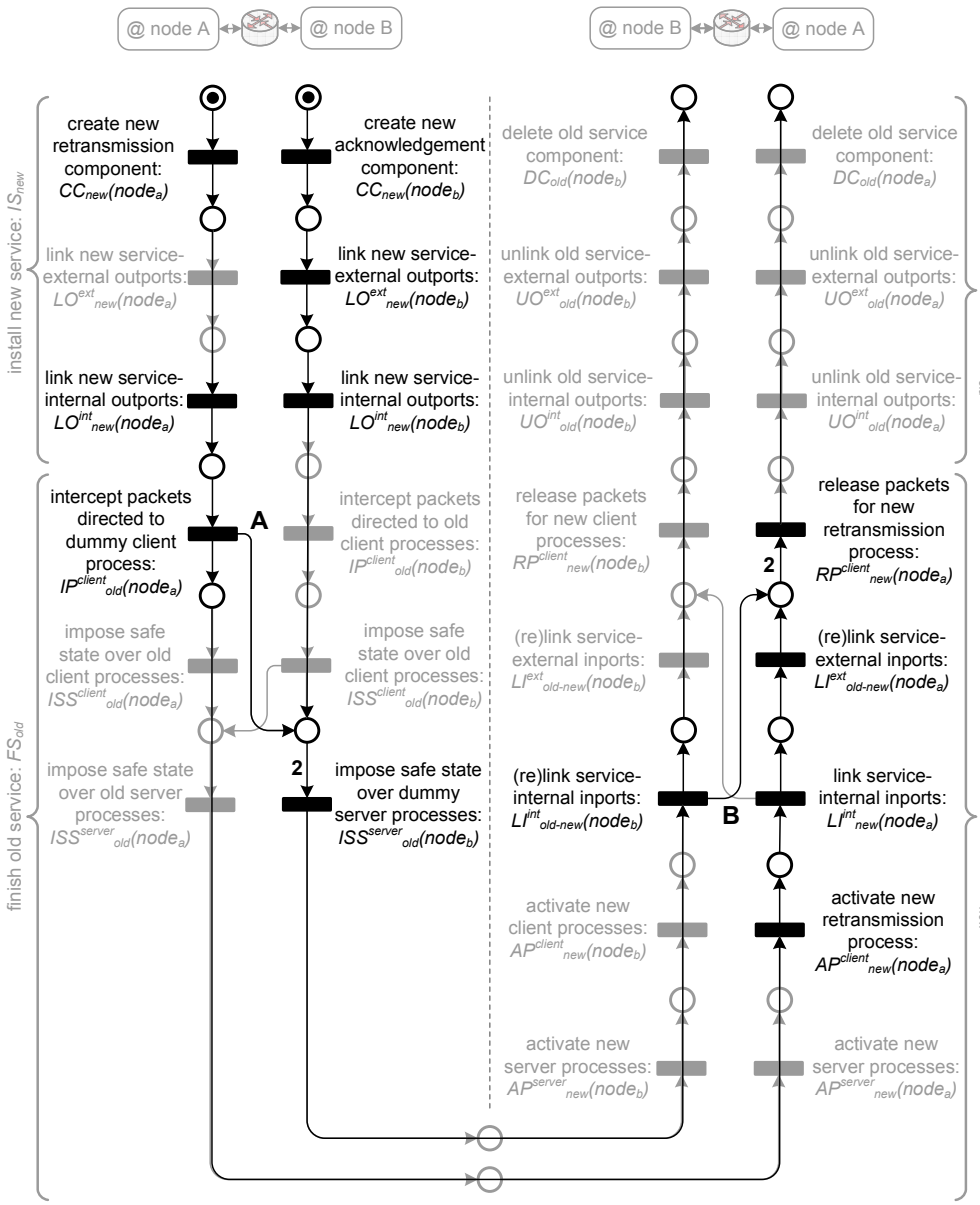
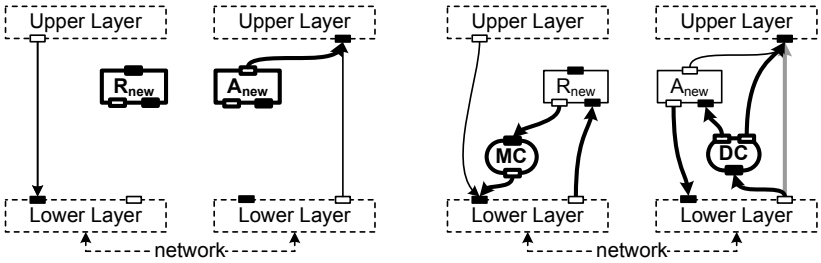
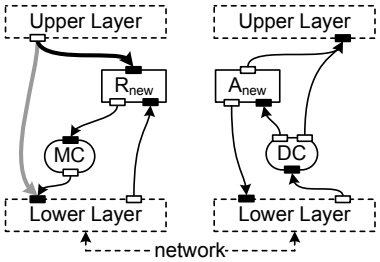


Figure H.4: Addition of new reliability service: customized reconfiguration algorithm which involves reaching quiescence before activation

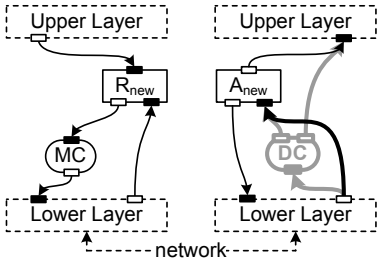


(a) Installation of new reliability components

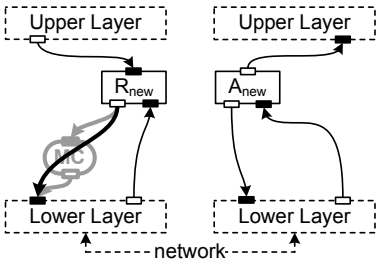
(b) Adding marking and dispatching support



(c) Activating the new reliability service



(d) Removing dispatching support



(e) Removing marking support

Figure H.5: Addition of new reliability service: scenario which involves activation before reaching quiescence

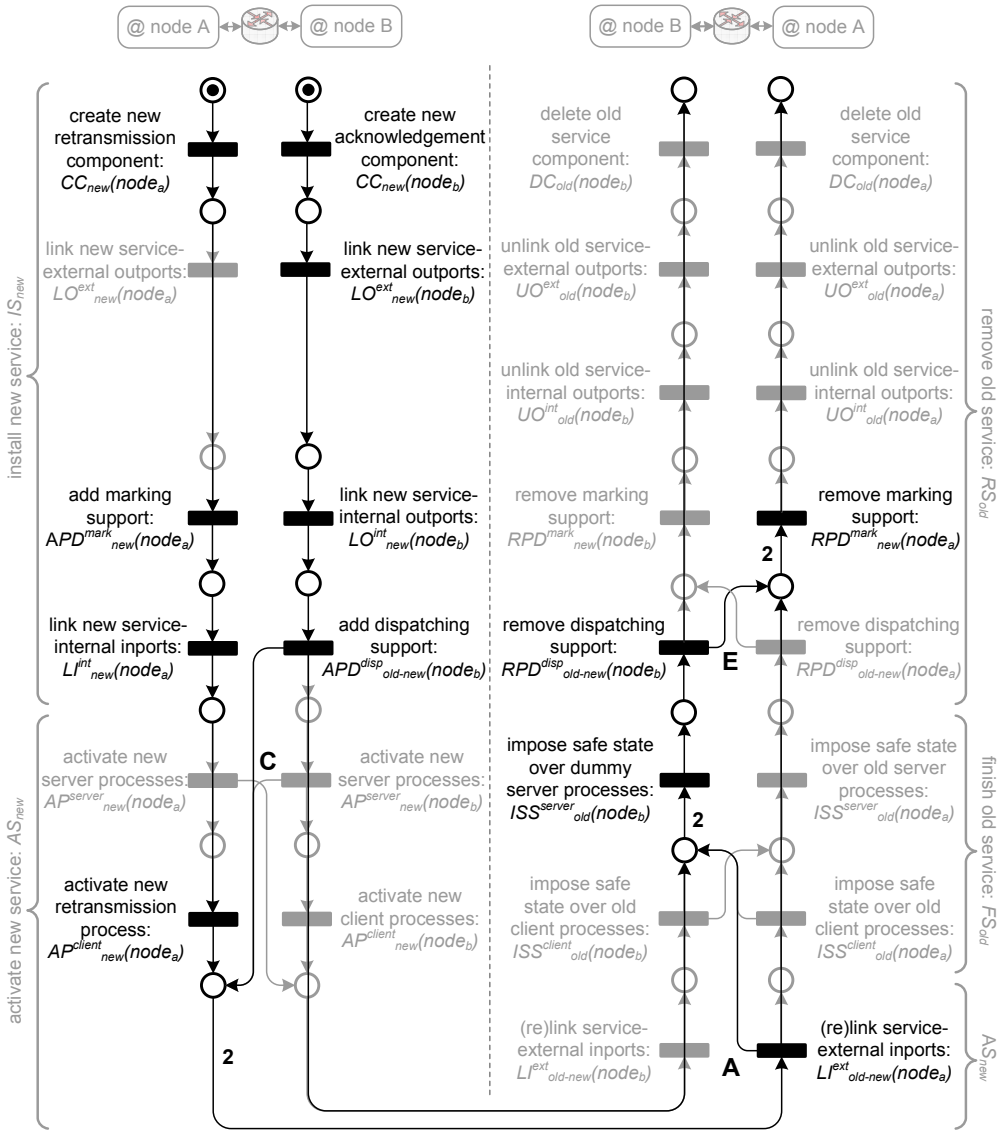
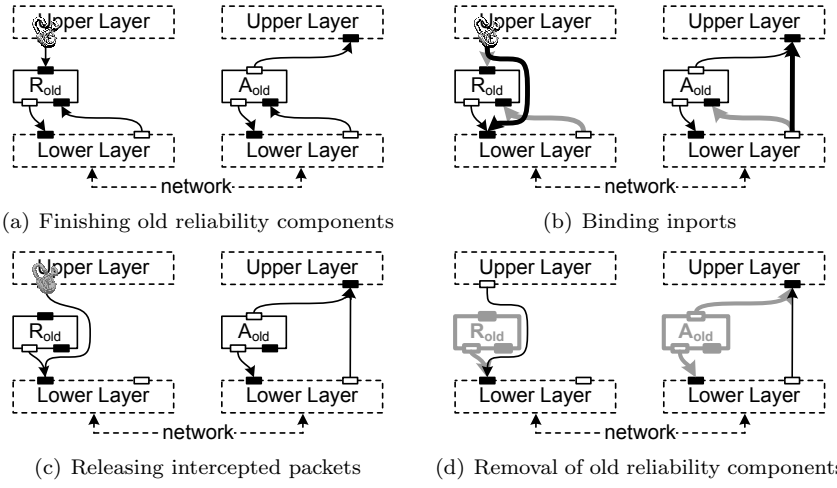


Figure H.6: Addition of new reliability service: customized reconfiguration algorithm which involves activation before reaching quiescence



(a) Finishing old reliability components

(b) Binding inports

(c) Releasing intercepted packets

(d) Removal of old reliability components

Figure H.7: Removal of old reliability service: scenario which involves finishing before activation

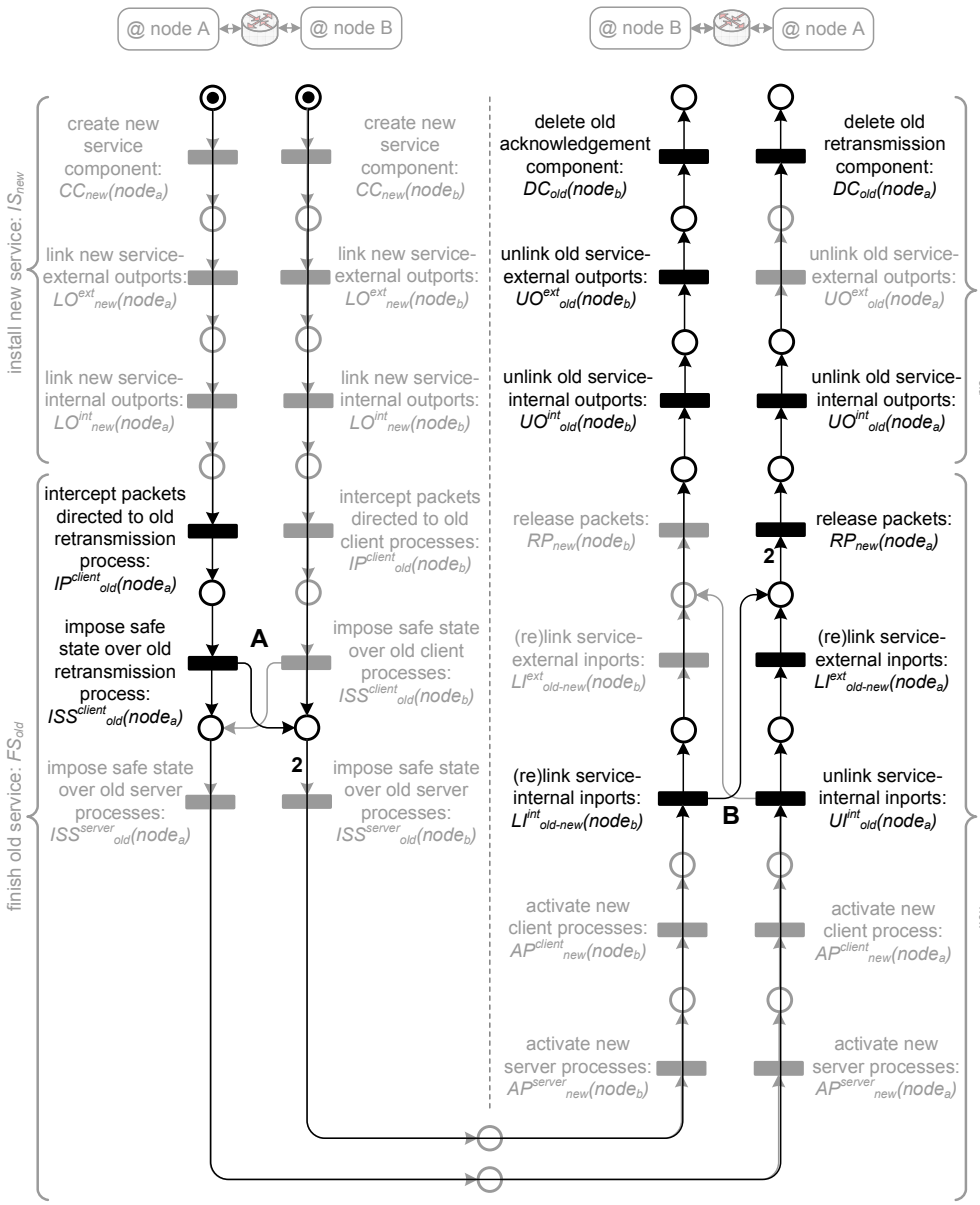
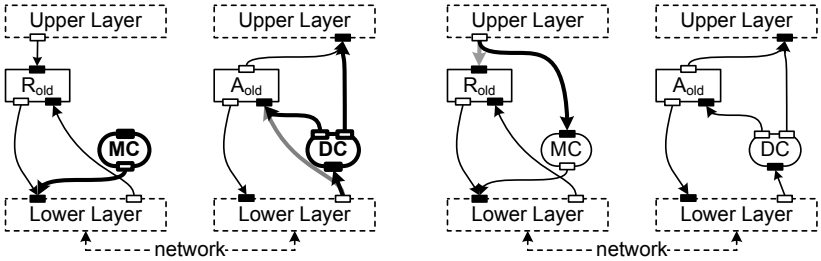
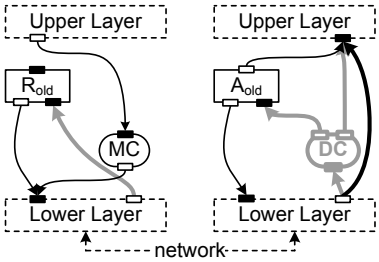


Figure H.8: Removal of old reliability service: customized reconfiguration algorithm which involves reaching quiescence before activation

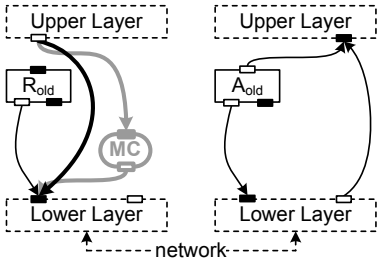


(a) Adding marking and dispatching support

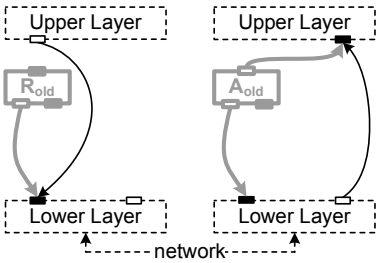
(b) Activating the new reliability service



(c) Removing dispatching support



(d) Removing marking support



(e) Deleting old reliability components

Figure H.9: Removal of old reliability service: activation before finishing

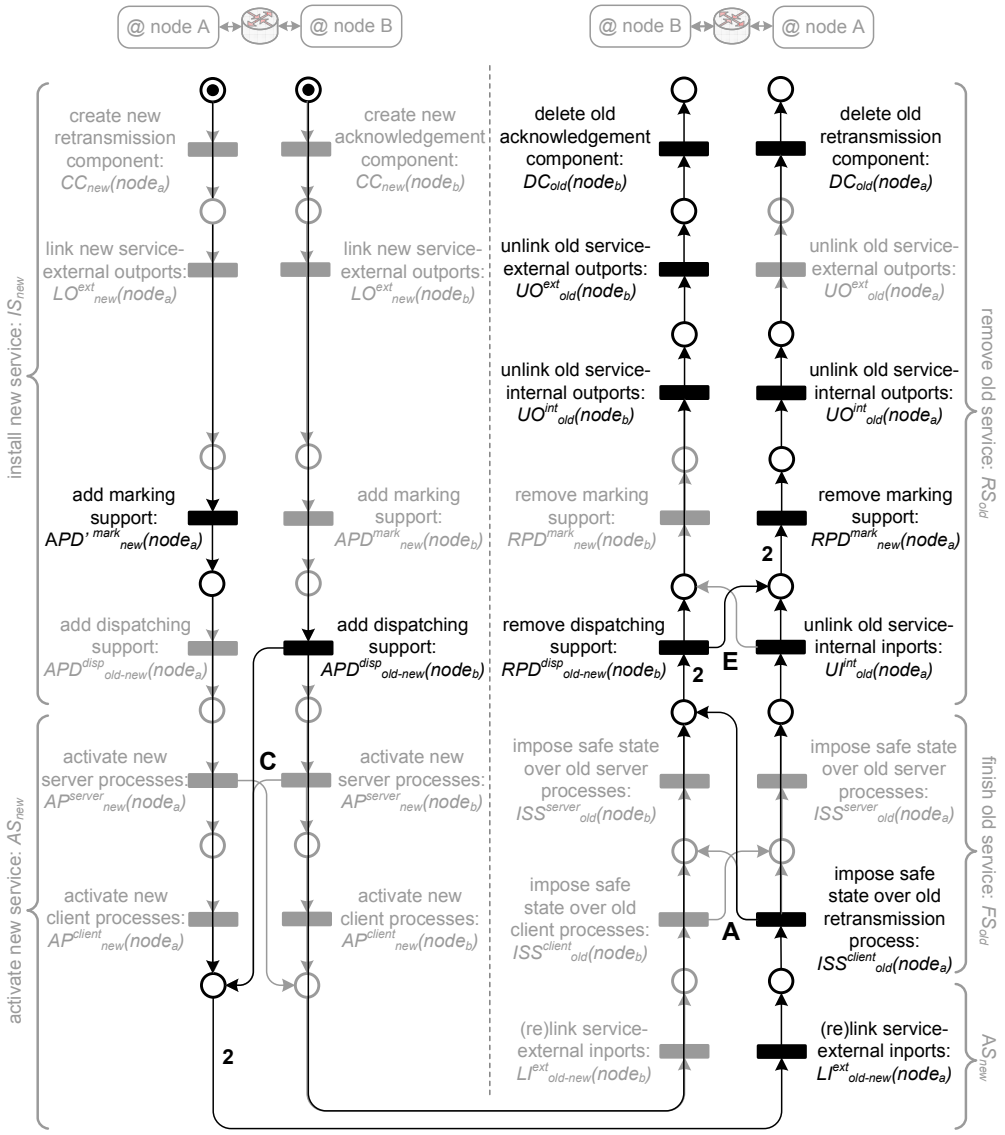


Figure H.10: Removal of old reliability service: customized reconfiguration algorithm which involves activation before reaching quiescence

Appendix I

Correctness of NeCoMan's algorithm for conducting synchronized distributed reconfigurations

This appendix demonstrates that NeCoMan's algorithm for conducting synchronized distributed reconfigurations meets all required reconfiguration conditions. These conditions are listed in Table 5.1. Related to the algorithm's Petri net model, each one of these reconfiguration condition formalizes a pre-condition to fire a transition. Therefore, to demonstrate that the presented algorithm conducts correct reconfigurations, we check for every transition modelled in Figure 5.13 if all pre-conditions (that the associated reconfiguration conditions impose) are met.

1. **Create new service component**. No pre-conditions must be fulfilled to safely execute $CC_{new}(node_x)$ (see Table 5.1). Consequently, there is no need to coordinate firing the transition that models the execution of CC_{new}.

2. **Link new service-external and service-internal outports**. As reconfiguration condition (5.1) dictates, the execution of LO^{ext}_{new} and LO^{int}_{new} can only be started on node x when $CC_{new}(node_x)$ has completed. We conclude from Table 5.2 that this is fulfilled as from reaching places p_2 and p_{17}. These places are the input places of $LO^{ext}_{new}(node_a)$ and $LO^{ext}_{new}(node_b)$, as well as the ancestor places of $LO^{int}_{new}(node_a)$ and $LO^{int}_{new}(node_b)$. The algorithm thus meets reconfiguration condition (5.1).

3. **Intercept packets directed to old client processes**. Similar as for the execution of $CC_{new}(node_x)$, we conclude from Table 5.1 that no pre-condition

must be fulfilled before executing $IP_{old}^{client}(node_x)$.

4. **Impose safe state over old client processes**. According to reconfiguration condition (5.2), NeCoMan can only instruct a node x to execute $ISS_{old}^{client}(node_x)$ once $IP_{old}^{client}(node_x)$ has completed. The latter is met as from reaching places p_5 and p_{20}, which are the input places of $ISS_{old}^{client}(node_a)$ and $ISS_{old}^{client}(node_b)$, respectively. So, the algorithm satisfies reconfiguration condition (5.2).

5. **Impose safe state over old server processes**. Reconfiguration condition (5.3) imposes to only initiate the execution of $ISS_{old}^{server}(node_x)$ once $ISS_{old}^{client}(node_y)$ is completed on every node $y \neq x$. We conclude from Table 5.2 that this condition is fulfilled as from reaching places p_6 and p_{21} – that is, after receiving synchronization message **A**. Because p_6 and p_{21} are the input places of $ISS_{old}^{server}(node_a)$ and $ISS_{old}^{server}(node_b)$, respectively, the algorithm meets reconfiguration condition (5.3).

6. **Activate new server processes**. According to reconfiguration conditions (5.7) and (5.12), NeCoMan can only execute AP_{new}^{server} on node x once CC_{new} and ISS_{old}^{server} have completed on that node x. Table 5.2 indicates that this pre-condition is met as from reaching places p_7 and p_{22}, which are the input places of $AP_{new}^{server}(node_a)$ and $AP_{new}^{server}(node_b)$, respectively.

7. **Activate new client processes**. As defined by reconfiguration conditions (5.7) and (5.11), the execution of AP_{new}^{client} can only be initiated on node x when $CC_{new}(node_x)$ and $ISS_{old}^{client}(node_x)$ have completed. This condition is met as from reaching places p_6 and p_{21}. Because these are the ancestor places of $AP_{new}^{client}(node_a)$ and $AP_{new}^{client}(node_b)$, the algorithm meets this pre-condition as well.

8. **(Re)link service-internal inports**. The execution of $LI_{old-new}^{int}(node_x)$ can only be started once CC_{new}, ISS_{old}^{client}, and ISS_{old}^{server} are completed on node x, as imposed by reconfiguration conditions (5.6) an (5.9). This is accomplished as from reaching places p_7 and p_{22}, which are the ancestor places of $LI_{old-new}^{int}(node_a)$ and $LI_{old-new}^{int}(node_b)$, respectively. Hence, the algorithm does not violate these reconfiguration conditions.

9. **(Re)link service-external inports**. Reconfiguration conditions (5.6) and (5.10) dictate that $LI_{old-new}^{ext}(node_x)$ can only be executed when $CC_{new}(node_x)$ and $ISS_{old}^{client}(node_x)$ are completed. This pre-condition is met as from reaching places p_6 and p_{21}. Since p_6 and p_{21} are ancestor places of the transitions that model $LI_{old-new}^{ext}(node_a)$ and $LI_{old-new}^{ext}(node_b)$, respectively, the algorithm does not violate reconfiguration conditions (5.6) and (5.10) either.

10. **Release packets for new client processes**. According to reconfiguration conditions (5.4) and (5.8), NeCoMan can only safely initiate the execution of RP_{new}^{client} on node x when

    (a) LO_{new}^{int}, $LI_{old-new}^{int}$, $LI_{old-new}^{ext}$, and AP_{new}^{client} are completed on that node x, and

    (b) LO_{new}^{int}, LO_{new}^{ext}, $LI_{old-new}^{int}$, and AP_{new}^{server} are executed as well on every other node $y \neq x$.

    Both conditions are met as from reaching places p_{11} and p_{26} – that is, after receiving synchronization message **B**. Because these places are the input places of $RP_{new}^{client}(node_a)$ and $RP_{new}^{client}(node_b)$, the algorithm fulfills reconfiguration conditions (5.4) and (5.8) as well.

11. **Unlink old service-internal outports**. As reconfiguration condition (5.13) defines, the execution of UO_{old}^{int} can only be started on node x when both $ISS_{old}^{client}(node_x)$ and $ISS_{old}^{server}(node_x)$ are completed. This pre-condition is met as from reaching places p_7 and p_{22}. Because p_7 and p_{22} are ancestor places of the transitions that model $UO_{old}^{int}(node_a)$ and $UO_{old}^{int}(node_b)$, reconfiguration condition (5.13) is fulfilled as well.

12. **Unlink old service-external outports**. Reconfiguration condition (5.14) imposes to only execute $UO_{old}^{ext}(node_x)$ once $ISS_{old}^{server}(node_x)$ has completed. The latter is accomplished as from reaching places p_7 and p_{22}. Since both places are ancestor places of the transitions that model $UO_{old}^{ext}(node_a)$ and $UO_{old}^{ext}(node_b)$, the algorithm does not violate this condition either.

13. **Delete old service component**. Finally, reconfiguration condition (5.5) dictates to only execute DC_{old} on node x once UO_{old}^{ext}, UO_{old}^{int}, $LI_{old-new}^{ext}$, and $LI_{old-new}^{int}$ are completed on that node. This pre-condition is fulfilled as from reaching places p_{14} and p_{29}. Because these places are the input places of the transitions that model $DC_{old}(node_a)$ and $DC_{old}(node_b)$, the algorithm fulfills reconfiguration condition (5.5) as well.

So, we conclude that the algorithm depicted in Figure 5.13 fulfills all required reconfiguration conditions, and therefore yields correct reconfigurations.

Appendix J

Affected reconfiguration conditions when customizing NeCoMan's algorithms for distributed reconfigurations

This appendix lists the reconfiguration conditions that are changed when applying customizations 6.5 to 6.9 to both basic distributed reconfiguration scenarios. Note that the effect of customizations 6.2 to 6.4 has already been discussed in detail in Chapter 6.

We first focus on customization 6.5. This customization involves omitting the execution of ISS_{old}^{server}. To illustrate the impact of this customization, Table J.1 lists all reconfiguration conditions that are changed when only a service's client processes must be finished to reach quiescence.

Next, we concentrate on customization 6.6, which involves the use of active objects. Tables J.2 and J.3 list all reconfiguration conditions related to NeCoMan's algorithm for synchronized distributed reconfiguration (as well as to the algorithms resulting from applying customizations 6.3 or 6.4 to this algorithm) that are changed when the new client and/or server processes do not employ active objects.

After that, we focus on customization 6.7. This customization involves omitting some reconfiguration actions when the old and new components encapsulate only client or server processes, instead of both. Table J.4 lists all reconfiguration conditions that are changed when the old and new component on node x encapsulate client processes only. Similarly, Table J.5 lists all reconfiguration conditions that are changed when the affected components encapsulate only server processes.

Next, we concentrate on customization 6.8. This customization is targeted at omitting reconfiguration actions that involve service-internal inports and outports.

affected reconfiguration condition	**resulting safety condition**
(5.3)	N/A
(5.9)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow \forall node_y : ISS^{client}_{old}(node_y)]$
(5.12)	N/A
(5.13)	$\forall node_x : [UO^{int}_{old}(node_x) \leftarrow \forall node_y : ISS^{client}_{old}(node_y)]$
(5.14)	$\forall node_x : [UO^{ext}_{old}(node_x) \leftarrow \forall node_y \neq node_x : ISS^{client}_{old}(node_y)]$
(6.7)	$\forall node_x : [RPD^{mark}_{new}(node_x) \wedge RPD^{disp}_{old-new}(node_x) \leftarrow \forall node_y : ISS^{client}_{old}(node_y)]$

Table J.1: Customization of NeCoMan's algorithm for synchronized distributed reconfiguration: overview of all reconfiguration conditions that become adapted when the old server processes do not have to be finished.

To illustrate the effect of this customization on synchronized distributed reconfigurations, Tables J.6 to J.11 list all reconfiguration conditions that this customization affects. In addition, Tables J.12 to J.15 list all reconfiguration conditions that NeCoMan must fulfill when replacing a unidirectional service with a bidirectional one, and vice versa, without finishing the old service.

Finally, we focus on customization 6.9, which seeks to tailor a reconfiguration to add or remove distributed network services instead of replacing them. Tables J.16 and J.17 list all reconfiguration conditions (related to the algorithm for synchronized distributed reconfiguration and the algorithms that result from applying customizations 6.3 or 6.4) that are changed in case of service addition and removal, respectively. Besides, Tables J.18 and J.19 list all reconfiguration conditions that are changed when NeCoMan conducts isolated distributed reconfigurations instead.

affected reconfiguration condition	**resulting safety condition**
(5.4)	$\forall node_x : [RP^{client}_{new}(node_x) \leftarrow LI^{int}_{old-new}(node_x) \wedge LI^{ext}_{old-new}(node_x) \wedge \forall node_y \neq node_x : [LI^{int}_{old-new}(node_y) \wedge AP^{server}_{new}(node_y)]]$
(5.7)	$\forall node_x : [AP^{server}_{new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.11)	N/A
(6.2)	$\forall node_x : [LI^{ext}_{old-new}(node_x) \leftarrow \forall node_y \neq node_x : AP^{server}_{new}(node_y)]$
(6.8)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow LO^{ext}_{new}(node_x) \wedge LO^{int}_{new}(node_x) \wedge AP^{server}_{new}(node_x)]$

Table J.2: Customization of NeCoMan's algorithm for synchronized distributed reconfiguration: overview of all reconfiguration conditions that become changed when the new components' client processes do not use active objects.

affected reconfiguration condition	**resulting safety condition**
(5.4)	$\forall node_x : [RP^{client}_{new}(node_x) \leftarrow LI^{int}_{old-new}(node_x) \wedge LI^{ext}_{old-new}(node_x) \wedge AP^{client}_{new}(node_x) \wedge \forall node_y \neq node_x : LI^{int}_{old-new}(node_y)]$
(5.7)	$\forall node_x : [AP^{client}_{new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.12)	N/A
(6.2)	$\forall node_x : [LI^{ext}_{old-new}(node_x) \leftarrow AP^{client}_{new}(node_x)]$
(6.8)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow LO^{ext}_{new}(node_x) \wedge LO^{int}_{new}(node_x) \wedge AP^{client}_{new}(node_x)]$

Table J.3: Customization of NeCoMan's algorithm for synchronized distributed reconfiguration: overview of all reconfiguration conditions that become changed when the new components' server processes do not use active objects.

affected reconfiguration condition	**resulting safety condition**
(5.1)	$\forall node_x : [LO^{int}_{new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.3)	N/A
(5.5)	$\forall node_x : [DC_{old}(node_x) \leftarrow UO^{int}_{old}(node_x) \wedge LI^{ext}_{old-new}(node_x) \wedge LI^{int}_{old-new}(node_x)]$
(5.7)	$\forall node_x : [AP^{client}_{new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.9)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow ISS^{client}_{old}(node_x)]$
(5.12)	N/A
(5.13)	$\forall node_x : [UO^{int}_{old}(node_x) \leftarrow ISS^{client}_{old}(node_x)]$
(5.14)	N/A
(6.1)	$\forall node_x : [APD^{mark}_{new}(node_x) \wedge APD^{disp}_{old-new}(node_x) \leftarrow CC_{new}(node_x)]$
(6.4)	$\forall node_x : [DC_{old}(node_x) \leftarrow LI^{ext}_{old-new}(node_x) \wedge RPD^{disp}_{old-new}(node_x) \wedge UO^{int}_{old}(node_x)]$
(6.7)	$\forall node_x : [RPD^{mark}_{new}(node_x) \wedge RPD^{disp}_{old-new}(node_x) \leftarrow ISS^{client}_{old}(node_x)]$
(6.8)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow LO^{int}_{new}(node_x) \wedge AP^{client}_{new}(node_x)]$
(6.10)	N/A

Table J.4: Customization of NeCoMan's algorithm for synchronized distributed reconfiguration: overview of all reconfiguration conditions that become adapted when the old and new component on node x encapsulate client processes only.

affected reconfiguration condition	**resulting safety condition**
(5.2)	N/A
(5.4)	N/A
(5.5)	$\forall node_x : [DC_{old}(node_x) \leftarrow UO^{ext}_{old}(node_x) \wedge UO^{int}_{old}(node_x) \wedge LI^{int}_{old-new}(node_x)]$
(5.6)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.7)	$\forall node_x : [AP^{server}_{new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.8)	N/A
(5.9)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow ISS^{server}_{old}(node_x)]$
(5.10)	N/A
(5.11)	N/A
(5.13)	$\forall node_x : [UO^{int}_{old}(node_x) \leftarrow ISS^{server}_{old}(node_x)]$
(6.2)	N/A
(6.4)	$\forall node_x : [DC_{old}(node_x) \leftarrow UO^{ext}_{old}(node_x) \wedge RPD^{disp}_{old-new}(node_x) \wedge UO^{int}_{old}(node_x)]$
(6.5)	N/A
(6.6)	N/A
(6.7)	$\forall node_x : [RPD^{mark}_{new}(node_x) \wedge RPD^{disp}_{old-new}(node_x) \leftarrow ISS^{server}_{old}(node_x)]$
(6.8)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow LO^{ext}_{new}(node_x) \wedge LO^{int}_{new}(node_x) \wedge AP^{server}_{new}(node_x)]$
(6.9)	N/A
(6.11)	$\forall node_x : [UO^{int}_{old}(node_x) \leftarrow LI^{int}_{old-new}(node_x)]$

Table J.5: Customization of NeCoMan's algorithm for synchronized distributed reconfiguration: overview of all reconfiguration conditions that become adapted when the old and new component on node x encapsulate server processes only.

affected reconfiguration condition	**resulting safety condition**
(5.1)	$\forall node_x : [LO^{int}_{new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.3)	N/A
(5.4)	$\forall node_x : [RP^{client}_{new}(node_x) \leftarrow LI^{ext}_{old-new}(node_x) \wedge AP^{client}_{new}(node_x) \wedge \forall node_y \neq node_x : [LI^{int}_{old-new}(node_y) \wedge AP^{server}_{new}(node_y)]]$
(5.5)	$\forall node_x : [DC_{old}(node_x) \leftarrow UO^{int}_{old}(node_x) \wedge LI^{ext}_{old-new}(node_x)]$
(5.6)	$\forall node_x : [LI^{ext}_{old-new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.7)	$\forall node_x : [AP^{client}_{new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.8)	$\forall node_x : [RP^{client}_{new}(node_x) \leftarrow LO^{int}_{new}(node_x) \wedge \forall node_y \neq node_x : LO^{ext}_{new}(node_y)]$
(5.9)	N/A
(5.12)	N/A
(5.13)	$\forall node_x : [UO^{int}_{old}(node_x) \leftarrow ISS^{client}_{old}(node_x)]$
(5.14)	N/A
(6.1)	$\forall node_x : [APD^{mark}_{new}(node_x) \leftarrow CC_{new}(node_x)]$
(6.4)	$\forall node_x : [DC_{old}(node_x) \leftarrow LI^{ext}_{old-new}(node_x) \wedge UO^{int}_{old}(node_x)]$
(6.5)	$\forall node_x : [LI^{ext}_{old-new}(node_x) \leftarrow APD^{mark}_{new}(node_x) \wedge \forall node_y \neq node_x : [LO^{ext}_{new}(node_y) \wedge APD^{disp}_{old-new}(node_y)]]$
(6.7)	$\forall node_x : [RPD^{mark}_{new}(node_x) \leftarrow ISS^{client}_{old}(node_x)]$
(6.8)	N/A
(6.9)	$\forall node_x : [LI^{ext}_{old-new}(node_x) \leftarrow LO^{int}_{new}(node_x) \wedge AP^{client}_{new}(node_x) \wedge \forall node_y \neq node_x : LI^{int}_{old-new}(node_y)]$
(6.10)	N/A
(6.11)	$\forall node_x : [UO^{int}_{old}(node_x) \leftarrow LI^{ext}_{old-new}(node_x)]$

Table J.6: Customization of NeCoMan's algorithm for synchronized distributed reconfiguration: overview of all reconfiguration conditions that become adapted when the **old and new** components on node x encapsulate only client processes, which employ unidirectional communication protocols.

affected reconfig-uration condition	**resulting safety condition**
(5.1)	$\forall node_x : [LO^{ext}_{new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.2)	N/A
(5.4)	N/A
(5.5)	$\forall node_x : [DC_{old}(node_x) \leftarrow UO^{ext}_{old}(node_x) \wedge LI^{int}_{old-new}(node_x)]$
(5.6)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.7)	$\forall node_x : [AP^{server}_{new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.8)	N/A
(5.9)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow ISS^{server}_{old}(node_x)]$
(5.10)	N/A
(5.11)	N/A
(5.13)	N/A
(6.1)	$\forall node_x : [LO^{ext}_{new}(node_x) \wedge APD^{disp}_{old-new}(node_x) \leftarrow CC_{new}(node_x)]$
(6.2)	N/A
(6.3)	N/A
(6.4)	$\forall node_x : [DC_{old}(node_x) \leftarrow UO^{ext}_{old}(node_x) \wedge RPD^{disp}_{old-new}(node_x)]$
(6.5)	N/A
(6.6)	N/A
(6.7)	$\forall node_x : [RPD^{disp}_{old-new}(node_x) \leftarrow ISS^{server}_{old}(node_x)]$
(6.8)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow LO^{ext}_{new}(node_x) \wedge AP^{server}_{new}(node_x)]$
(6.9)	N/A
(6.11)	N/A

Table J.7: Customization of NeCoMan's algorithm for synchronized distributed reconfiguration: overview of all reconfiguration conditions that become adapted when the **old and new** components on node x encapsulate only server processes, which employ unidirectional communication protocols.

affected reconfiguration condition	**resulting safety condition**
(5.1)	$\forall node_x : [LO^{int}_{new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.3)	N/A
(5.4)	$\forall node_x : [RP^{client}_{new}(node_x) \leftarrow LI^{ext}_{old-new}(node_x) \wedge AP^{client}_{new}(node_x) \wedge \forall node_y \neq node_x : [LI^{int}_{old-new}(node_y) \wedge AP^{server}_{new}(node_y)]]$
(5.5)	$\forall node_x : [DC_{old}(node_x) \leftarrow UO^{int}_{old}(node_x) \wedge LI^{ext}_{old-new}(node_x) \wedge UI^{int}_{old}(node_x)]$
(5.6)	$\forall node_x : [LI^{ext}_{old-new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.7)	$\forall node_x : [AP^{client}_{new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.8)	$\forall node_x : [RP^{client}_{new}(node_x) \leftarrow LO^{int}_{new}(node_x) \wedge \forall node_y \neq node_x : LO^{ext}_{new}(node_y)]$
(5.9)	$\forall node_x : [UI^{int}_{old}(node_x) \leftarrow ISS^{client}_{old}(node_x)]$
(5.12)	N/A
(5.13)	$\forall node_x : [UO^{int}_{old}(node_x) \leftarrow ISS^{client}_{old}(node_x)]$
(5.14)	N/A
(6.1)	$\forall node_x : [APD^{mark}_{new}(node_x) \leftarrow CC_{new}(node_x)]$
(6.4)	$\forall node_x : [DC_{old}(node_x) \leftarrow LI^{ext}_{old-new}(node_x) \wedge UI^{int}_{old}(node_x) \wedge UO^{int}_{old}(node_x)]$
(6.5)	$\forall node_x : [LI^{ext}_{old-new}(node_x) \leftarrow APD^{mark}_{new}(node_x) \wedge \forall node_y \neq node_x : [LO^{ext}_{new}(node_y) \wedge APD^{disp}_{old-new}(node_y)]]$
(6.7)	$\forall node_x : [RPD^{mark}_{new}(node_x) \wedge UI^{int}_{old}(node_x) \leftarrow ISS^{client}_{old}(node_x)]$
(6.8)	N/A
(6.9)	$\forall node_x : [LI^{ext}_{old-new}(node_x) \leftarrow LO^{int}_{new}(node_x) \wedge AP^{client}_{new}(node_x) \wedge \forall node_y \neq node_x : LI^{int}_{old-new}(node_y)]$
(6.10)	N/A
(6.11)	$\forall node_x : [UO^{int}_{old}(node_x) \leftarrow LI^{ext}_{old-new}(node_x)]$

Table J.8: Customization of NeCoMan's algorithm for synchronized distributed reconfiguration: overview of all reconfiguration conditions that become adapted when the old and new components on node x encapsulate only client processes, and the **new** client processes communicate by a unidirectional communication protocol.

affected reconfig-uration condition	**resulting safety condition**
(5.1)	$\forall node_x : [LO^{ext}_{new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.2)	N/A
(5.4)	N/A
(5.5)	$\forall node_x : [DC_{old}(node_x) \leftarrow UO^{ext}_{old}(node_x) \wedge UO^{new}_{old}(node_x) \wedge LI^{int}_{old-new}(node_x)]$
(5.6)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.7)	$\forall node_x : [AP^{server}_{new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.8)	N/A
(5.9)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow ISS^{server}_{old}(node_x)]$
(5.10)	N/A
(5.11)	N/A
(5.13)	$\forall node_x : [UO^{int}_{old}(node_x) \leftarrow ISS^{server}_{old}(node_x)]$
(6.1)	$\forall node_x : [LO^{ext}_{new}(node_x) \wedge APD^{disp}_{old-new}(node_x) \leftarrow CC_{new}(node_x)]$
(6.2)	N/A
(6.3)	N/A
(6.8)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow LO^{ext}_{new}(node_x) \wedge AP^{server}_{new}(node_x)]$
(6.9)	N/A
(6.10)	$\forall node_x : [UO^{ext}_{old}(node_x) \leftarrow LI^{int}_{old-new}(node_x)]$
(6.11)	$\forall node_x : [UO^{int}_{old}(node_x) \leftarrow LI^{int}_{old-new}(node_x)]$

Table J.9: Customization of NeCoMan's algorithm for synchronized distributed reconfiguration: overview of all reconfiguration conditions that become adapted when the old and new components on node x encapsulate only server processes, and the **new** server processes communicate by a unidirectional communication protocol.

affected reconfiguration condition	**resulting safety condition**
(5.1)	$\forall node_x : [LO^{int}_{new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.3)	N/A
(5.4)	$\forall node_x : [RP^{client}_{new}(node_x) \leftarrow LI^{int}_{new}(node_x) \wedge LI^{ext}_{old-new}(node_x) \wedge AP^{client}_{new}(node_x) \wedge \forall node_y \neq node_x : [LI^{int}_{old-new}(node_y) \wedge AP^{server}_{new}(node_y)]]$
(5.5)	$\forall node_x : [DC_{old}(node_x) \leftarrow UO^{int}_{old}(node_x) \wedge LI^{ext}_{old-new}(node_x)]$
(5.6)	$\forall node_x : [LI^{int}_{new}(node_x) \wedge LI^{ext}_{old-new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.7)	$\forall node_x : [AP^{client}_{new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.9)	N/A
(5.12)	N/A
(5.13)	$\forall node_x : [UO^{int}_{old}(node_x) \leftarrow ISS^{client}_{old}(node_x)]$
(5.14)	N/A
(6.1)	$\forall node_x : [APD^{mark}_{new}(node_x) \leftarrow CC_{new}(node_x)]$
(6.4)	$\forall node_x : [DC_{old}(node_x) \leftarrow LI^{ext}_{old-new}(node_x) \wedge UO^{int}_{old}(node_x)]$
(6.5)	$\forall node_x : [LI^{ext}_{old-new}(node_x) \leftarrow APD^{mark}_{new}(node_x) \wedge LI^{int}_{new}(node_x) \wedge \forall node_y \neq node_x : [LO^{ext}_{new}(node_y) \wedge LO^{int}_{new}(node_y) \wedge APD^{disp}_{old-new}(node_y)]]$
(6.7)	$\forall node_x : [RPD^{mark}_{new}(node_x) \leftarrow ISS^{client}_{old}(node_x)]$
(6.8)	$\forall node_x : [LI^{int}_{new}(node_x) \leftarrow LO^{int}_{new}(node_x) \wedge AP^{client}_{new}(node_x)]$
(6.9)	$\forall node_x : [LI^{ext}_{old-new}(node_x) \leftarrow LO^{int}_{new}(node_x) \wedge AP^{client}_{new}(node_x) \wedge LI^{int}_{new}(node_x) \wedge \forall node_y \neq node_x : LI^{int}_{old-new}(node_y)]$
(6.10)	N/A
(6.11)	$\forall node_x : [UO^{int}_{old}(node_x) \leftarrow LI^{ext}_{old-new}(node_x)]$

Table J.10: Customization of NeCoMan's algorithm for synchronized distributed reconfiguration: overview of all reconfiguration conditions that become adapted when the old and new components on node x encapsulate only client processes, and the **old** client processes communicate by a unidirectional communication protocol.

affected reconfig-uration condition	**resulting safety condition**
(5.2)	N/A
(5.4)	N/A
(5.5)	$\forall node_x : [DC_{old}(node_x) \leftarrow UO^{ext}_{old}(node_x) \wedge LI^{int}_{old-new}(node_x)]$
(5.6)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.7)	$\forall node_x : [AP^{server}_{new}(node_x) \leftarrow CC_{new}(node_x)]$
(5.8)	N/A
(5.9)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow ISS^{server}_{old}(node_x)]$
(5.10)	N/A
(5.11)	N/A
(5.13)	N/A
(6.1)	$\forall node_x : [LO^{ext}_{new}(node_x) \wedge LO^{int}_{new}(node_x) \wedge APD^{disp}_{old-new}(node_x) \leftarrow CC_{new}(node_x)]$
(6.3)	N/A
(6.4)	$\forall node_x : [DC_{old}(node_x) \leftarrow UO^{ext}_{old}(node_x) \wedge RPD^{disp}_{old-new}(node_x)]$
(6.5)	N/A
(6.6)	N/A
(6.7)	$\forall node_x : [RPD^{disp}_{old-new}(node_x) \leftarrow ISS^{server}_{old}(node_x)]$
(6.8)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow LO^{ext}_{new}(node_x) \wedge LO^{int}_{new}(node_x) \wedge AP^{server}_{new}(node_x)]$
(6.9)	N/A
(6.11)	N/A

Table J.11: Customization of NeCoMan's algorithm for synchronized distributed reconfiguration: overview of all reconfiguration conditions that become adapted when the old and new components on node x encapsulate only server processes, and the **old** server processes communicate by a unidirectional communication protocol.

	reconfiguration condition
(3.1)	$LO^{int}_{new} \leftarrow CC_{new}$
(3.6)	$DC_{old} \leftarrow UO^{int}_{old} \wedge LI^{ext}_{old-new} \wedge UI^{int}_{old}$
(3.7)	$LI^{ext}_{old-new} \leftarrow CC_{new}$
(4.3)	$LI^{ext}_{old-new} \leftarrow AP^{client}_{new} \wedge LO^{int}_{new}$
(4.9)	$UO^{int}_{old} \leftarrow UI^{int}_{old} \wedge LI^{ext}_{old-new}$

Table J.12: Customization of NeCoMan's algorithm for independent distributed reconfiguration: overview of all reconfiguration conditions that have to be fulfilled when replacing bidirectional client processes with unidirectional ones without reaching quiescence.

	reconfiguration condition
(3.1)	$LO^{ext}_{new} \leftarrow CC_{new}$
(3.6)	$DC_{old} \leftarrow UO^{ext}_{old} \wedge UO^{int}_{old} \wedge LI^{int}_{old-new}$
(3.7)	$LI^{int}_{old-new} \leftarrow CC_{new}$
(4.2)	$LI^{int}_{old-new} \leftarrow AP^{server}_{new} \wedge LO^{ext}_{new}$
(4.9)	$UO^{int}_{old} \leftarrow LI^{int}_{old-new}$
(4.10)	$UO^{ext}_{old} \leftarrow LI^{int}_{old-new}$

Table J.13: Customization of NeCoMan's algorithm for independent distributed reconfiguration: overview of all reconfiguration conditions that have to be fulfilled when replacing bidirectional server processes with unidirectional ones without reaching quiescence.

	reconfiguration condition
(3.1)	$LO^{int}_{new} \leftarrow CC_{new}$
(3.6)	$DC_{old} \leftarrow UO^{int}_{old} \wedge LI^{ext}_{old-new}$
(3.7)	$LI^{ext}_{old-new} \wedge LI^{int}_{new} \leftarrow CC_{new}$
(4.2)	$LI^{int}_{new} \leftarrow AP^{client}_{new} \wedge LO^{int}_{new}$
(4.3)	$LI^{ext}_{old-new} \leftarrow AP^{client}_{new} \wedge LO^{int}_{new} \wedge LI^{int}_{new}$
(4.9)	$UO^{int}_{old} \leftarrow LI^{ext}_{old-new}$

Table J.14: Customization of NeCoMan's algorithm for independent distributed reconfiguration: overview of all reconfiguration conditions that have to be fulfilled when replacing unidirectional client processes with bidirectional ones without reaching quiescence.

reconfiguration condition	
(3.1)	$LO^{ext}_{new} \wedge LO^{int}_{new} \leftarrow CC_{new}$
(3.6)	$DC_{old} \leftarrow UO^{ext}_{old} \wedge LI^{int}_{old-new}$
(3.7)	$LI^{int}_{old-new} \leftarrow CC_{new}$
(4.2)	$LI^{int}_{old-new} \leftarrow AP^{server}_{new} \wedge LO^{ext}_{new} \wedge LO^{int}_{new}$
(4.10)	$UO^{ext}_{old} \leftarrow LI^{int}_{old-new}$

Table J.15: Customization of NeCoMan's algorithm for independent distributed reconfiguration: overview of all reconfiguration conditions that have to be fulfilled when replacing unidirectional server processes with bidirectional ones without reaching quiescence.

affected reconfiguration condition	**resulting safety condition**
(5.2)	N/A
(5.3)	$\forall node_x : [ISS^{server}_{old}(node_x) \leftarrow \forall node_y \neq node_x : IP^{client}_{old}(node_y)]$
(5.5)	N/A
(5.9)	$\forall node_x : [LI^{int}_{old-new}(node_x) \leftarrow ISS^{server}_{old}(node_x)]$
(5.10)	$\forall node_x : [LI^{ext}_{old-new}(node_x) \leftarrow IP^{client}_{old}(node_x)]$
(5.11)	N/A
(5.12)	N/A
(5.13)	N/A
(5.14)	N/A
(6.4)	N/A
(6.6)	N/A
(6.7)	$\forall node_x : [RPD^{disp}_{old-new}(node_x) \leftarrow ISS^{server}_{old}(node_x)]$
(6.10)	N/A
(6.11)	N/A

Table J.16: Customization of NeCoMan's algorithm for synchronized distributed reconfiguration: overview of all reconfiguration conditions that become changed in case of service addition.

affected reconfiguration condition	**resulting safety condition**
(5.1)	N/A
(5.4)	$\forall node_x : [RP_{new}^{client}(node_x) \leftarrow LI_{old-new}^{ext}(node_x) \wedge \forall node_y \neq node_x : LI_{old-new}^{int}(node_y)]$
(5.6)	N/A
(5.7)	N/A
(5.8)	N/A
(5.9)	$\forall node_x : [LI_{old-new}^{int}(node_x) \leftarrow ISS_{old}^{client}(node_x) \wedge ISS_{old}^{server}(node_x)]$
(5.10)	$\forall node_x : [LI_{old-new}^{ext}(node_x) \leftarrow ISS_{old}^{client}(node_x)]$
(5.11)	N/A
(5.12)	N/A
(6.1)	N/A
(6.2)	N/A
(6.4)	$\forall node_x : [DC_{old}(node_x) \leftarrow UO_{old}^{ext}(node_x) \wedge LI_{old-new}^{'ext}(node_x) \wedge RPD_{old-new}^{disp}(node_x) \wedge UO_{old}^{int}(node_x)]$
(6.5)	$\forall node_x : [LI_{old-new}^{'ext}(node_x) \leftarrow APD_{new}^{'mark}(node_x) \wedge \forall node_y \neq node_x : APD_{old-new}^{disp}(node_y)]$
(6.6)	$\forall node_x : [ISS_{old}^{client}(node_x) \leftarrow LI_{old-new}^{'ext}(node_x)]$
(6.8)	N/A
(6.9)	$\forall node_x : [LI_{old-new}^{ext}(node_x) \leftarrow \forall node_y \neq node_x : LI_{old-new}^{int}(node_y)]$

Table J.17: Customization of NeCoMan's algorithm for synchronized distributed reconfiguration: overview of all reconfiguration conditions that become changed in case of service removal.

affected reconfiguration condition	**resulting safety condition**
(3.2)	N/A
(3.3)	N/A
(3.6)	N/A
(3.8)	$RP_{new}^{client} \leftarrow LO_{new}^{int}$
(3.9)	$RP_{new}^{server} \leftarrow LO_{new}^{ext} \wedge LO_{new}^{int}$
(3.10)	$LI_{old-new}^{int} \leftarrow IP_{old}^{server}$
(3.11)	$LI_{old-new}^{ext} \leftarrow IP_{old}^{client}$
(3.12)	N/A
(3.13)	N/A
(3.14)	N/A
(4.4)	N/A
(4.5)	N/A
(4.9)	N/A
(4.10)	N/A

Table J.18: Customization of NeCoMan's algorithm for independent distributed reconfiguration: overview of all reconfiguration conditions that are changed in case of service addition.

affected reconfiguration condition	**resulting safety condition**
(3.1)	N/A
(3.4)	$RP_{new}^{client} \leftarrow LI_{old-new}^{ext}$
(3.5)	$RP_{new}^{server} \leftarrow LI_{old-new}^{int}$
(3.7)	N/A
(3.8)	N/A
(3.9)	N/A
(3.11)	$LI_{old-new}^{ext} \leftarrow ISS_{old}^{client}$
(3.12)	N/A
(4.1)	N/A
(4.2)	N/A
(4.3)	N/A

Table J.19: Customization of NeCoMan's algorithm for independent distributed reconfiguration: overview of all reconfiguration conditions that are changed in case of service removal.

Appendix K

Effect of distributed customizations on reconfiguration overhead

This appendix evaluates the impact on reconfiguration overhead that some of the customizations presented in Chapter 6 may bring about. To be precise, we analyze the first three customizations to NeCoMan's basic distributed reconfiguration algorithms[1]. To evaluate the effect of these customizations, we compare the cost incurred by NeCoMan's basic distributed algorithms (depicted in Figures 5.13 and K.1) with the cost incurred when these customizations are applied. Similar as for NeCoMan's local reconfiguration algorithms, this cost will be evaluated in terms of communication disruption and the time that it takes to complete a reconfiguration (reconfiguration time). Besides, since we target distributed reconfigurations, we also evaluate the bandwidth that a reconfiguration consumes. This enables to compare the impact of (among others) sending synchronization messages.

K.1 No coordinated activation

A first customization seeks to optimize synchronized distributed reconfigurations by omitting the distributed synchronization that is needed to correctly activate new network service components. As explained in Section 6.2, NeCoMan uses its second distributed reconfiguration algorithm instead when applying this customization to its first distributed reconfiguration algorithm. Therefore, to illustrate the effect of this customization, we compare the cost incurred by using NeCoMan's algorithms

[1]These customizations include "no coordinated activation" (see Section 6.2), "activate before finishing" (see Section 6.3), and "no finishing" (see Section 6.4).

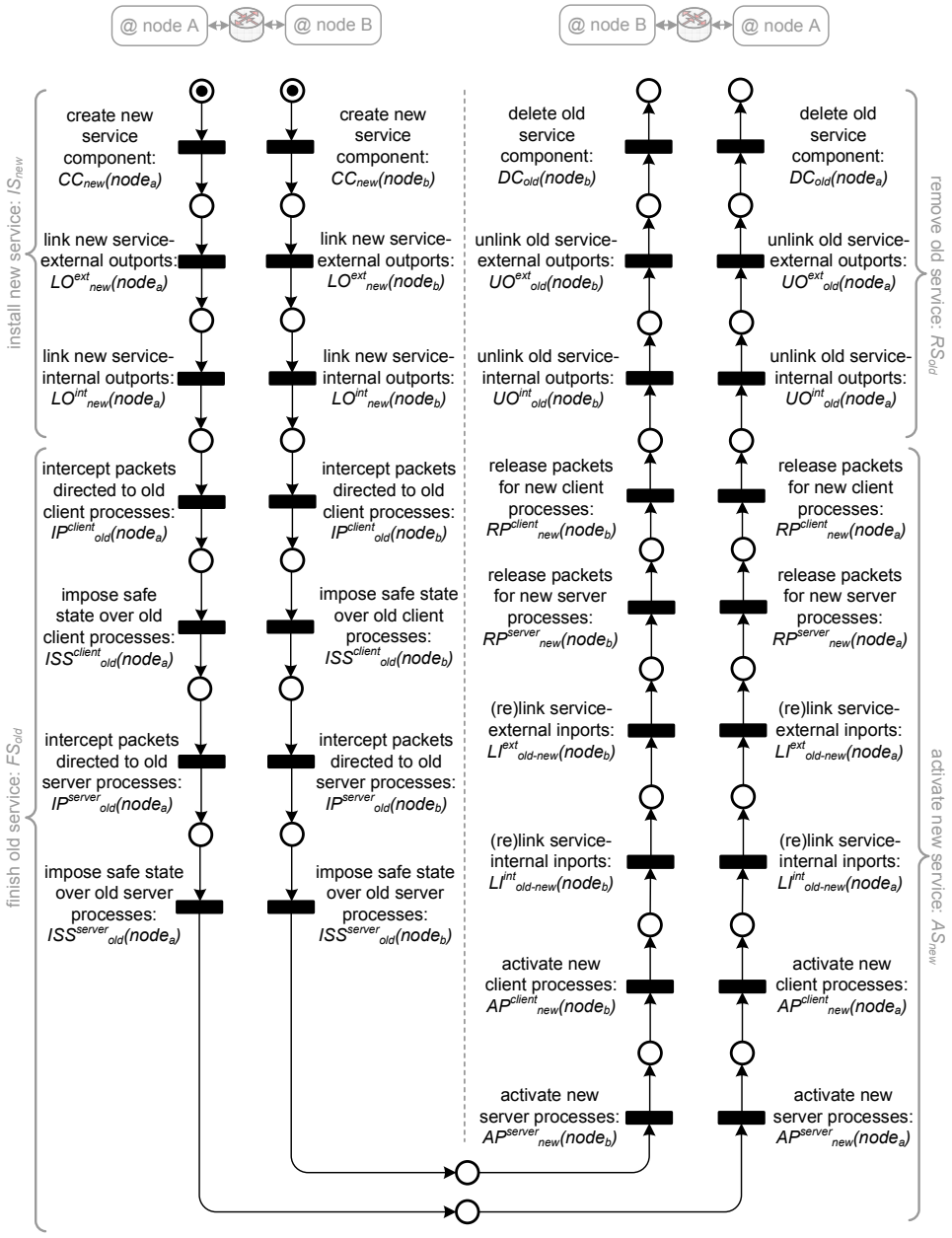


Figure K.1: Petri net representation of NeCoMan's algorithm for independent distributed reconfiguration.

for synchronized and independent reconfigurations with each others. These algorithms are depicted in Figures 5.13 and K.1, respectively.

Communication disruption

When NeCoMan employs its algorithm for synchronized distributed reconfiguration, the operation of all affected *client processes* located on node x will be interrupted as from the moment that packets are intercepted on that node for reaching quiescence until these packets are released again. End-node applications that (indirectly) invoke these client processes, therefore, will experience communication disruption as from the moment that NeCoMan executes $IP_{old}^{client}(node_x)$ until it completes $RP_{new}^{client}(node_x)$. We can infer from the model of NeCoMan's algorithm for synchronized distributed reconfiguration, which is depicted in Figure 5.13, that this period of communication disruption equals

$$\begin{aligned}\Delta_{disrupt}^{basic1}(client) = t(IP_{old}^{client}(node_x)) + t(ISS_{old}^{client}(node_x)) + \\ t(ISS_{old}^{server}(node_x)) + t(AP_{new}^{client}(node_x)) + t(AP_{new}^{server}(node_x)) + \\ t(LI_{old-new}^{int}(node_x)) + t(LI_{old-new}^{ext}(node_x)) + t(RP_{new}^{client}(node_x)) + \\ t(waitForEveryMessageA(node_x)) + t(waitForEveryMessageB(node_x))\end{aligned}$$

where *t(waitForEveryMessageA(node$_x$))* and *t(waitForEveryMessageB(node$_x$))* denote the time that NeCoMan must wait for all expected instances of messages **A** and **B** to arrive on node x.

The period in which *server processes* become disrupted when NeCoMan employs its algorithm for synchronized distributed reconfiguration, however, is slightly different. Server processes continue accepting and servicing packets (which are sent by collaborating client processes) until quiescence comes about and a reconfiguration-safe state is reached. Furthermore, note that the new server processes located on node x are brought into use once RP_{new}^{client} is executed on an arbitrary collaborating node y. So, the period in which server processes located on node x will be disrupted during the execution of a synchronized distributed reconfiguration equals

$$\begin{aligned}\Delta_{disrupt}^{basic1}(server) \quad = \quad & t(ISS_{old}^{server}(node_x)) + t(AP_{new}^{client}(node_x)) + \\ & t(AP_{new}^{server}(node_x)) + t(LI_{old-new}^{int}(node_x)) + \\ & t(synchExecRP_{new}^{client}(node_y)) + t(RP_{new}^{client}(node_y))\end{aligned}$$

where *t(synchExecRP$_{new}^{client}$(node$_y$))* denotes the time that it takes before NeCoMan can execute RP_{new}^{client} on node y.

The period in which client and server processes are interrupted when NeCoMan employs its second distributed reconfiguration algorithm, in contrast, is similar to when its local algorithm for distributed services is executed. This is due to the fact that NeCoMan's algorithm for independent distributed reconfiguration performs the distributed execution of its local algorithm for isolated services. Hence, as already

explained in Section 4.2.1, the period in which the old client processes located on node x will be disrupted thus equals

$$\Delta^{basic2}_{disrupt}(client) = t(IP^{client}_{old}(node_x)) + t(IP^{server}_{old}(node_x)) + t(ISS^{client}_{old}(node_x)) + t(ISS^{server}_{old}(node_x)) + t(AP^{client}_{new}(node_x)) + t(AP^{server}_{new}(node_x)) + t(LI^{ext}_{old-new}(node_x)) + t(LI^{int}_{old-new}(node_x)) + t(RP^{client}_{new}(node_x)) + t(RP^{server}_{new}(node_x))$$

Besides, the period in which the old server processes located on node x will be interrupted equals

$$\Delta^{basic2}_{disrupt}(server) = t(IP^{server}_{old}(node_x)) + t(ISS^{server}_{old}(node_x)) + t(AP^{client}_{new}(node_x)) + t(AP^{server}_{new}(node_x)) + t(LI^{ext}_{old-new}(node_x)) + t(LI^{int}_{old-new}(node_x)) + t(RP^{server}_{new}(node_x))$$

So, the difference in communication disruption that NeCoMan's algorithms for synchronized and independent distributed reconfiguration cause on the affected client process equals

$$\Delta^{basic1}_{disrupt}(client) - \Delta^{basic2}_{disrupt}(client) = t(waitForEveryMessageA(node_x)) + t(waitForEveryMessageB(node_x)) - t(IP^{server}_{old}(node_x)) - t(RP^{server}_{new}(node_x))$$

From this we can conclude that NeCoMan's algorithm for synchronized distributed reconfiguration causes less communication disruption on the affected client processes than its second distributed algorithm if

$$t(waitForEveryMessageA(node_x)) + t(waitForEveryMessageB(node_x)) < t(IP^{server}_{old}(node_x)) + t(RP^{server}_{new}(node_x))$$

This will be the case, among others, when the reconfiguration of all affected nodes occurs perfectly synchronized. This includes that all required instances of messages **A** and **B** have already arrived at node x when NeCoMan completes the execution of ISS^{client}_{old} and $LI^{ext}_{old-new}$, respectively. Besides, note that the time to wait for the arrival of all instances of messages **A** and **B** may potentially increase as a reconfiguration involves more nodes hosting server processes.

Similarly, the difference in communication disruption that NeCoMan's algorithm for synchronized and independent distributed reconfigurations cause on the affected server process equals

$$\Delta^{basic1}_{disrupt}(server) - \Delta^{basic2}_{disrupt}(server) = t(synchExecRP^{client}_{new}(node_y)) + t(RP^{client}_{new}(node_y)) - t(IP^{server}_{old}(node_x)) - t(RP^{server}_{new}(node_x))$$

NeCoMan's algorithm for synchronized distributed reconfiguration thus causes less communication disruption on the affected server processes than its second algorithm if

$$\begin{aligned} t(synchExecRP_{new}^{client}(node_y)) &+ t(RP_{new}^{client}(node_y)) < \\ t(IP_{old}^{server}(node_x)) &+ t(RP_{new}^{server}(node_x)) \end{aligned}$$

Reconfiguration time

To define the time that it takes to complete a reconfiguration on node x when NeCoMan uses its algorithm for synchronized distributed reconfiguration, we consult (again) the model depicted in Figure 5.13. From this model we infer that Δ_{reconf}^{basic1} equals

$$\begin{aligned} &t(CC_{new}) + t(LO_{new}^{ext}) + t(LO_{new}^{int}) + t(IP_{old}^{client}) + t(ISS_{old}^{client}) + \\ &t(ISS_{old}^{server}) + t(AP_{new}^{client}) + t(AP_{new}^{server}) + t(LI_{old-new}^{ext}) + \\ &t(LI_{old-new}^{int}) + t(RP_{new}^{client}) + t(UO_{old}^{ext}) + t(UO_{old}^{int}) + t(DC_{old}) + \\ &t(waitForEveryMessageA) + t(waitForEveryMessageB) \end{aligned}$$

Besides, the time that it takes to complete a reconfiguration on node x when NeCoMan uses algorithm for independent reconfiguration is similar as when using the algorithm for synchronized reconfiguration. Hence, Δ_{reconf}^{basic2} equals

$$\begin{aligned} &t(CC_{new}) + t(LO_{new}^{ext}) + t(LO_{new}^{int}) + t(IP_{old}^{client}) + t(ISS_{old}^{client}) + \\ &t(IP_{old}^{server}) + t(ISS_{old}^{server}) + t(AP_{new}^{client}) + t(AP_{new}^{server}) + \\ &t(LI_{old-new}^{ext}) + t(LI_{old-new}^{int}) + t(RP_{new}^{client}) + t(RP_{new}^{server}) + \\ &t(UO_{old}^{ext}) + t(UO_{old}^{int}) + t(DC_{old}) \end{aligned}$$

Comparing both Δ_{reconf}^{basic1} with Δ_{reconf}^{basic2} then results in

$$\begin{aligned} \Delta_{reconf}^{basic1} - \Delta_{reconf}^{basic2} &= t(waitForEveryMessageA) + t(waitForEveryMessageB) - \\ &\quad t(IP_{old}^{server}) - t(RP_{new}^{server}) \end{aligned}$$

This is equal to the difference in communication disruption that NeCoMan's algorithms for synchronized and independent distributed reconfigurations cause on the client processes of node x. Hence, Δ_{reconf}^{basic1} will be smaller than Δ_{reconf}^{basic2} when the reconfiguration of all affected nodes occurs perfectly synchronized.

Bandwidth consumption

Finally, we compare the bandwidth that both algorithms consume. Recall that NeCoMan's algorithm for synchronized distributed reconfiguration exchanges synchronization messages **A** and **B**. NeCoMan's algorithm for independent distributed

reconfiguration, in contrast, does not require distributed synchronization. The first algorithm therefore consumes more bandwidth than the second one. The exact bandwidth consumption, however, depends on how many instances of messages **A** and **B** are transmitted during reconfiguration. This, in turn, depends on what other customizations NeCoMan has applied, as well as on the amount of nodes that participate in a reconfiguration.

K.2 Activate before finishing

A second customization that applies to NeCoMan's distributed reconfiguration algorithms involves activating a new service before the old one is finished, as explained in Section 6.3. This way, NeCoMan seeks to reduce the communication disruption that a reconfiguration causes.

K.2.1 Synchronized distributed reconfigurations

To evaluate the effect of this customization on the reconfiguration overhead that NeCoMan's algorithm for synchronized distributed reconfiguration causes, we compare the cost incurred by the algorithm modelled in Figure 5.13 with the cost incurred by using the algorithm depicted in Figure 6.5.

Communication disruption

When NeCoMan activates a new service before finishing the old one, the affected *client processes* located on node x will be disrupted

1. during the integration of dispatching components,
2. when packet flows are redirected towards the service-external inports of the new service components,
3. during the removal of the employed marking components, and
4. during the removal of the employed dispatching components.

Hence, the total period in which the continuity of these client processes becomes disrupted equals

$$\Delta_{disrupt}^{basic1+cust2}(client) = t(LI_{old-new}^{ext}) + \Delta_{disrupt}(APD_{old-new}^{disp}) + \Delta_{disrupt}(RPD_{new}^{mark}) + \Delta_{disrupt}(RPD_{old-new}^{disp})$$

Besides, recall that NeCoMan uses its second local reconfiguration algorithm tailored with the "activate before finishing" customization to execute $APD_{old-new}$,

RPD^{mark}_{new}, and $RPD^{disp}_{old-new}$ on node x. From this, we infer that

$$\Delta_{disrupt}(APD^{disp}_{old-new}) + \Delta_{disrupt}(RPD^{mark}_{new}) + \Delta_{disrupt}(RPD^{disp}_{old-new}) = 3 * t(LI_{old-new})$$

such that

$$\Delta^{basic1+cust2}_{disrupt}(client) = t(LI^{ext}_{old-new}) + 3 * t(LI_{old-new})$$

So, the difference in communication disruption that both algorithms cause on the affected client processes equals

$$\Delta^{basic1}_{disrupt}(client) - \Delta^{basic1+cust2}_{disrupt}(client) = t(IP^{client}_{old}) + t(ISS^{client}_{old}) + t(ISS^{server}_{old}) + t(AP^{client}_{new}) + t(AP^{server}_{new}) + t(LI^{int}_{old-new}) + t(RP^{client}_{new}) + t(waitForEveryMessageA) + t(waitForEveryMessageB) - 3 * t(LI_{old-new})$$

From this conclude that in most cases $\Delta^{basic1+cust2}_{disrupt}(client)$ will be significantly smaller than $\Delta^{basic1}_{disrupt}(client)$. Note, however, that this difference reduces when the old service becomes finished in a negligible period of time. Besides, when the reconfiguration of all affected nodes also occurs perfectly synchronized in this case, and the new processes employ no active objects, then the difference in communication disruption reduces to

$$\Delta^{basic1}_{disrupt}(client) - \Delta^{basic1+cust2}_{disrupt}(client) \approx t(IP^{client}_{old}) + t(LI^{int}_{old-new}) + t(RP^{client}_{new}) - 3 * t(LI_{old-new})$$

So, in this case switching the order of activation and finishing does not significantly reduce the communication disruption that a reconfiguration causes on the affected client processes.

Furthermore, when NeCoMan activates a new service before finishing the old one, the affected *server processes* located on node x will be disrupted during the execution of $APD^{disp}_{old-new}$, RPD^{mark}_{new}, and $RPD^{disp}_{old-new}$. Hence, the total period in which the continuity of these server processes becomes disrupted equals

$$\Delta^{basic1+cust2}_{disrupt}(server) = 3 * t(LI_{old-new}(node_x))$$

So, the difference in communication disruption that switching the order of activation and finishing causes on the affected server processes equals

$$\Delta^{basic1}_{disrupt}(server) - \Delta^{basic1+cust2}_{disrupt}(server) = t(ISS^{server}_{old}(node_x)) + t(AP^{client}_{new}(node_x)) + t(AP^{server}_{new}(node_x)) + t(LI^{int}_{old-new}(node_x)) + t(synchExecRP^{client}_{new}(node_y)) + t(RP^{client}_{new}(node_y)) - 3 * t(LI_{old-new}(node_x))$$

Similarly as for client processes, this difference in communication disruption reduces significantly when

- the old service finishes in a negligible period of time,
- the new service processes do not employ active objects, and
- the reconfiguration of all affected nodes also occurs perfectly synchronized.

So, when these conditions are met, the difference in communication disruption reduces to

$$\Delta_{disrupt}^{basic1}(server) - \Delta_{disrupt}^{basic1+cust2}(server) \approx$$
$$t(LI_{old-new}^{int}(node_x)) + t(RP_{new}^{client}(node_y)) - 3 * t(LI_{old-new}(node_x))$$

Hence, in this case switching the order of activation and finishing does not significantly reduce the communication disruption that a reconfiguration causes on the affected server processes.

Reconfiguration time

From the model depicted in Figure 6.5, we can infer that the time it takes to complete a reconfiguration that applies this customizations equals

$$\begin{aligned}\Delta_{reconf}^{basic1+cust2} \quad = \quad & t(CC_{new}) + t(LO_{new}^{ext}) + t(APD_{new}^{mark}) + t(APD_{new}^{disp}) + \\ & t(AP_{new}^{client}) + t(AP_{new}^{server}) + t(LI_{old-new}^{ext}) + t(ISS_{old}^{client}) + \\ & t(ISS_{old}^{server}) + t(RPD_{new}^{mark}) + t(RPD_{new}^{disp}) + t(UO_{old}^{ext}) + \\ & t(UO_{old}^{int}) + t(DC_{old}) + t(waitForEveryMessageC) + \\ & t(waitForEveryMessageA) + t(waitForEveryMessageE)\end{aligned}$$

Comparing this with the time that it takes to complete NeCoMan's algorithm for synchronized distributed reconfiguration results in

$$\begin{aligned}\Delta_{reconf}^{basic1} \quad - \quad & \Delta_{reconf}^{basic1+cust2} = t(LO_{new}^{int}) + t(IP_{old}^{client}) + t(RP_{new}^{client}) + \\ & t(waitForEveryMessageB) - t(APD_{new}^{mark}) - t(APD_{new}^{disp}) - \\ & t(RPD_{new}^{mark}) - t(RPD_{new}^{disp}) - t(waitForEveryMessageC) - \\ & t(waitForEveryMessageE)\end{aligned}$$

So, we cannot conclude that the time it takes to complete a reconfiguration will always be shorter for one of both scenarios[2]. Note, however, that for most reconfigurations Δ_{reconf}^{basic1} will be smaller than $\Delta_{reconf}^{basic1+cust2}$. This is because NeCoMan's basic algorithm for synchronized distributed reconfiguration needs less synchronization than when the activation and finishing actions are switched. Besides, take into

[2]that is, in contrast to local reconfigurations as explained in Section E.1.1

account that executing APD_{new}^{disp}, RPD_{new}^{mark}, and RPD_{new}^{disp} involves dynamic recomposition. Hence, the time it takes to complete these actions will in most cases be larger than the time to complete LO_{new}^{int}, IP_{old}^{client}, and RP_{new}^{client}. Note, however, that we cannot make hard statements about all this, since much depends on the efficiency in which the affected nodes execute the reconfiguration operations that NeCoMan invokes.

Bandwidth consumption

Finally, we compare the bandwidth that both algorithms consume. Recall that NeCoMan's algorithm for synchronized distributed reconfiguration transmits synchronization messages **A** and **B**. When activating a new service before the old one is finished, however, three synchronization messages are involved (**C**, **A**, and **E**). In addition, extra header bits are set to mark packets. Hence, when NeCoMan applies the "activate before finishing" customization to its algorithm for synchronized distributed reconfiguration, bandwidth consumption increases.

Conclusion

We conclude from this evaluation that the current version of NeCoMan cannot always take the most optimal decision when choosing whether or not to apply customization 6.3. For instance, when replacing a stateless service that takes a long time to finish, activating the new service before the old one finishes reduces communication disruption. But, when the service reaches the finished state in a negligible period of time, NeCoMan's basic algorithm might be a better choice. This solution needs less synchronization and lacks the potential performance overhead that the packet-distinguishing support causes.

K.2.2 Independent distributed reconfigurations

As explained in Section 6.3.2, applying this customization to NeCoMan's algorithm for independent distributed reconfiguration is identical as when its local algorithm for distributed services is involved. The effect of this customization on the overhead that NeCoMan's algorithm for independent distributed reconfiguration causes, therefore, differs in nothing from the evaluation presented is Section E.1.1.

K.3 No finishing

An additional customization to optimize NeCoMan's distributed reconfiguration algorithms involves omitting all finishing actions. This customization has been presented in Section 6.4.

K.3.1 Synchronized distributed reconfigurations

To evaluate the effect of this customization on the reconfiguration overhead that NeCoMan's algorithm for synchronized distributed reconfiguration causes, we compare the cost incurred by the algorithm modelled in Figure 5.13 with the cost incurred by using the algorithm depicted in Figure 6.7.

Communication disruption

When all finishing actions are omitted, the affected client processes located on node x are only disrupted while executing $LI^{int}_{old-new}$ and $LI^{ext}_{old-new}$. So,

$$\Delta^{basic1+cust4}_{disrupt}(client) = t(LI^{int}_{old-new}) + t(LI^{ext}_{old-new})$$

In addition, recall that server processes do not expose service-external inports according to our component model. When NeCoMan applies this customization, the affected server processes located on node x therefore are only disrupted while executing $LI^{int}_{old-new}$. Hence,

$$\Delta^{basic1+cust4}_{disrupt}(server) = t(LI^{int}_{old-new}(node_x))$$

Comparing $\Delta^{basic1}_{disrupt}(client)$ with $\Delta^{basic1+cust4}_{disrupt}(client)$ then results in

$$\begin{aligned}\Delta^{basic1}_{disrupt}(client) - \Delta^{basic1+cust4}_{disrupt}(client) = t(IP^{client}_{old}) + t(ISS^{client}_{old}) + \\ t(ISS^{server}_{old}) + t(AP^{client}_{new}) + t(AP^{server}_{new}) + t(RP^{client}_{new}) + \\ t(waitForEveryMessageA) + t(waitForEveryMessageB)\end{aligned}$$

Similarly, the difference in communication disruption that both scenarios cause on the affected server processes equals

$$\begin{aligned}\Delta^{basic1}_{disrupt}(server) \quad - \quad \Delta^{basic1+cust4}_{disrupt}(server) = t(ISS^{server}_{old}(node_x)) + \\ t(AP^{client}_{new}(node_x)) + t(AP^{server}_{new}(node_x)) + \\ t(synchExecRP^{client}_{new}(node_y)) + t(RP^{client}_{new}(node_y))\end{aligned}$$

From these equations we can conclude that omitting all finishing actions indeed reduces the reconfiguration overhead. Similar to local reconfigurations, however, this comparison does not take into account the effect of inconsistencies on the network performance in general.

Reconfiguration time

From the model depicted in Figure 6.7 we can deduce that the time it takes to complete a reconfiguration that involves no finishing action equals

$$\begin{aligned}\Delta^{basic1+cust4}_{reconf} = t(CC_{new}) + t(LO^{ext}_{new}) + t(LO^{int}_{new}) + t(AP^{client}_{new}) + \\ t(AP^{server}_{new}) + t(LI^{ext}_{old-new}) + t(LI^{int}_{old-new}) + t(UO^{ext}_{old}) + \\ t(UO^{int}_{old}) + t(DC_{old}) + t(waitForEveryMessageF)\end{aligned}$$

Compare this with the time that it takes to complete NeCoMan's algorithm for synchronized distributed reconfiguration results in

$$\Delta_{reconf}^{basic1} - \Delta_{reconf}^{basic1+cust4} = t(IP_{old}^{client}) + t(ISS_{old}^{client}) + t(ISS_{old}^{server}) + t(RP_{new}^{client}) + t(waitForEveryMessageA) + t(waitForEveryMessageB) - t(waitForEveryMessageF)$$

Because message **F** is the equivalent of message **B**, we conclude that a reconfiguration takes less time to complete when all finishing actions are omitted.

Bandwidth consumption.

Finally, we compare the bandwidth that both algorithms consume. Recall that NeCoMan's algorithm for synchronized distributed reconfiguration transmits synchronization messages **A** and **B**. When all finishing actions are omitted, however, only message **F** is used. So, when NeCoMan applies this customization, bandwidth consumption decreases.

K.3.2 Independent distributed reconfigurations

Applying this customization to NeCoMan's algorithm for independent distributed reconfiguration (again) is identical as when its local algorithm for distributed services is involved (as explained in Section 6.4.2). The effect of this customization on the overhead that NeCoMan's independent distributed reconfiguration causes, therefore, differs in nothing from the evaluation presented is Section E.2.1.

Appendix L

Customization procedure for distributed reconfigurations

This appendix briefly sketches in what order NeCoMan applies the customizations presented in Chapter 6 to its distributed reconfiguration algorithms. The flowcharts depicted in Figures L.1, L.2 and L.3 model how NeCoMan tailors its distributed reconfiguration algorithms in case of service addition, removal and replacement, respectively. In addition, Table L.1 lists the service characteristics and reconfiguration semantics that NeCoMan uses to identify which customizations it can apply. Finally, Figure L.4 illustrates how NeCoMan tailors its algorithm for synchronized distributed reconfiguration when taking into account the service characteristics and reconfiguration semantics listed in Table 7.1.

	service characteristics
S	the old and new services are compatible
C	the old service components are stateless
T	the old service components share their execution state with their client applications
E	the new components are able to continue processing all ongoing protocol-transactions
D	the new service components restore from or tolerate inconsistent execution states
U	the old service components communicate by a protocol that terminates locally
V	the old server processes do not encapsulate state that goes beyond the execution of a single protocol transaction
F	the new client processes do not employ active objects
G	the new server processes do not employ active objects
I	the old and new components encapsulate only client processes
J	the old and new components encapsulate only server processes
K	the old and/or new components encapsulate both client and server processes
L	the old and/or new components communicate by a unidirectional communication protocol
	reconfiguration semantics
W	the affected nodes impose a safe state by deactivating the old components immediately and transferring their execution state, instead of waiting until a quiescent execution state comes about
M	the network tolerates packet-reordering
N	the affected components operate in a best-effort network
O	the network restores from or tolerates inconsistent execution states
X	the network can handle inconsistent service compositions

Table L.1: The service characteristics and reconfiguration semantics that NeCoMan uses to identify which customizations it can apply to its distributed reconfiguration algorithms. These properties result from the network administrator answering the questions listed in Tables 6.3 and 6.4.

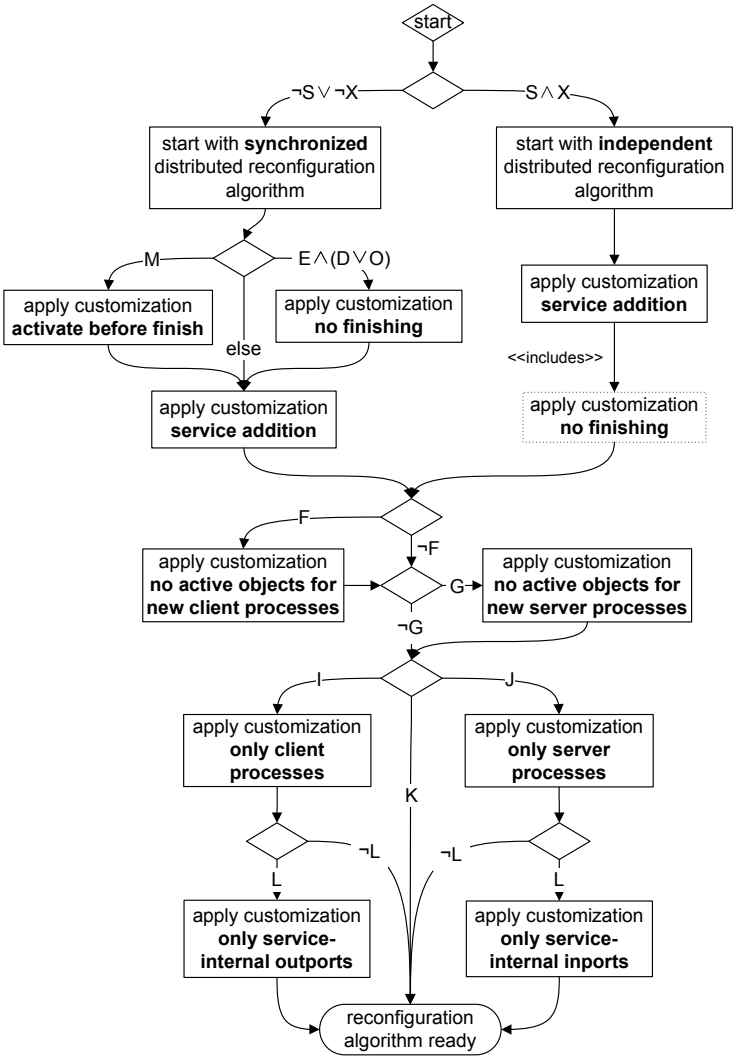


Figure L.1: Procedure to customize both distributed reconfiguration algorithms in case of service addition. The involved service characteristics and reconfiguration semantics are listed in Table L.1.

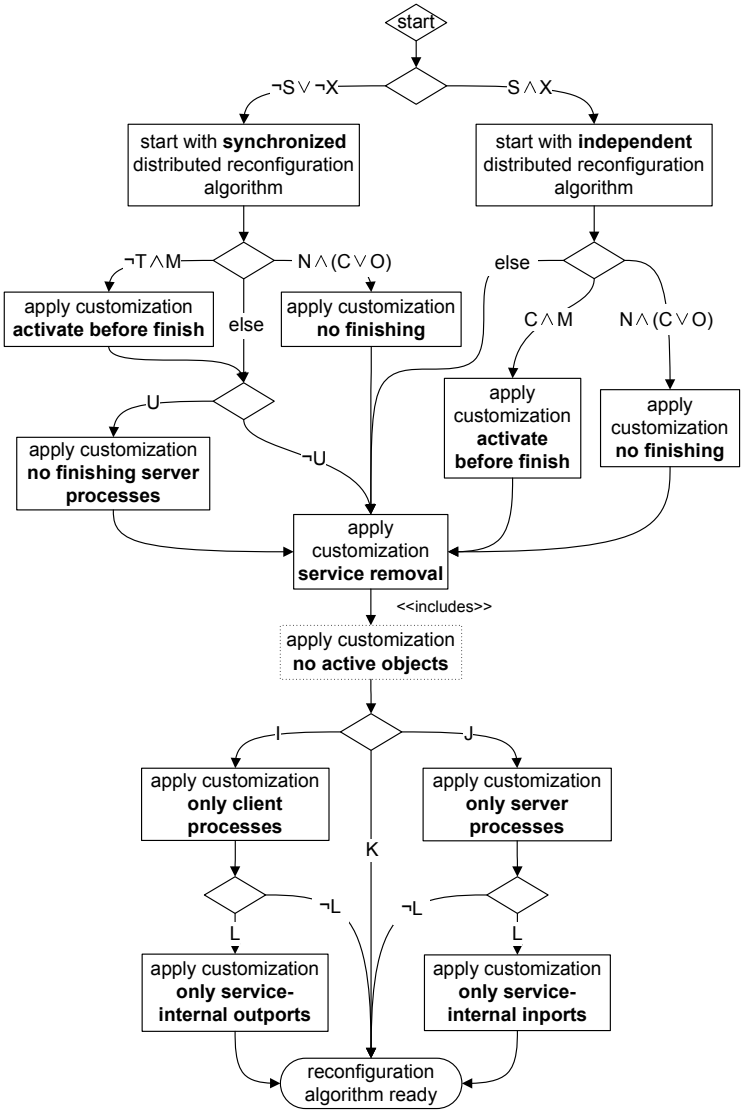


Figure L.2: Procedure to customize both distributed reconfiguration algorithms in case of service removal. The involved service characteristics and reconfiguration semantics are listed in Table L.1.

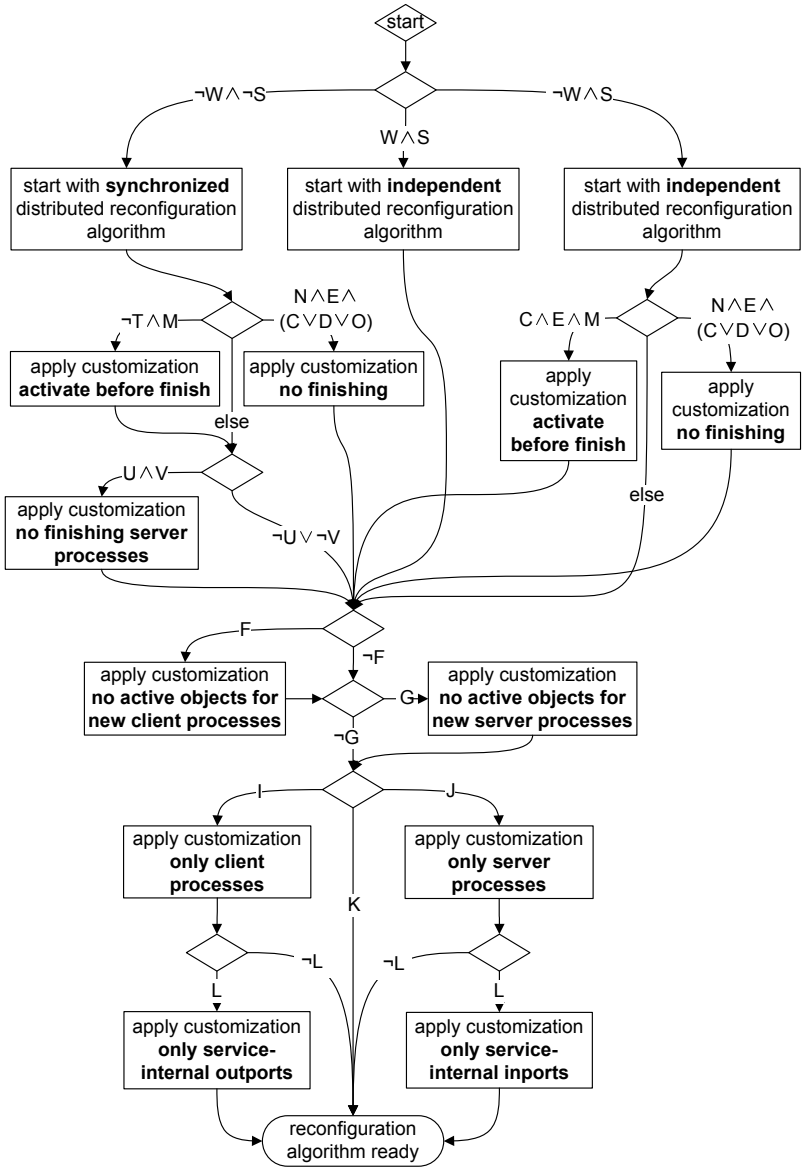


Figure L.3: Procedure to customize both distributed reconfiguration algorithms in case of service replacement. The involved service characteristics and reconfiguration semantics are listed in Table L.1.

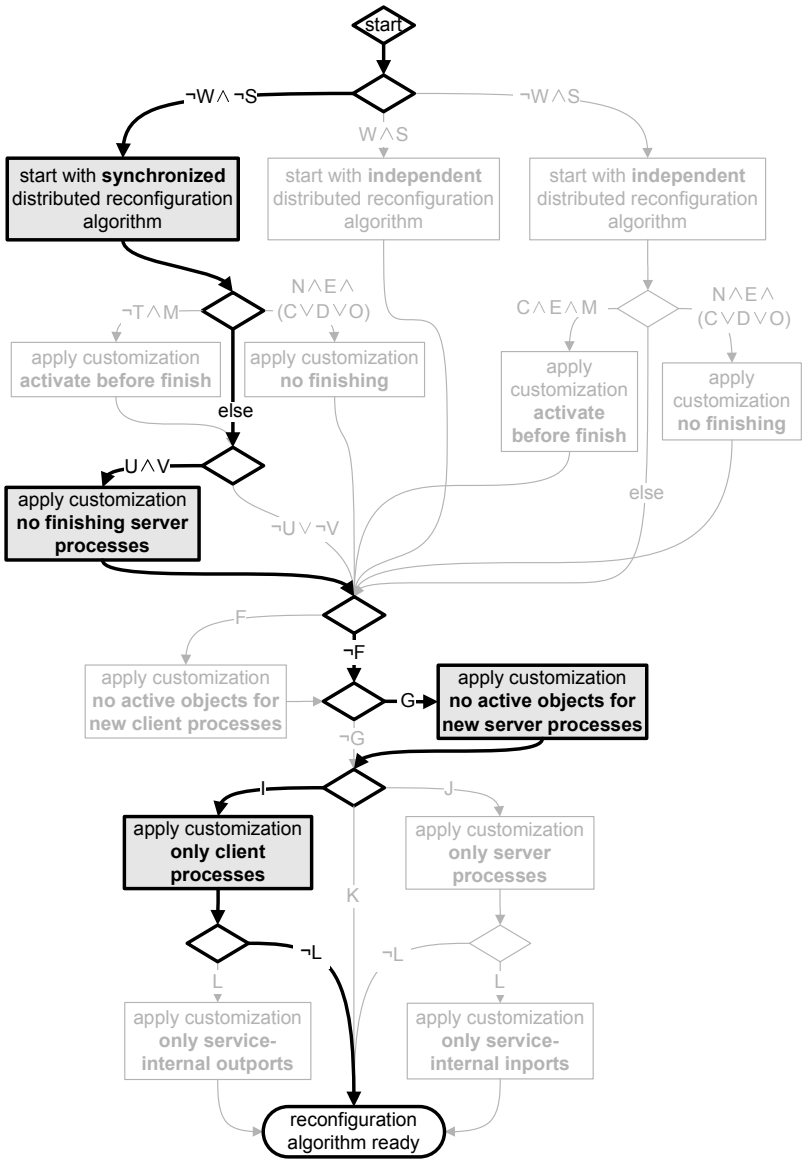


Figure L.4: Customization procedure when replacing R_{old} with R_{new} (as part of replacing the complete reliability service). This customization takes into account the service characteristics and reconfiguration semantics listed in Table 7.1.

List of Publications

Articles in international reviewed journals

1. N. Janssens, W. Joosen, and P. Verbaeten, NeCoMan: middleware for safe distributed-service adaptation in programmable networks, *IEEE Distributed Systems Online* **6** (7), pp. 1-11, July, 2005

Parts of books

1. S. Michiels, N. Janssens, L. Desmet, T. Mahieu, W. Joosen, and P. Verbaeten, A component platform for flexible protocol stacks, In *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends* (Atkinson, C. and Bunse, C. and Gross, H-G. and Peper, C., eds.), vol. 3778/2005, Lecture Notes in Computer Science, Springer, pp. 185-208, November 2005

Contributions at international conferences, published in proceedings

1. A. Schoutteet, J. Hoebeke, T. Van Leeuwen, I. Moerman, B. Dhoedt, P. Demeester, N. Janssens, and P. Verbaeten, An UDP protocol booster for multimedia communications over 802.11 Wireless LAN, In *Proceedings of the 5th Annual Information Technology & Telecommunications Conference* (IT&T 2005), pp. 151-161, 2005

2. S. Michiels, N. Janssens, W. Joosen, and P. Verbaeten, Decentralized cooperative management: a bottom up approach, In *Proceedings of the IADIS International Conference Applied Computing 2005* (Guimaraes, N. and Isaias, P., eds.), pp. 401-408, 2005

3. N. Janssens, L. Desmet, S. Michiels, and P. Verbaeten, NeCoMan: middleware for safe distributed service deployment in programmable networks, *Middleware 2004 companion workshop proceedings* (Kon, F. and de Lara, E. and Jacobsen, H. and de Camargo, R., eds.), pp. 256-261, 2004

4. N. Janssens, E. Steegmans, T. Holvoet, and P. Verbaeten, An agent design method promoting separation between computation and coordination, In *Proceedings of the 2004 ACM Symposium on Applied Computing* (Ossowski, S. and Menezes, R., eds.), pp. 456-461, 2004

5. N. Janssens, S. Michiels, T. Holvoet, and P. Verbaeten, A modular approach enforcing safe reconfiguration of producer-consumer applications, In *Proceedings of the 20th IEEE International Conference on Software Maintenance* (Azada, D., ed.), pp. 274-283, 2004

6. L. Desmet, N. Janssens, S. Michiels, F. Piessens, W. Joosen, and P. Verbaeten, Towards preserving correctness in self-managed software systems, In *Proceedings of the 2004 ACM SIGSOFT Workshop on Self-Managing Systems* (Garlan, D. and Kramer, J. and Wolf, A., eds.), pp. 34-38, 2004

7. N. Janssens, S. Michiels, T. Mahieu, and P. Verbaeten, Towards Transparent Hot-Swapping Support for Producer-Consumer Components. In *Proceedings of the 2nd International Workshop on Unanticipated Software Evolution* (Costanza, P. and Kniesel, G., eds.), pp. 9-16, 2003

8. E. Steegmans, T. Holvoet, N. Janssens, S. Michiels, Y. Berbers, P. Verbaeten, P. Valckenaers, and H. Van Brussel, Ant Algorithms in a Graph Environment: a Meta-scheme for Coordination and Control, *Artificial Intelligence and Applications* (Hanza, M.H., ed.), pp. 435-440, 2002

9. I. Sora, N. Janssens, P. Verbaeten, and Y. Berbers, A Component Composition Model to Support Unanticipated Customization of Systems, *Object-Oriented Technology - ECOOP 2002 Workshop Reader* (Hernandez, J. and Moreira, A., eds.), pp. 95, 2002

10. S. Michiels, L. Desmet, N. Janssens, T. Mahieu, and P. Verbaeten, Self-Adapting Concurrency: The DMonA Architecture, *Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)* (Garlan, D. and Kramer, J. and Wolf, A., eds.), pp. 43-48, 2002

11. N. Janssens, S. Michiels, T. Mahieu, and P. Verbaeten, Towards Hot-Swappable System Software: The DiPS/CuPS Component Framework, *Object-Oriented Technology - ECOOP 2002 Workshop Reader* (Hernandez, J. and Moreira, A., eds.), pp. 73-74, 2002

Contributions at international conferences, not published or only as abstract

1. S. Michiels, D. Walravens, N. Janssens, and P. Verbaeten, DiPSUnit: a JUnit Extension for the DiPS Framework, *3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering, XP2002*, Alghero, Sardinia, Italy, May 26-30, 2002, Universit di Cagliari, Free University of Bolzano-Bozen, BCI Italia

2. S. Michiels, D. Walravens, N. Janssens, and P. Verbaeten, DiPS: Filling the Gap between System Software and Testing, *Workshop on Testing in XP, WTiXP2002*, Alghero, Sardinia, Italy, May 27, 2002, Universit di Cagliari, Free University of Bolzano-Bozen, BCI Italia

Technical reports

1. F. Matthijs, N. Janssens, and P. Verbaeten, Automatic service composition: a case for active networks usability, Department of Computer Science, K.U.Leuven, Report CW 356, Leuven, Belgium, January, 2003

2. S. Michiels, D. Walravens, N. Janssens, and P. Verbaeten, DiPSUnit: an Extension of the JUnit Test Framework for DiPS, Department of Computer Science, K.U.Leuven, Report CW 333, Leuven, Belgium, February, 2002

3. N. Janssens, S. Michiels, and P. Verbaeten, DiPS/CuPS: a Framework for Runtime Customizable Protocol Stacks, Department of Computer Science, K.U.Leuven, Report CW 328, Leuven, Belgium, November, 2001

Curriculum vitae

Nico Janssens was born in Leuven (Belgium) on May 10, 1978. He received a Bachelor degree (Kandidaat Informatica) and a Master's degree (Licentiaat Informatica) in Informatics from the Katholieke Universiteit Leuven. In 2000 he graduated magna cum lauda with the thesis 'Optimization of a distributed robot controller' under the supervision of Prof. Pierre Verbaeten. The same year, he started working as a researcher in the DistriNet (Distributed systems and computer Networks) research group at K.U. Leuven's Department of Computer Science. During the first year, he worked on a project investigating how to improve QoS in the SmartMove (Acunia) architecture. From 2001 to 2005, he participated in a GOA project on Agents for Coordination and Control (AgCo2). Besides, from 2002 onwards, he also worked on an FWO project on Run-time Adaptable Component Infrastructure for active ad-hoc NetworkinG (RACING). Futhermore, Nico has been an external reviewer for the Journal on Software Maintenance and Evolution.

Dynamische Software-herconfiguratie in Programmeerbare Netwerken

Nederlandse samenvatting

Beknopte samenvatting

Programmeerbare netwerken maken het mogelijk voor niet-fabrikanten om de gebruikte netwerkinfrastructuur te herprogrammeren. Door de uitvoeringsomgeving van routers, firewalls, gateways, etc. toegankelijk te maken, kan het gedrag van de gebruikte netwerkinfrastructuur aanpast worden indien gewenst. Dit maakt van programmeerbare netwerken een interessante technologie voor het bouwen van adaptieve netwerken, evenals om de toenemende evolutie van netwerk-software te ondersteunen.

Tevens merken we op dat vele gedistribueerde toepassingen strenge beschikbaarheids- en performantievereisten opleggen aan de netwerkinfrastructuur, onder andere als gevolg van toenemende gebruikersverwachtingen. Het onderbreken van de netwerkcommunicatie om de software van een programmeerbare netwerkknoop bij te werken of aan te passen kan bijgevolg verstrekkende gevolgen hebben. Dit proefschrift focust zich daarom op dynamische herconfiguratie van netwerk-software – dat wil zeggen, de uitvoering van herconfiguraties zonder tijdelijk de werking van (een deel van) de netwerkinfrastructuur te onderbreken.

Of dergelijke dynamische herconfiguraties al dan niet zinvol zijn, hangt in sterke mate af van de doeltreffendheid en efficiëntie waarmee ze uitgevoerd worden. Bovendien is de realisatie van een correcte herconfiguratie met een minimale uitvoeringskost een complexe en foutgevoelige opgave (wat op zijn beurt het voordeel van dynamische herconfiguratie compromitteert). Dit alles illustreert de nood aan ondersteuning voor dynamische herconfiguraties die (1) de doeltreffende en efficiëntie herconfiguratie van netwerk-software coördineert en (2) de complexiteit van dergelijke herconfiguraties afschermt van de gebruikers die deze wensen uit te voeren.

Dit proefschrift stelt de NeCoMan (*Ne*twerk her*Co*nfiguarie *Ma*nagement) middleware voor om herconfiguraties uit te voeren in programmeerbare netwerken. Deze middleware coördineert het dynamisch toevoegen, verwijderen en vervangen van lokale en gedistribueerde netwerkdiensten. Het innovatieve aan deze middleware zit in de mogelijkheid om het herconfiguratieproces op maat te laten maken. Om dit te realiseren bevat de NeCoMan middleware verschillende algoritmes evenals een uitgebreide verzameling aanpassingen aan deze algoritmes. Dit laat de NeCoMan middleware toe om elke herconfiguratie op maat te maken vertrekkende van (1) een declaratieve beschrijving van de herconfiguratie die moet worden uitgevoerd en (2) een specificatie van de eigenschappen van de betrokken netwerkdiensten en van de herconfiguratiesemantiek.

Tot slot vatten we de belangrijkste bijdragen van dit proefschrift kort samen. Naast de voorstelling van een middleware die de dynamische herconfiguratie van programmeerbare netwerkknopen coördineert en de validatie hiervan, omvat dit proefschrift een uitgebreide analyse van de coördinatie die vereist is om correcte lokale en gedistribueerde herconfiguraties uit te voeren. Verder stelt dit proefschrift voor om ondersteuning voor dynamische herconfiguratie aanpasbaar te maken, dit in tegenstelling tot bestaande initiatieven waarbij typisch een niet-aanpasbaar algoritme wordt gebruikt.

1 Inleiding

Computernetwerken staan in voor het overbrengen van berichten tussen twee of meerdere computers die deel uitmaken van een gedistribueerd systeem. Hierbij is de beschikbaarheid en performantie van het netwerk cruciaal, aangezien dit de werking van dergelijke gedistribueerde systemen sterk compromitteert. Gedurende lange tijd werd getracht aan deze performantie- en beschikbaarheidsvereisten te voldoen door het netwerk zo simpel mogelijk te houden. De (netwerk-)functies van het Internet, bijvoorbeeld, zijn lang beperkt gebleven tot routering, de controle van congestie en simpele ondersteuning om kwaliteitsgaranties te bieden (QoS) [16, 21, 3]. Daarenboven zijn deze netwerkfuncties typisch transparant en ontoegankelijk voor eindgebruikers en (gedistribueerde) toepassingen. Deze tussenliggende knopen in een netwerk, bijvoorbeeld (zoals routers en switches), vormen typisch verticaal geïntegreerde systemen waarvan de functies enkel door de fabrikant ge-(her)programmeerd kunnen worden.

Sinds het midden van de jaren 90 hebben verschillende initiatieven als doel gesteld (1) de netwerkinfrastructuur meer open te maken voor wijzigingen en (2) de programmeerbaarheid ervan te verhogen [4]. Door aanpassen van de netwerksoftware op zowel de tussenliggende knopen als de eindpunten mogelijk te maken, evolueert het netwerk naar een volledig programmeerbare omgeving. Op die manier kan het ganse netwerk aangepast worden aan wijzigende omstandigheden en/of gebruikersvereisten, om zodoende efficiënter te opereren [16].

Tevens merken we op dat vele gedistribueerde toepassingen strenge beschikbaarheids- en performantievereisten opleggen aan de netwerkinfrastructuur, onder andere als gevolg van toenemende gebruikersverwachtingen. Het onderbreken van de netwerkcommunicatie om de software van een programmeerbare netwerkknoop bij te werken of aan te passen kan bijgevolg verstrekkende gevolgen hebben. Dit proefschrift focust zich daarom op dynamische herconfiguratie van netwerk-software – dat wil zeggen, de uitvoering van herconfiguraties zonder tijdelijk de werking van (een deel van) de netwerkinfrastructuur te onderbreken.

Of dergelijke dynamische herconfiguraties al dan niet zinvol zijn, hangt in sterke mate af van de doeltreffendheid en efficiëntie waarmee ze uitgevoerd worden. Bovendien is de realisatie van een correcte herconfiguratie met een minimale uitvoeringskost een complexe en foutgevoelige opgave (wat op zijn beurt het voordeel van dynamische herconfiguratie compromitteert). Dit alles illustreert de nood aan ondersteuning voor dynamische herconfiguraties die (1) de doeltreffende en efficiëntie herconfiguratie van netwerk-software coördineert en (2) de complexiteit van dergelijke herconfiguraties afschermt van gebruikers die deze wensen uit te voeren.

Dit proefschrift stelt de NeCoMan (*Ne*twerk her*Co*nfiguarie *Ma*nagement) middleware voor om herconfiguraties uit te voeren in programmeerbare netwerken. Deze middleware coördineert het dynamisch toevoegen, verwijderen en vervangen van lokale en gedistribueerde netwerkdiensten [10, 11, 12]. Daarbij moet deze middleware aan de volgende vereisten voldoen:

- **Correcte herconfiguraties.** NeCoMan moet steeds correcte herconfiguraties uitvoeren, om zodoende te voorkomen dat de werking van het netwerk faalt als gevolg van het herconfiguratieproces.
- **Beperkte herconfiguratiekost.** NeCoMan moet elke herconfiguratie op maat kunnen maken, hierbij rekening houdend met (1) de eigenschappen van de betrokken netwerkdiensten en (2) de herconfiguratiesemantiek. Op die manier kan NeCoMan de impact van een herconfiguratie op de beschikbaarheid en performantie van het netwerk beperken.
- **Beperkte openheid.** NeCoMan moet de complexiteit van dynamische herconfiguraties van netwerk-software afschermen door middel van een beperkte herconfiguratie-API. Een gebruiker wordt hierbij verondersteld om alleen te beschrijven *welke* herconfiguratie NeCoMan moet uitvoeren, zonder te beschrijven *hoe* deze moet worden uitgevoerd.
- **Herbruikbaarheid.** NeCoMan moet gebruikt kunnen worden boven verschillende knooparchitecturen (zoals Click [13], DiPS+ [19, 20] en Netkit [6]). Om dit mogelijk te maken moeten de afhankelijkheden tussen de NeCoMan middleware en de betrokken knopen beperkt worden gehouden.

2 Positionering

Alvorens in de volgende secties dieper in te gaan op de NeCoMan middleware zelf, bakenen we eerst duidelijk het bereik van dit proefschrift af. We doen dit door af te lijnen hoe dit onderzoek zich positioneert in het domein van (1) programmeerbare netwerken, (2) dynamische software-herconfiguratie en (3) beheerssystemen voor dynamische herconfiguratie.

2.1 Programmeerbare netwerken

In de laatste 10 jaar zijn verschillende aspecten van programmeerbare netwerken onderzocht. In dit brede spectrum richt voorliggend onderzoek zich op *dynamische software-herconfiguratie* van *extern programmeerbare netwerkknopen*.

Bij *extern programmeerbare netwerkknopen* worden de netwerkprogramma's en de datapakketten via logisch en/of fysisch gescheiden communicatiekanalen vervoerd[1]. Beide worden bijgevolg onafhankelijk van elkaar op de programmeerbare knopen verwerkt. Hoewel we niet beweren dat deze aanpak voor netwerk programmering dominant zal worden in gebruik, zijn we toch overtuigd van het potentieel

[1] Deze netwerkprogramma's variëren van (aan een eind van het spectrum) een scalair argument dat een bepaalde functie op de knopen oproept, tot (aan de andere kant van het spectrum) mobiele code geschreven in een Tuning-complete taal die op de knopen geïnterpreteerd en uitgevoerd wordt [3].

dat deze aanpak biedt om knopen te programmeren op een gecontroleerde en veilige manier.

Verder stellen we vast dat de meerderheid van programmeerbare knoop-architecturen geen dynamische software-herconfiguratie ondersteunt[2]. Dergelijke knoop-architecturen laten wel toe nieuwe netwerkdiensten in gebruik te nemen, maar ondersteunen geen herconfiguratie van diensten die reeds gebruikt worden (onder andere omdat deze zich in een toestand bevinden die verschillend is van de initiële uitvoeringstoestand) [7]. Hierdoor kunnen deze architecturen niet omgaan met wijzigingen ten gevolge van software-evolutie.

2.2 Dynamische software-herconfiguratie

In het domein van dynamische software-herconfiguratie richt dit onderzoek zich op *dynamische compositionele wijzigingen* in *buis-en-filter software-architecturen*. In [18] beschrijven McKinley en zijn collega's dynamische compositionele wijzigingen als volgt:

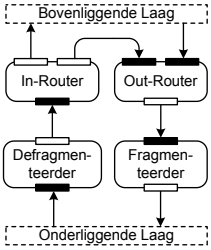
> *Compositionele wijzigingen laten toe algoritmes of structurele componenten te vervangen door nieuwe varianten. Deze flexibiliteit gaat verder dan het aanpassen van programmavariabelen of het selecteren van een nieuwe strategie. Compositionele adaptatie maakt het immers mogelijk om de compositie van de gebruikte software te veranderen tijdens de uitvoering. Hierdoor kan een applicatie omgaan met aangelegenheden die niet waren voorzien tijdens de ontwikkelingsfaze.*

Compositionele wijziging maakt het bijgevolg mogelijk om dynamische software-herconfiguratie in programmeerbare netwerken te realiseren. Deze techniek laat immers toe om netwerk-software die reeds in gebruik is genomen, aan te passen door software-componenten toe te voegen, te vervangen of te verwijderen. Dit vereist echter wel dat de software-architectuur van de programmeerbare knopen voorzien is op dergelijke aanpassingen.

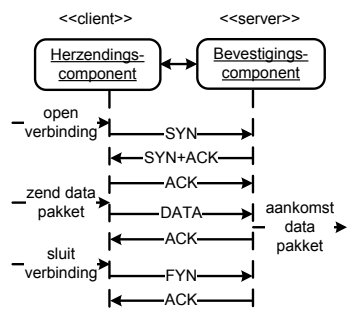
Een architectuur die hiervoor in aanmerking komt is de *buis-en-filter (pipe-and-filter) software-architectuur*. Zoals Shaw en Garlan in [22] beschrijven, dwingt deze architectuur de programmeur om onafhankelijke componenten te ontwikkelen (filters) die inkomende gegevens verwerken en de resultaten hiervan beschikbaar maken voor andere filters. Deze componenten worden dan met elkaar verbonden via connectoren (buizen) om zo een functioneel systeem te verkrijgen. Merk op dat deze stijl op een natuurlijke manier kan toegepast worden op netwerk-software. Een protocolstapel, bijvoorbeeld, omvat verschillende functies (zoals fragmentatie, routering, compressie, enz.) die achtereenvolgens worden uitgevoerd op ontvangen pakketten.

Ter illustratie hiervan verwijzen we naar Figuur 1, waarin de functionaliteit van een simpele DiPS+ router wordt voorgesteld (zoals beschreven door Matthijs

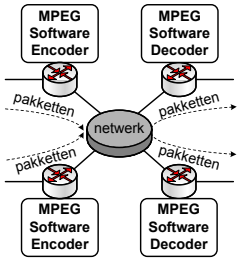
[2] Uitzonderingen hierop zijn Click [13], Ensemble [23], Cactus [5], Netkit [6] en de dynamisch herconfigureerbare protocolstapel van Lee [15].



Figuur 1: Voorstelling van een DiPS+ router.



Figuur 2: Protocol van een dienst gebaseerd op TCP.



Figuur 3: Configuratie waarbij MPEG-codering in het netwerk wordt gebruikt.

in [17]). Deze router werd gebouwd in het DiPS+ raamwerk (framework) [19, 20], een buis-en-filter raamwerk dat werd ontwikkeld binnen DistriNet om aanpasbare protocolstapels te bouwen. Dit raamwerk werd eveneens in voorliggend onderzoek gebruikt ter validatie van de NeCoMan middleware.

2.3 Beheerssystemen voor dynamische herconfiguratie

Beheerssystemen voor dynamische herconfiguratie (zoals voorgesteld in [14, 24, 1, 8]) trachten de complexiteit van dynamische software-herconfiguratie van de applicatieprogrammeur af te schermen op een applicatie-onafhankelijke manier. Behalve het raamwerk van Hillman en Warren (beschreven in [8]) bevatten deze systemen typisch slechts één enkel niet-aanpasbaar herconfiguratie-algoritme. In geval van programmeerbare netwerken – die aan strikte performantievereisten moeten voldoen – is deze aanpak echter nefast voor de efficiëntie van een aantal herconfiguraties, aangezien het niet mogelijk is om elke herconfiguratie op maat te maken. Daarom stellen we dat het mogelijk moet zijn voor het beheerssysteem om het gebruikte algoritme aan te passen aan de eigenschappen van de betrokken netwerkdiensten en van de herconfiguratiesemantiek. Het beheerssysteem moet bijgevolg aanpasbaar zijn.

2.4 Netwerkdiensten

Tot slot beschrijven we kort de eigenschappen van de netwerkdiensten die in aanmerking komen voor herconfiguratie door de NeCoMan middleware.

2.4.1 Geïsoleerde netwerkdiensten

Geïsoleerde netwerkdiensten opereren onafhankelijk van elkaar evenals van andere netwerkdiensten. Als voorbeeld van dergelijke netwerkdiensten verwijzen we naar een filter die willekeurig pakketten verwerpt op een verzadigde knoop. Verder gaan we ervan uit dat deze diensten reactief werken. Ze reageren alleen indien pakketten verwerkt moeten worden en zullen bijgevolg nooit autonoom opereren. Merk echter wel op dat dit niet impliceert dat de componenten die deze diensten implementeren geen "actieve objecten" mogen gebruiken[3].

2.4.2 Gedistribueerde netwerkdiensten

Naast geïsoleerde netwerkdiensten is de NeCoMan middleware eveneens in staat om herconfiguraties uit te voeren waarbij gedistribueerde netwerkdiensten betrokken zijn. Voorbeelden van dergelijke netwerkdiensten zijn codering, compressie, fragmentatie, betrouwbaarheid en encryptie. Deze diensten hebben de volgende eigenschappen gemeenschappelijk:

Gedistribueerde afhankelijkheden. Deze diensten worden geïmplementeerd door gedistribueerde componenten die moeten samenwerken om de betreffende dienst uit te voeren en bijgevolg sterk afhankelijk zijn van elkaar om de dienst correct uit te voeren. Deze gedistribueerde afhankelijkheden kunnen worden afgeleid uit het gebruikte communicatieprotocol, dat specificeert hoe een component zijn tegenhanger aanspreekt. Het protocol dat geïllustreerd wordt in Figuur 2, bijvoorbeeld, toont aan hoe beide componenten van een dienst gebaseerd op TCP moeten samenwerken (en dus van elkaar afhangen) om deze dienst correct uit te voeren[4].

Client-server gebaseerde samenwerking. De componenten van de gedistribueerde netwerkdiensten die in aanmerking komen voor herconfiguratie werken samen volgens het client-server samenwerkingsmodel [2]. Hierbij initieert een client-proces de aanvraag tot dienstverlening die door een server-proces wordt verwerkt. Toegepast op gedistribueerde netwerkdiensten start het client-proces dus de uitvoering van het gemeenschappelijke communicatieprotocol, terwijl het server-proces hierop reageert (eveneens conform dat protocol). De herzendingscomponent van de dienst in Figuur 2, bijvoorbeeld, bevat het client-proces terwijl het bijhorende server-proces vervat zit in de bevestigingscomponent.

Reactief gedrag. Gelijkaardig als voor geïsoleerde diensten zijn de componenten die gedistribueerde diensten bevatten reactief.

[3] Een actief object voert bepaalde taken uit in zijn eigen uitvoeringsdraad (thread).

[4] Merk op dat deze dienst alleen dient ter illustratie. Het bijhorend protocol is niet volledig conform de RFC van TCP.

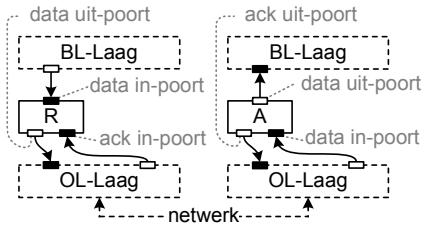
Asynchrone gebufferde communicatie. De componenten die een gedistribueerde netwerkdienst bevatten, communiceren onderling via asynchrone gebufferde uitwisseling van pakketten. Bijgevolg zullen de betrokken componenten de ontvangst van een pakket niet afwachten alvorens een nieuw exemplaar te versturen. Het netwerk fungeert bijgevolg als een gedeelde buffer die door de samenwerkende componenten wordt gebruikt om berichten uit te wisselen.

Aantal betrokken componenten. Hoewel de gedistribueerde netwerkdiensten die in aanmerking komen voor herconfiguratie conceptueel uit twee samenwerkende componenten bestaan, zullen in bepaalde configuraties meerdere instanties van deze componenten moeten worden gebruikt. Om dit te illustreren verwijzen we naar de opstelling in Figuur 3, waarbij MPEG-codering wordt gebruikt rond een traag draadloos subnet. Om een correct werking van dit netwerk te garanderen, moeten alle pakketten die dit subnet binnenkomen en verlaten, gecodeerd en gedecodeerd worden. Dit betekent dat voor deze opstelling meerdere coderende- en decoderende-componenten moeten worden gebruikt.

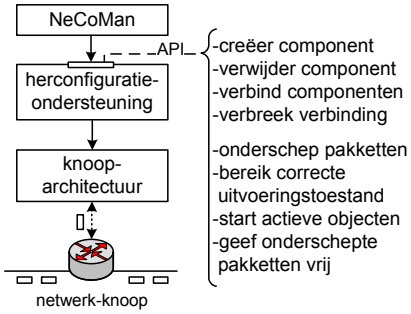
2.4.3 Betrouwbaarheidsdienst

In het verdere verloop van deze samenvatting illustreren we NeCoMan's herconfiguratie-algoritmes aan de hand van een betrouwbaarheidsdienst. Zoals Figuur 4 illustreert, bestaat deze betrouwbaarheidsdienst uit een herzendingscomponent (voorgesteld als component R) en een bevestigingscomponent (voorgesteld als component A). Om een pakket betrouwbaar te verzenden, moet dit worden afgeleverd bij de "data in-poort" van component R. Deze houdt dan een kopie bij van dit pakket, voegt een volgnummer toe en start de tijdopnemer die (eventuele) herzendingen van dit pakket zal initiëren. Vervolgens levert R dit pakket (en alle latere herzendingen) via zijn "data uit-poort" af aan de onderliggende laag (voorgesteld als *OL-Laag*). Wanneer dit pakket zijn bestemming bereikt, wordt het doorgegeven aan de "data in-poort" van component A. Indien alles in orde is met dit pakket geeft A het door via zijn "data uit-poort" aan de bovenliggende laag (voorgesteld als *BL-Laag*) en stuurt een bevestigingspakket naar de zender. Dit pakket bereikt zijn bestemming via A's "ack uit-poort" en R's "ack in-poort". Bij het verwerken van dit pakket verwijdert R het bevestigde datapakket en stopt de bijhorende tijdopnemer.

Merk bovendien op dat we bij componenten van gedistribueerde netwerkdiensten een onderscheid maken tussen *dienst-externe* en *dienst-interne* poorten. Zo gebruiken client- en server-processen de dienst-interne poorten van de betrokken componenten om het gemeenschappelijke communicatieprotocol uit te voeren. In het geval van de betrouwbaarheidsdienst betekent dit dat de "data uit-poort" en "ack in-poort" van component R, evenals de "data in-poort" en "ack uit-poort" van component A, fungeren als dienst-interne poorten. Dienst-externe poorten, daarentegen, verbinden de betrokken processen met andere componenten in de protocolstapel die niet deelnemen aan de uitvoering van het communicatieprotocol.



Figuur 4: Voorstelling van de betrouwbaarheidsdienst



Figuur 5: Koppeling tussen knoopspecifieke herconfiguratie-ondersteuning en de NeCoMan middleware

Dit is bijvoorbeeld het geval voor de "data in-poort" van component R en de "data uit-poort" van component A.

3 Lokale herconfiguraties

Om de herbruikbaarheid van een herconfiguratie-middleware (zoals NeCoMan) mogelijk te maken moet de koppeling met de onderliggende knooparchitectuur zo laag mogelijk worden gehouden. NeCoMan bevat daarom zelf geen knoopspecifieke herconfiguratiefunctionaliteit, maar coördineert de uitvoering van een aantal algemene operaties. Zoals geïllustreerd in Figuur 5 moeten deze operaties door de "herconfiguratie-ondersteuning" van de betrokken knooparchitecturen worden aangeboden en uitgevoerd.

3.1 Herconfiguratie-ondersteuning

De herconfiguratie-ondersteuning van een programmeerbare knoop moet de (knoopspecifieke) implementatie van acht operaties aanbieden. Vier van deze operaties dienen om een software-compositie aan te passen, de overige hebben betrekking op de uitvoeringstoestand van de betrokken knoop.

Zoals Kramer en Magee beschrijven in [14] moeten wijzigingen aan een softwarecompositie uitgedrukt worden in functie van diens structuur. De herconfiguratieondersteuning van een programmeerbare knoop wordt daarom verwacht operaties aan te bieden om (1) componenten aan te maken, (2) deze te verbinden met andere componenten, (3) componenten los te koppelen van andere componenten en (4) ze te verwijderen uit de compositie. Op deze manier assisteert de herconfiguratieondersteuning van een knooparchitectuur de NeCoMan middleware in het wijzigen van een software-compositie (zie Figuur 5).

Daarnaast moet deze herconfiguratie-ondersteuning de NeCoMan middleware eveneens bijstaan bij het beheren van de uitvoeringstoestand van de betrokken knoop. De NeCoMan middleware verwacht hiervoor van vier operaties gebruik te kunnen maken (zie opnieuw Figuur 5). De eerste operatie is verantwoordelijk voor het onderscheppen van pakketten en zal aangeroepen worden om lokaal in een compositie de verwerking van pakketten tijdelijk te onderbreken. De tweede operatie zorgt ervoor dat de betrokken componenten een correcte uitvoeringstoestand bereiken. In deze toestand zal de correcte werking van het netwerk niet gecompromitteerd worden door de herconfiguratie. Deze toestand kan bereikt worden door de betrokken componenten te monitoren of door de uitvoeringstoestand van de oude componenten over te dragen naar de nieuwe componenten. De derde operatie heeft als taak de actieve objecten (indien aanwezig) van de nieuwe componenten te starten. Bij het oproepen van de laatste operatie, tenslotte, zullen de pakketten die bij de uitvoering van de eerste operatie zijn onderschept, terug worden vrijgegeven.

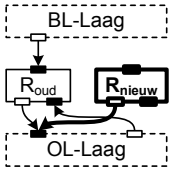
3.2 Lokale herconfiguratie van gedistribueerde netwerkdiensten

NeCoMan gebruikt twee (basis)algoritmes om lokale herconfiguraties uit te voeren. Het eerste algoritme coördineert herconfiguraties waarbij een component van een gedistribueerde netwerkdienst betrokken is. Het andere algoritme daarentegen, coördineert herconfiguraties van geïsoleerde netwerkdiensten. Beide algoritmes hebben als doel een ruim aantal herconfiguraties correct uit te kunnen voeren en houden bijgevolg geen rekening met de eigenschappen van de betrokken diensten en/of de herconfiguratiesemantiek.

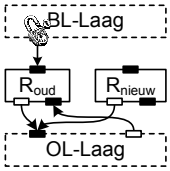
In het vervolg van deze sectie beschrijven we het algoritme dat NeCoMan gebruikt om een component van een gedistribueerde netwerkdienst op een bepaalde knoop te vervangen. We illustreren dit algoritme met de vervanging van een oude herzendingscomponent (voorgesteld als R_{oud}) door een nieuwe (voorgesteld als R_{nieuw}), die beide deel uitmaken van de gedistribueerde betrouwbaarheidsdienst. Dit algoritme zal achtereenvolgens (1) de nieuwe component installeren, (2) de oude component beëindigen, (3) de nieuwe component activeren en (4) de oude component verwijderen.

3.2.1 Installatie van nieuwe component

Het vervangen van R_{oud} door R_{nieuw} begint met de installatie van R_{nieuw} op de betrokken knoop. NeCoMan roept hiervoor de herconfiguratie-ondersteuning van deze knoop op om R_{nieuw} aan te maken. Vervolgens draagt NeCoMan deze knoop op om de nieuwe herzendingscomponent reeds deels te integreren in diens protocolstapelcompositie. Vermits de nieuwe component nog niet in gebruik wordt genomen in deze fase, beperkt deze integratie zich tot het linken van de uit-poorten van de nieuwe component. In het geval van R_{nieuw} betekent dit dat de “data uit-poort”



Figuur 6: Installatie van R_{nieuw}



Figuur 7: Beëindigen van R_{oud}

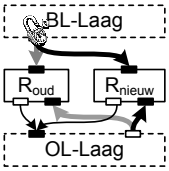
verbonden wordt met de in-poort van de onderliggende protocollaag. Figuur 6 illustreert de compositie van de betrokken knoop na deze installatiefaze. Nieuw aangemaakte componenten en verbindingen worden hierbij voorgesteld door dikkere zwarte rechthoeken en lijnen.

3.2.2 Beëindiging van oude component

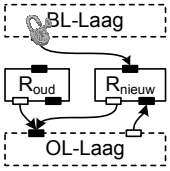
Vervolgens wordt R_{oud} in een toestand gebracht waarbij de correcte werking van het netwerk niet gecompromitteerd wordt door het verwijderen van deze component. NeCoMan roept hiervoor de herconfiguratie-ondersteuning van de betrokken knoop eerst op om alle pakketten gericht aan R_{oud} te onderscheppen. Zoals geïllustreerd in Figuur 7 betekent dit dat er (voor deze herconfiguratie) geen nieuwe pakketten meer afgeleverd worden aan de externe in-poort van R_{oud} (de "data in-poort"). Vervolgens draagt NeCoMan de betrokken knoop op om een toestand te bereiken waarin R_{oud} kan verwijderd worden zonder de correcte werking van het netwerk te compromitteren. We veronderstellen dat voor deze herconfiguratie de betrokken knoop R_{oud} controleert tot alle verzonden pakketten bevestigd zijn. Vervolgens stopt de knoop de tijdopnemer (timer) van R_{oud} en draagt het laatst gebruikte volgnummer over naar R_{nieuw}.

3.2.3 Activering van nieuwe component

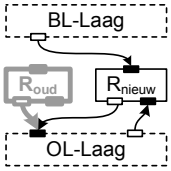
Na het beëindigen van de oude component kan de nieuwe veilig geactiveerd worden. Als eerste stap van deze activering roept NeCoMan de knoop op om de verbindingen van alle betrokken in-poorten aan te passen zodoende dat pakketten aan de nieuwe component zullen worden afgeleverd. Zoals geïllustreerd wordt in Figuur 8, houdt dit in dat (voor het vervangen van R_{oud} door R_{nieuw}) de "ack in-poort" en "data in-poort" van R_{oud} worden losgekoppeld van de uit-poorten die behoren tot de onder- en bovenliggende laag (dit wordt voorgesteld door dikkere grijze lijnen). Vervolgens worden deze uit-poorten verbonden met de "ack in-poort" en "data in-poort" van R_{nieuw}. Van zodra dit gerealiseerd is, draagt NeCoMan de betrokken knoop op om het herzendingsproces van R_{nieuw} op te starten, waardoor de tijdopnemer van R_{nieuw} begint te lopen. Vanaf nu kan R_{nieuw} door de knoop in gebruik genomen worden. NeCoMan voltooit dan de activering van R_{nieuw} door de betrokken knoop op te roepen alle onderschepte pakketten vrij te geven (zie Figuur 9).



Figuur 8: Verbinden van de in-poorten van R_{nieuw}



Figuur 9: Vrijgeven van onderschepte pakketten



Figuur 10: Verwijdering van R_{oud}

3.2.4 Verwijdering van oude component

Tenslotte wordt de oude (herzendings)component verwijderd van de betrokken knoop, zoals geïllustreerd in Figuur 10. Hiervoor moeten de bestaande verbindingen van deze component met de rest van de protocolstapel eerst worden verbroken. NeCoMan beveelt de betrokken knoop daarom om alle verbindingen met uit-poorten die behoren tot de oude component te verbreken. Vervolgens draagt NeCoMan de knoop op om de oude component te verwijderen.

3.3 Lokale herconfiguratie van geïsoleerde netwerkdiensten

Naast het vervangen van componenten die deel uitmaken van een gedistribueerde netwerkdienst, coördineert NeCoMan ook het toevoegen, verwijderen en vervangen van componenten die geïsoleerde netwerkdiensten bevatten. Vermits deze componenten niet samenwerken met andere componenten valt het onderscheid tussen dienst-externe en dienst-interne poorten weg. Dit heeft op zijn beurt een effect op de manier waarop de vier fazes van een herconfiguratie worden geïmplementeerd. NeCoMan gebruikt daarom een licht gewijzigd algoritme om deze herconfiguraties uit te voeren.

4 Aanpassingen bij lokale herconfiguraties

Door enkel deze basisalgoritmes aan te bieden kan NeCoMan echter niet voor elke herconfiguratie een beperkte herconfiguratiekost garanderen. Voor sommige herconfiguraties kan deze herconfiguratiekost immers verminderd worden door het gebruikte algoritme aan te passen. Verder moet dit soort aanpassingen kunnen worden gerealiseerd met minimale inbreng van de gebruiker, dit om de kans op fouten te beperken.

NeCoMan voorziet daarom een aantal voorgedefinieerde aanpassingen voor beide basisalgoritmes. Deze aanpassingen zijn ontstaan door na te gaan wanneer er acties veilig van volgorde kunnen gewisseld worden of veilig verwijderd kunnen worden. Van al deze aanpassingen zijn enkel diegene bewaard gebleven die de kost van

een herconfiguratie verminderen zonder de correctheid ervan in gevaar te brengen. Om de contributie van de gebruiker te beperken, past NeCoMan deze aanpassingen toe op basis van de eigenschappen van de betrokken netwerkdiensten en van de herconfiguratiesemantiek. In het verloop van deze sectie schetsen we kort de belangrijkste van deze aanpassingen.

4.1 Activering nieuwe component alvorens de oude te beëindigen

Een eerste aanpassing is gericht op het omkeren van de activerings- en beëindigingsfaze, waardoor de nieuwe component geactiveerd zal worden alvorens de oude beëindigd is. In dit geval zullen ook geen pakketten meer onderschept worden om de oude component te beëindigen. Van zodra de nieuwe component in gebruik genomen is worden pakketten immers exclusief bij de in-poorten van de (nieuwe) component afgeleverd. Dit alles maakt dat de communicatie-onderbreking die het beëindigen van de oude component veroorzaakt (bij de uitvoering van de basisalgoritmes), wordt verminderd.

Om deze aanpassing veilig uit te kunnen voeren, moet er echter aan drie voorwaarden voldaan zijn. Ten eerste mag de oude component geen toestandsinformatie bevatten. Indien dit toch het geval is, moet de oude component immers eerst beëindigd worden om te voorkomen dat de herconfiguratie inconsistente uitvoeringstoestanden voortbrengt. Ten tweede moet de nieuwe component in staat zijn de uitvoering van protocols die reeds geïnitieerd werden, verder af te handelen. Wanneer NeCoMan de nieuwe component activeert alvorens de oude beëindigd is, is er immers geen kennis over de toestand waarin deze protocols zich bevinden op het moment dat de nieuwe component in gebruik wordt genomen. Tenslotte kan deze aanpassing enkel toegepast worden indien het netwerk of de applicaties die hiervan gebruik maken, niet vereisen dat alle pakketten aankomen in de volgorde waarin ze verstuurd zijn[5]. Wanneer NeCoMan de oude component beëindigd terwijl de nieuwe reeds in gebruik is genomen, zullen beide componenten (tijdelijk) in parallel functioneren. Bijgevolg bestaat de kans dat pakketten die door de nieuwe component worden verwerkt, hun bestemming eerder bereiken dan pakketten die nog door de oude component worden afgehandeld.

4.2 Geen beëindiging van oude component

Een volgende aanpassing houdt in dat de oude component niet meer wordt beëindigd alvorens hem te verwijderen. Indien het netwerk hiermee kan omgaan (bijvoorbeeld door zichzelf te herstellen wanneer inconsistenties optreden) zal dit de communicatie-onderbreking die beide basisalgoritmes veroorzaken, opnieuw verminderen.

[5]Dit is bijvoorbeeld het geval wanneer een toestandsloze component (zoals een compressie-component) wordt gebruikt in een TCP netwerk. TCP garandeert immers dat alle pakketten hun bestemming bereiken in de volgorde waarin ze werden verzonden.

Om deze aanpassing veilig uit te kunnen voeren, moet aan volgende voorwaarden voldaan zijn. Ten eerste moet het netwerk of de gedistribueerde applicaties kunnen omgegaan met het verlies van pakketten (bijvoorbeeld door gebruik te maken van betrouwbaarheidsprotocollen zoals TCP). Wanneer de oude component verwijderd wordt alvorens beëindigd te zijn, zullen de pakketten die op dat moment in verwerking waren immers verloren gaan. Ten tweede moet de nieuwe component in staat zijn de uitvoering van protocols die reeds geïnitieerd werden, verder af te handelen. De rede hiervoor is dezelfde als bij het omwisselen van de activerings- en beëindigingsfaze. Tenslotte mogen inconsistente uitvoeringstoestanden de correcte werking van het netwerk niet in gevaar brengen. Het netwerk moet deze inconsistente toestanden dus tolereren of moet zelf in staat zijn consistentie te herstellen.

4.3 Toevoeging en verwijdering van geïsoleerde dienst

Het toevoegen (in plaats van vervangen) van een geïsoleerde dienst aan een programmeerbare knoop, maakt de uitvoering van een aantal herconfiguratie-acties overbodig. In tegenstelling tot het vervangen van een geïsoleerde dienst moet bij deze herconfiguratie immers geen oude component verwijderd worden. Verder is het eveneens overbodig de oude componenten te beëindigen. NeCoMan slaat deze acties daarom over in dit geval.

Het verwijderen van een geïsoleerde dienst maakt dan weer andere herconfiguratie-acties overbodig. Zo moeten er geen tijdopnemers voor de nieuwe dienst gestart worden en moeten er geen nieuwe componenten geïnstalleerd worden. NeCoMan zal deze acties bijgevolg overslaan.

5 Gedistribueerde herconfiguraties

Naast lokale herconfiguraties coördineert NeCoMan eveneens de uitvoering van gedistribueerde herconfiguraties. NeCoMan gebruikt hiervoor opnieuw twee basisalgoritmes die dienen om een ruim aantal herconfiguraties uit te voeren (al dan niet met beperkte herconfiguratiekost). Deze algoritmes verschillen van elkaar door de manier waarop de betrokken gedistribueerde netwerkdienst wordt beëindigd.

5.1 Beëindiging van gedistribueerde diensten

NeCoMan ondersteunt twee manieren om de oude gedistribueerde netwerkdienst te beëindigen[6]. In een eerste aanpak wordt er gewacht tot de componenten van de oude dienst al hun activiteiten hebben afgerond, dat wil zeggen tot er een rustpunt is bereikt. In een tweede aanpak wordt de activiteit van elke component afzonderlijk stopgezet en wordt de toestand van deze componenten overgedragen naar de nieuwe componenten.

[6]Herinner dat NeCoMan dit niet zelf realiseert, maar in plaats daarvan de herconfiguratie-ondersteuning van de betrokken knopen aanroept om de oude componenten te beëindigen.

5.1.1 Bereiken van rustpunt

Kramer en Magee beschrijven in [14] dat een knoop in een gedistribueerd systeem een rustpunt (quiescence) bereikt indien (1) deze knoop niet participeert in transacties die hij zelf heeft gestart, (2) deze knoop geen nieuwe transacties zal starten, (3) deze knoop niet participeert in transacties die door andere knopen werden gestart en (4) deze knoop niet zal deelnemen aan transacties die door andere knopen zullen opgestart worden. Bij gedistribueerde netwerkdiensten wordt dit rustpunt bereikt wanneer (1) elke uitvoering van het gebruikte protocol beëindigd is en (2) geen nieuwe uitvoeringen van dit protocol geïnitieerd zullen worden tot na het afronden van de herconfiguratie. De betrouwbaarheidsdienst, bijvoorbeeld, bereikt een rustpunt wanneer de oude herzendingscomponent geen nieuwe pakketten meer kan verzenden en alle verstuurde pakketten bevestigd zijn. Op dit moment is een consistente toestand bereikt, waardoor beide betrouwbaarheidscomponenten verwijderd kunnen worden zonder de werking van het netwerk te compromitteren.

5.1.2 Toestandsoverdracht

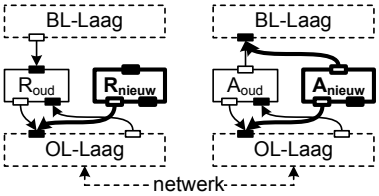
Wachten tot elke uitvoering van het gebruikte protocol beëindigd is, kan echter lang duren, vooral wanneer verschillende uitvoeringen van het gebruikte protocol tegelijkertijd actief zijn. Dit is uiteraard nefast voor de duur van de herconfiguratie evenals voor de veroorzaakte communicatie-onderbreking. NeCoMan's tweede basisalgoritme steunt daarom op toestandsoverdracht om de componenten van een gedistribueerde dienst onmiddellijk te kunnen beëindigen. Hierbij wordt elke component van de betrokken dienst afzonderlijk stopgezet zonder rekening te houden met de toestand waarin de gebruikte protocols zich op dat moment bevinden. Vervolgens wordt de uitvoeringstoestand van elk van deze componenten overgedragen naar de nieuwe componenten om zodoende consistentie te herstellen. Het is duidelijk dat deze aanpak alleen kan toegepast worden bij het vervangen van diensten en dus niet bij het toevoegen of verwijderen ervan.

5.2 Gedistribueerde herconfiguratie met bereiken van rustpunt

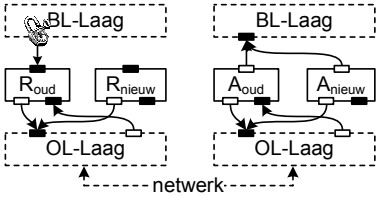
NeCoMan's eerste basisalgoritme is van toepassing indien de betrokken knopen wachten tot de componenten van de oude dienst een rustpunt bereiken. We illustreren dit algoritme met de vervanging van de volledige betrouwbaarheidsdienst door een nieuwe versie.

5.2.1 Installatie van nieuwe dienst

NeCoMan start deze herconfiguratie met de installatie van de nieuwe betrouwbaarheidscomponenten op de betrokken knopen. Gelijkaardig als bij lokale herconfiguraties draagt NeCoMan elk van de knopen op om de nieuwe componenten aan te



Figuur 11: Installatie van R_{nieuw} en A_{nieuw}



Figuur 12: Beëindigen van R_{oud} en A_{oud}

maken en hun uit-poorten te verbinden. Figuur 11 illustreert dit aan de hand van de installatie van R_{nieuw} en A_{nieuw}. Merk op dat NeCoMan deze gedistribueerde installatie niet synchroniseert vermits nieuwe componenten niet in gebruik worden genomen tijdens hun installatie.

5.2.2 Beëindiging van oude dienst

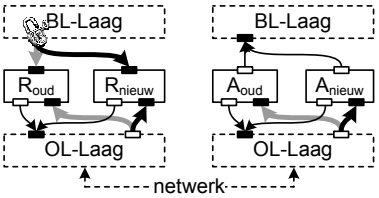
Vervolgens moeten de oude componenten in hun rusttoestand worden gebracht. Om dit te realiseren beveelt NeCoMan de betrokken knopen eerst alle pakketten te onderscheppen die gericht zijn aan de externe in-poorten van deze componenten. Op die manier wordt voorkomen dat de oude dienst nog nieuwe protocoltransacties initieert. In het geval van de betrouwbaarheidsdienst wordt dit gerealiseerd door geen pakketten meer af te leveren aan de "data in-poort" van R_{oud} (zie Figuur 12).

Van zodra deze pakketten onderschept zijn, draagt NeCoMan de betrokken knopen op om de client-processen van de oude dienst te monitoren tot deze hun rustpunt hebben bereikt. Vervolgens doet NeCoMan hetzelfde voor de server-processen van deze dienst. Merk op dat de gedistribueerde uitvoering van deze acties gesynchroniseerd moet worden. Van zodra de client-processen op knoop x hun rustpunt bereikt hebben, zal NeCoMan vanaf deze knoop een synchronisatieboodschap versturen naar elke knoop $y \neq x$ waarop de server-processen actief zijn waarmee de client-processen op knoop x samenwerken.

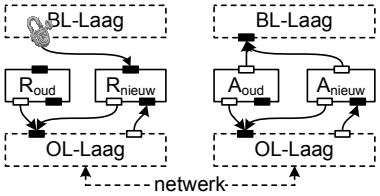
5.2.3 Activering van nieuwe dienst

Van zodra de componenten van de oude dienst hun rustpunt hebben bereikt, activeert NeCoMan de componenten van de nieuwe dienst. Gelijkaardig als bij lokale herconfiguraties houdt dit in dat verbindingen worden gewijzigd, de processen van de nieuwe componenten worden opgestart en de onderschepte pakketten worden vrijgegeven. Figuren 13 en 14 illustreren dit met de activering van de nieuwe betrouwbaarheidsdienst.

Om de nieuwe dienst correct te activeren, moet NeCoMan de gedistribueerde uitvoering van deze acties synchroniseren. Een component C kan immers slechts in gebruik worden genomen indien alle andere componenten waarmee C zal samen-



Figuur 13: Verbinden van de in-poorten van R_{nieuw} en A_{nieuw}



Figuur 14: Vrijgeven van onderschepte pakketten

werken reeds operationeel zijn. NeCoMan zal knoop x daarom pas opdragen om onderschepte pakketten vrij te geven van zodra alle andere knopen $y \neq x$ via synchronisatieboodschappen te kennen hebben gegeven dat alle verbindingen verlegd zijn (van de oude naar de nieuwe componenten).

5.2.4 Verwijdering van oude dienst

Tenslotte worden de oude componenten verwijderd. Gelijkaardig als bij lokale herconfiguraties draagt NeCoMan elke knoop op om overblijvende verbindingen met oude componenten te verbreken alvorens deze componenten te verwijderen. Merk op dat de gedistribueerde uitvoering van deze acties niet gesynchroniseerd moet worden aangezien de betrokken componenten reeds beëindigd zijn.

5.3 Gedistribueerde herconfiguratie met toestandsoverdracht

NeCoMan gebruikt een verschillend basisalgoritme wanneer de betrokken knopen niet wachten tot de oude componenten een rustpunt hebben bereikt maar in plaats daarvan deze componenten onmiddellijk stopzetten en hun huidige uitvoeringstoestand overdragen naar de nieuwe componenten. Alvorens dit algoritme te bespreken schetsen we kort de impact van deze aanpak op het herconfiguratieproces.

Zo kan deze aanpak enkel toegepast worden bij het vervangen van diensten waarbij de oude en nieuwe componenten onderling verenigbaar zijn. Dit is uiteraard nooit het geval bij het toevoegen en verwijderen van diensten. Verder is er geen gedistribueerde synchronisatie vereist om de oude dienst te beëindigen. Het stopzetten van de betrokken componenten en het overzetten van hun huidige uitvoeringstoestand kan immers onafhankelijk gebeuren voor elke betrokken knoop (dit in tegenstelling tot wanneer een rustpunt moet bereikt worden voor alle componenten van de oude dienst).

Tenslotte moet ook de gedistribueerde uitvoering van de activeringsfaze niet gesynchroniseerd worden. Door de oude componenten onmiddellijk stop te zetten zonder het bereiken van een rustpunt af te wachten, is het mogelijk dat er nog protocoltransacties actief zijn op het moment dat de nieuwe componenten in gebruik worden genomen. De nieuwe componenten moeten daarom in staat zijn

deze protocoltransacties verder uit te voeren. Bovendien vereist NeCoMan bij deze herconfiguraties ook dat oude componenten pakketten kunnen verwerken die door componenten van de nieuwe dienst zijn verstuurd. Zodoende kunnen de componenten van de oude en nieuwe dienst elkaars protocoltransacties afwerken, waardoor de gedistribueerde activering van de nieuwe dienst niet moet gesynchroniseerd worden.

Dit alles heeft tot gevolg dat NeCoMan de betrokken knopen volledig onafhankelijk van elkaar kan herconfigureren. Het basisalgoritme voor deze herconfiguratie omvat bijgevolg de onafhankelijke uitvoering van NeCoMan's lokale basisalgoritme voor het vervangen van componenten die behoren tot een gedistribueerde dienst (zie Sectie 3.2).

6 Aanpassingen bij gedistribueerde herconfiguraties

Analoog als bij lokale herconfiguraties voorziet NeCoMan een aantal voorgedefinieerde aanpassingen die het kan toepassen op de gedistribueerde basisalgoritmes. In het verloop van deze sectie schetsen we kort de belangrijkste van deze aanpassingen.

6.1 Geen gesynchroniseerde activering

Een eerste aanpassing betreft de gedistribueerde synchronisatie die nodig is om de componenten van een (nieuwe) netwerkdienst op een correcte manier te activeren. Voor sommige herconfiguraties is deze synchronisatie immers niet vereist. Dit is onder meer het geval bij de vervanging van een netwerkdienst waarbij oude en nieuwe componenten onderling verenigbaar zijn. Vermits deze componenten elkaars protocoltransacties kunnen afwerken, moet de gedistribueerde activering van de nieuwe dienst niet gesynchroniseerd worden.

Deze synchronisatie is evenmin vereist wanneer het netwerk of de applicaties kunnen omgaan met diensten die niet op een correcte manier geactiveerd zijn. Als voorbeeld hiervan verwijzen we naar de toevoeging van een compressiedienst in een programmeerbaar netwerk. Wanneer de gedistribueerde activering van deze dienst niet gesynchroniseerd wordt, bestaat het risico dat pakketten gecomprimeerd hun bestemming bereiken. Dit kan echter voorkomen worden indien het netwerk of de betrokken applicaties deze pakketten filteren of zelf decoderen.

Merk op dat het ontbreken van gedistribueerde synchronisatie bij het activeren van de nieuwe dienst ook invloed heeft op het beëindigen van de oude dienst. Wanneer oude en nieuwe componenten elkaars protocoltransacties kunnen verwerken is het immers niet meer noodzakelijk om te wachten tot de oude componenten hun rustpunt hebben bereikt. In plaats daarvan kunnen deze componenten onafhankelijk van elkaar beëindigd worden, waardoor ook hierbij geen gedistribueerde synchronisatie meer nodig is. NeCoMan gebruikt bijgevolg het algoritme dat ontworpen werd om gedistribueerde herconfiguraties met toestandsoverdracht uit te

voeren wanneer de activering van de nieuwe dienst niet gesynchroniseerd dient te worden.

6.2 Activering nieuwe dienst alvorens de oude te beëindigen

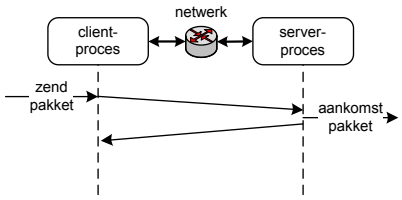
Een volgende aanpassing is (opnieuw) gericht op het omkeren van de activerings- en beëindigingsfaze. Deze aanpassing kan enkel veilig worden toegepast op het algoritme dat NeCoMan gebruikt voor gedistribueerde herconfiguraties waarbij een rustpunt moet worden bereikt indien aan volgende voorwaarden is voldaan. Zo kan de nieuwe dienst enkel geactiveerd worden alvorens de oude is beëindigd wanneer beide diensten hun uitvoeringstoestand niet delen met gebruikers. Indien dit toch het geval is bestaat immers het risico dat deze herconfiguratie inconsistente uitvoeringstoestanden voortbrengt.

Verder laat NeCoMan niet toe om de nieuwe dienst te activeren alvorens de oude is beëindigd wanneer bij het beëindigen de uitvoeringstoestand van de oude naar de nieuwe componenten moet worden overgedragen. Het is immers niet zinvol om, eenmaal de nieuwe componenten reeds geactiveerd zijn, hun uitvoeringstoestand nog te overschrijven met hoogstwaarschijnlijk verouderde toestandsinformatie. Ten slotte kan deze aanpassing enkel toegepast worden wanneer het netwerk of de applicaties die hiervan gebruik maken niet vereisen dat alle pakketten aankomen in de volgorde waarin ze verstuurd zijn. De reden hiervoor is dezelfde als bij lokale herconfiguraties.

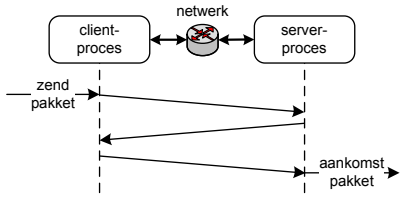
De impact van deze aanpassing is groter bij gedistribueerde dan bij lokale herconfiguraties. Zo is er bijvoorbeeld ondersteuning nodig om de pakketten die door de oude en nieuwe dienst zijn verstuurd van elkaar te onderscheiden, vermits de oude en de nieuwe componenten tijdens de herconfiguratie (tijdelijk) in parallel zullen samenwerken. NeCoMan zal daarom gebruik maken van speciale componenten die pakketten markeren en classificeren op basis van deze markering. Deze componenten worden dynamisch aan de betrokken compositie toegevoegd tijdens de installatiefaze en worden nadien verwijderd samen met de oude dienst. Verder heeft het omkeren van de activerings- en beëindigingsfaze tot gevolg dat er tijdens de herconfiguratie meerdere synchronisatieboodschappen worden uitgewisseld (3 in plaats van 2) en dat de activering van de nieuwe dienst beperkt wordt aangepast.

6.3 Geen beëindiging van oude dienst

Een derde aanpassing houdt in dat de oude dienst niet meer wordt beëindigd alvorens de bijhorende componenten te verwijderen. De voorwaarden waaraan voldaan moet zijn om deze aanpassing veilig toe te passen bij gedistribueerde herconfiguraties zijn gelijk aan deze voor lokale herconfiguraties. Merk op dat NeCoMan slechts één synchronisatieboodschap uitwisselt wanneer het deze aanpassing toepast (in plaats van twee indien NeCoMan het algoritme gebruikt waarbij de oude dienst een rustpunt moet bereiken). Deze overblijvende synchronisatieboodschap moet zorgen



Figuur 15: Lokale beëindiging van protocol



Figuur 16: Niet-lokale beëindiging van protocol

dat de nieuwe componenten op een correcte manier worden geactiveerd.

6.4 Geen beëindiging van oude server-processen

Indien een rustpunt moet worden bereikt bij een gedistribueerde herconfiguratie draagt NeCoMan's bijhorende basisalgoritme de betrokken knopen op om zowel hun client- als server-processen te monitoren tot alle protocoltransacties zijn afgewerkt. Om de oude betrouwbaarheidsdienst te beëindigen volstaat het echter om alleen de herzendingscomponent (die het client-proces bevat) te controleren. Van zodra deze component aangeeft dat alle verzonden pakketten bevestigd zijn, hebben beide betrouwbaarheidscomponenten immers een rustpunt bereikt. Bijgevolg is het overbodig om ook A_{oud} nog extra te controleren.

NeCoMan zal daarom voor sommige herconfiguraties de betrokken knopen opdragen om alleen hun client-processen te monitoren tot deze een rustpunt hebben bereikt. Deze aanpassing kan echter alleen worden toegepast indien de oude componenten communiceren via een protocol dat eindigt bij het client-proces (zoals geïllustreerd wordt in Figuur 15). Vermits dit client-proces de uitvoering van het protocol ook initieert, kan in dit geval nagegaan worden of alle protocoltransacties zijn beëindigd door alleen de client-processen te controleren. Wanneer het protocol eindigt bij een server-proces daarentegen (zoals geïllustreerd wordt in Figuur 16), moet NeCoMan de server-processen eveneens controleren om zeker te zijn dat alle protocoltransacties zijn beëindigd.

6.5 Toevoeging en verwijdering van gedistribueerde diensten

Tot slot beschrijven we kort de aanpassingen die NeCoMan doorvoert wanneer een gedistribueerde dienst toegevoegd of verwijderd moet worden (in plaats van deze te vervangen). We beperken ons hierbij tot het algoritme waarbij de oude dienst een rustpunt bereikt.

Zo worden er bij het toevoegen van een gedistribueerde dienst geen componenten verwijderd. Verder wordt het beëindigen van de "oude dienst" gerealiseerd door pakketten tegen te houden op de knopen waarop client-processen zullen worden

geïnstalleerd. Op de knopen waarop server-processen worden toegevoegd zal er vervolgens gecontroleerd worden of alle verwachtte pakketten zijn aangekomen[7].

Bij het verwijderen van een gedistribueerde dienst daarentegen, moeten er geen nieuwe componenten geïnstalleerd en geactiveerd worden. NeCoMan zal deze acties bijgevolg overslaan.

7 Ontwerp

Om hergebruik mogelijk te maken werd het *samenstellen* van een (op maat gemaakt) herconfiguratie-algoritme en de *uitvoering* van dit algoritme van elkaar losgekoppeld. Op die manier kan de logica om herconfiguratie-algoritmes te construeren herbruikt worden voor verschillende knooparchitecturen. Dit is een belangrijk voordeel gezien de complexiteit van deze logica.

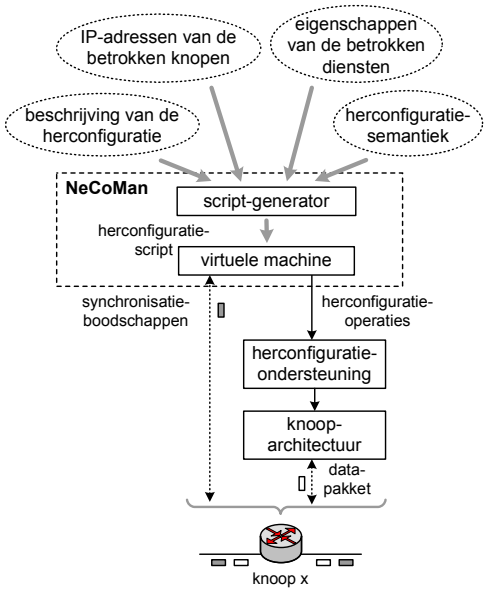
Zoals geïllustreerd in Figuur 17 werd deze loskoppeling gerealiseerd door de NeCoMan functionaliteit op te splitsen in een script-generator (die de logica bevat om een herconfiguratie-algoritme op maat te maken) en één of meerdere virtuele machines (die deze algoritmes zullen uitvoeren). De script-generator creëert voor elke knoop die deelneemt aan de herconfiguratie een script op basis van (1) een beschrijving van de herconfiguratie die moet uitgevoerd worden, (2) de eigenschappen van de betrokken diensten, (3) de herconfiguratiesemantiek en (4) de IP-adressen van de betrokken knopen. Dit script bevat een (knoop-onafhankelijke) beschrijving van de verschillende acties die de virtuele machine van de betrokken knoop moet uitvoeren. Bij de uitvoering van dit script wordt de herconfiguratie-ondersteuning van de betrokken knoop aangesproken om de gespecificeerde herconfiguratie-operaties uit te voeren. De uitvoering van de synchronisatie-operaties daarentegen, neemt de virtuele machine zelf voor zijn rekening.

Deze opsplitsing heeft als bijkomend voordeel dat de herconfiguratielogica niet noodzakelijk op de programmeerbare netwerkknopen zelf moet worden uitgevoerd. De herconfiguratiescripts kunnen immers perfect op speciaal daartoe voorziene knopen of zelfs buiten het netwerk samengesteld worden. Bij de uitvoering worden deze scripts dan verdeeld over de betrokken knopen. Op die manier gebruikt de herconfiguratielogica geen van de (dikwijls schaarse) hulpbronnen (resources) die de betrokken knopen gebruiken om pakketten te verwerken.

8 Besluit

Als besluit sommen we de bijdragen op van dit onderzoek. We doen dit voor elk van de domeinen beschreven in Sectie 2.

[7] Herinner dat NeCoMan zelf geen functionaliteit bevat om dit te realiseren. Dit is de verantwoordelijkheid voor de herconfiguratie-ondersteuning van de betrokken knopen.



Figuur 17: Hoog-niveau overzicht van de NeCoMan architectuur

8.1 Programmeerbare netwerken

In de context van programmeerbare netwerken stellen we de NeCoMan middleware voor om dynamische herconfiguraties van extern programmeerbare netwerkknopen uit te voeren. Deze middleware coördineert het dynamisch toevoegen, verwijderen en vervangen van lokale en gedistribueerde netwerkdiensten. Verder voldoet deze middleware aan de vooropgestelde vereisten:

- **Correcte herconfiguraties.** De gebruikte basisalgoritmes voldoen aan alle opgelegde *herconfiguratievoorwaarden*. Deze herconfiguratievoorwaarden bepalen de volgorde waarin de verschillende stappen van een herconfiguratie dienen uitgevoerd te worden. Ook het toepassen van de voorgedefinieerde aanpassingen brengt de correctheid van een herconfiguratie niet in gevaar, zolang tenminste aan alle voorwaarden is voldaan om de betrokken aanpassing veilig uit te kunnen voeren.

- **Beperkte herconfiguratiekost.** Voor sommige herconfiguraties kan de kost van de basisalgoritmes verminderd worden door het gebruikte algoritme op maat te maken. NeCoMan voorziet daarom een aantal voorgedefinieerde aanpassingen die toegepast kunnen worden op de basisalgoritmes. Op die manier tracht NeCoMan de impact van een herconfiguratie op de beschikbaarheid en performantie van het netwerk te beperken.

- **Beperkte openheid.** NeCoMan vereist enkel een beschrijving van (1) de herconfiguratie die moet uitgevoerd worden en (2) de eigenschappen van de betrokken diensten en de herconfiguratiesemantiek alvorens een herconfiguratie uit te kunnen voeren. Op basis van deze gegevens selecteert NeCoMan het gepaste basisalgoritme en past hier (indien opportuun) één of meerdere voorgedefinieerde aanpassingen op toe. Zodoende schermt NeCoMan de complexiteit die eigen is aan (correcte en efficiënte) herconfiguraties van netwerk-software af van de gebruikers.
- **Herbruikbaarheid.** Om NeCoMan te kunnen gebruiken boven verschillende knooparchitecturen werd de samenstelling van een (op maat gemaakt) herconfiguratie-algoritme losgekoppeld van de uitvoering hiervan. Dit laat toe NeCoMan's script-generator te herbruiken voor verschillende (knoopspecifieke) virtuele machines. Dit is een belangrijk voordeel gezien de complexiteit van de logica om herconfiguraties op maat te maken.

De NeCoMan middleware is gevalideerd aan de hand van verschillende herconfiguraties. Hierbij werden een compressiedienst, een betrouwbaarheidsdienst en (een DiPS+ implementatie van) de TCP-booster ontwikkeld aan de Universiteit van Gent [9] in een DiPS+ netwerk toegevoegd, verwijderden vervangen.

8.2 Dynamische software-herconfiguratie

Dit proefschrift beschrijft hoe NeCoMan de lokale en gedistribueerde uitvoering van compositionele aanpassingen coördineert. Deze coördinatie volgt uit een aantal herconfiguratievoorwaarden waaraan voldaan moet worden om correcte en efficiënte herconfiguraties uit te voeren. Door deze voorwaarden expliciet te maken bieden ze een hulpmiddel bij de ontwikkeling van nieuwe systemen die dynamische herconfiguratie moeten ondersteunen.

8.3 Beheerssystemen voor dynamische herconfiguratie

Tot slot werd in dit proefschrift het gebruik van *aanpasbare* beheerssystemen voor dynamische herconfiguratie in programmeerbare netwerken onderzocht. In tegenstelling tot bestaande "gesloten" beheerssystemen enerzijds, laat deze aanpak toe om het gebruikte herconfiguratie-algoritme op maat te maken voor elke herconfiguratie. In tegenstelling tot "open" beheerssystemen anderzijds, blijft de gebruiker afgeschermd van de complexiteit die eigen is aan het samenstellen van een correct en efficiënt herconfiguratie-algoritme. Zodoende werd een goed evenwicht gevonden tussen (1) het aanbieden van voldoende flexibiliteit om de efficiëntie van herconfiguraties te verbeteren indien mogelijk en (2) het beperken van de kost en risico's die verbonden zijn aan dynamische software-herconfiguratie.

Referenties

[1] João Paulo A. Almeida, Maarten Wegdam, Marten van Sinderen, and Lambert J. M. Nieuwenhuis. Transparent Dynamic Reconfiguration for CORBA. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA 2001)*, pages 197–207, Rome, Italy, September 2001. IEEE Computer Society.

[2] Gregory R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):49–90, 1991.

[3] Kenneth L. Calvert, Samrat Bhattacharjee, Ellen Zegura, and James Sterbenz. Directions in Active Networks. *IEEE Communications Magazine, Special Issue on Programmable Networks*, 36(10):72–78, October 1998.

[4] Andrew T. Campbell, Herman G. De Meer, Michael E. Kounavis, Kazuho Miki, John B. Vicente, and Daniel Villela. A Survey of Programmable Networks. *ACM SIGCOMM Computer Communications Review*, 29(2):7–23, April 1999.

[5] Wen-Ke Chen, M.A. Hiltunen, and R.D. Schlichting. Constructing Adaptive Software in Distributed Systems. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS'01)*, pages 635–643. IEEE Computer Society, 2001.

[6] Geoff Coulson, Gordon Blair, David Hutchison, Ackbar Joolia, Kevin Lee, Jo Ueyama, Antônio Gomes, and Yimin Ye. Netkit: a software component-based approach to programmable networking. *ACM SIGCOMM Computer Communication Review*, 33(5):55–66, 2003.

[7] Michael Hicks and Scott Nettles. Active networking means evolution (or enhanced extensibility required). In Hiroshi Yashuda, editor, *Proceedings of the Second International Working Conference on Active Networks (IWAN 2000)*, volume 1942 of *Lecture Notes in Computer Science*, pages 16–32. Springer-Verlag, October 2000.

[8] Jamie Hillman and Ian Warren. An open framework for dynamic reconfiguration. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 594–603, Washington, DC, USA, 2004. IEEE Computer Society.

[9] Jeroen Hoebeke, Tom Van Leeuwen, Liesbeth Peters, Koen Cooreman, Ingrid Moerman, Bart Dhoedt, and Piet Demeester. Development of a TCP protocol booster over a wireless link. In *Proceedings of the 9th Symposium on Communications and Vehicular Technology in the Benelux (SCVT 2002)*, Louvain la Neuve, oct 2002.

[10] Nico Janssens, Lieven Desmet, Sam Michiels, and Pierre Verbaeten. NeCoMan: Middleware for Safe Distributed Service Deployment in Programmable Networks. In *Proceedings of the 3rd workshop on Adaptive and Reflective Middleware*, pages 256–261, Toronto, Ontario, Canada, 2004. ACM Press.

[11] Nico Janssens, Wouter Joosen, and Pierre Verbaeten. NeCoMan: Middleware for Safe Distributed-Service Adaptation in Programmable Networks. *IEEE Distributed Systems Online*, 6(7), July 2005.

[12] Nico Janssens, Elke Steegmans, Tom Holvoet, and Pierre Verbaeten. An Agent Design Method Promoting Separation Between Computation and Coordination. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC'04)*, pages 456–461, Nicosia, Cyprus, 2004. ACM Press. Special Track on Coordination Models, Languages and Applications.

[13] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[14] Jeff Kramer and Jeff Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, 1990.

[15] Yueh-Feng Lee and Ruei-Chuan Chang. Developing dynamic-reconfigurable communication protocol stacks using Java. *Softw. Pract. Exper.*, 35(6):601–620, 2005.

[16] Ulana Legedza, David Wetherall, and John V. Guttag. Improving the Performance of Distributed Applications Using Active Networks. In *Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 1998)*, volume 2, pages 590–599, San Francisco, CA, USA, April 1998.

[17] Frank Matthijs. *Component Framework Technology for Protocol Stacks*. PhD thesis, K.U. Leuven, December 1999.

[18] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing Adaptive Software. *IEEE Computer*, 37(7):56–64, July 2004.

[19] Sam Michiels. *Component Framework Technology for Adaptable and Manageable Protocol Stacks*. PhD thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, November 2003.

[20] Sam Michiels, Nico Janssens, Lieven Desmet, Tom Mahieu, Wouter Joosen, and Pierre Verbaeten. Connecting Embedded Devices Using a Component

Platform for Adaptable Protocol Stacks. In *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends*, volume 3778/2005 of *Lecture Notes in Computer Science*, pages 185–208. Springer-Verlag, 2005.

[21] Konstantinos Psounis. Active Networks: Applications, Security, Safety, and Architectures. *IEEE Communications Surveys*, pages 1–16, First Quarter 1999.

[22] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[23] Robbert van Renesse, Ken Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building Adaptive Systems Using Ensemble. *Software – Practice & Experience*, 28(9):963–979, 1998.

[24] Ian Warren and Ian Sommerville. Dynamic configuration abstraction. In W. Schäfer and P. Botella, editors, *Proceedings of the Fifth European Software Engineering Conference*, pages 173–190. Springer-Verlag, 1995.