

**KATHOLIEKE UNIVERSITEIT LEUVEN**
FACULTEIT TOEGEPASTE WETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — B-3001 Leuven

# Component Framework Technology for Adaptable and Manageable Protocol Stacks

Promotoren :
Prof. Dr. ir. P. VERBAETEN
Prof. Dr. ir. W. JOOSEN

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de toegepaste wetenschappen

door

**Sam MICHIELS**

November 2003



**KATHOLIEKE UNIVERSITEIT LEUVEN**
FACULTEIT TOEGEPASTE WETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — B-3001 Leuven

# Component Framework Technology for Adaptable and Manageable Protocol Stacks

Jury :
Prof. Dr. ir.-arch, J. Delrue, voorzitter
Prof. Dr. ir. P. Verbaeten, promotor
Prof. Dr. ir. W. Joosen, promotor
Prof. Dr. ir. Y. Berbers
Prof. Dr. ir. E. Steegmans
Prof. Dr. ir. B. Dhoedt
Prof. Dr. ir. E. Van Lil

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de toegepaste wetenschappen

door

**Sam MICHIELS**

U.D.C. 681.3∗C22

November 2003

©Katholieke Universiteit Leuven – Faculteit Toegepaste Wetenschappen
Arenbergkasteel, B-3001 Heverlee (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotocopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm or any other means without written permission from the publisher.

D/2003/7515/58
ISBN 90–5682–446–5

# Abstract

The context of this dissertation and the corresponding challenges are formed by network services and their requirements to the underlying protocol stack. The relevance of this research is confirmed by three recent trends in network services and their execution environment: growing reliance of businesses and individuals on network connectivity, highly dynamic network characteristics, and a wide range of connected client device types. These three trends reveal a common challenge for protocol stack software: offering support for flexibility at design-, development-, and execution-time, tailored to application- and/or network-specific characteristics.

This dissertation presents DiPS+ (Distrinet Protocol Stack+), a sophisticated software architecture and component framework to support the development of protocol stacks that are easily customizable to application- and environment-specific requirements. DiPS+ has been used to address three core research tracks in protocol stack development: self-adaptability and management, testing, and framework optimization.

The strength of this dissertation is that these three tracks are not addressed by individual solutions, yet share a well-defined software architecture and component framework underneath. The DiPS+ component framework does not only offer programmers support in developing protocol stacks, but also lays the foundation for a robust and lightweight development process and a balanced software lifecycle.

*"No wasted time, we're alive today*
*Churnin up the past, there's no easier way*
*Time's been between us, a means to an end*
*God it's good to be here walkin together my friend."*
*– From D. Bramhall and B. Logan – Life by the drop,*
*performed by S. R. Vaughan).*

# Voorwoord – Preface

Deze tekst beschrijft het werk dat ik verricht heb in de DistriNet onderzoeksgroep van het departement Computerwetenschappen aan de K.U.Leuven. De kern van dit werk is het DiPS+ (Distrinet Protocol Stack+) componentenraamwerk dat verder bouwt op het originele DiPS raamwerk. DiPS+ ondersteunt het bouwen van protocolstapels en heeft specifieke aandacht voor het beheren, testen, en optimaliseren ervan.

Het tot een goed einde brengen van dit onderzoek was geen eenvoudige opgave. Het is pas na veel vallen en opstaan dat ik geleidelijk aan de kneepjes van het vak geleerd heb. Bij het begin van mijn proefschrift zou ik dan ook een aantal mensen willen bedanken die me (bewust of onbewust) geholpen en gesteund hebben bij mijn evolutie van schuchtere thesisstudent tot zelfzekere doctorandus.

In de eerste plaats bedank ik mijn promotoren, Prof. Pierre Verbaeten en Prof. Wouter Joosen, voor het vertrouwen dat ze in mij stelden tijdens mijn onderzoek, ook in moeilijke dagen. Hun werkkracht, ambitie, openheid en stiptheid zijn voor mij altijd een voorbeeld geweest. Ik hoop dat ik nog veel van hen mag leren. Daarnaast dank ik ook van harte de andere leden van de begeleidingscommissie, Prof. Yolande Berbers en Prof. Eric Steegmans, voor het grondig nalezen van deze tekst. Ook de voorzitter en de leden van de doctoraatsjury, Prof. Jan Delrue, Prof. Emmanuel Van Lil en Prof. Bart Dhoedt wil ik langs deze weg bedanken.

Ik zou ook iedereen van de DistriNet onderzoeksgroep – en in het bijzonder de netwerkgroep – willen bedanken voor de aangename omgeving waarin ik heb mogen werken. Lieven Desmet, Nico Janssens, Tom Mahieu, Frank Matthijs en Dirk Walravens stonden altijd klaar voor een interessante en leerrijke discussie. Tijdens de voorbije maanden toonden ze ook veel begrip toen ik bezig was aan deze tekst, en spaarden ze hun vragen en discussies op voor 's donderdags zodat ik de rest van de week kon doorwerken. Jean Huens en Bart Swennen verdienen

ook een speciale vermelding. Niet alleen staan ze – samen met de rest van de systeemgroep – altijd klaar om kleine en grote computerproblemen op te lossen, Jean en Bart zorgen ook voor noodzakelijke ontspanningsmomenten door even binnen te springen in ons bureau tijdens het wachten op hun koffie.

Wie me kent weet dat mijn vrienden me zeer dierbaar zijn. Tijdens de voorbije maanden heb ik hun steun goed kunnen gebruiken. Ze beseften waarschijnlijk niet altijd wat het voor mij betekende wanneer ze me vroegen of ik 'still alive' was omdat ik een tijdje niets van me had laten horen, of wanneer ze voorstelden om nog eens een pint te gaan drinken als ze zagen dat ik mijn ontspanning vergat te nemen.

Ik kan hier niet alle namen opnoemen van vrienden die een speciale plaats innemen in mijn leven. Een paar 'kameraden' wil ik toch wel expliciet vermelden, jullie begrijpen wel waarom. Guy, Tom en Filip zorg(d)en voor onvergetelijke en toffe jaren in Aarschot en omstreken. Ik zal ook Arn zijn voetbalanalyses nooit vergeten, evenals Davy en Goedele hun kookkunsten, of de uitstapjes op de Oude Markt met Merlijn, Hans en Wilfried. Daarnaast heeft Johan er voor gezorgd dat mijn fysieke conditie op punt bleef door legendarische step-competities, steeds langere fietstochten en de wekelijkse kine-praatuurtjes waar menig voetballiefhebber een vette kluif aan zou hebben. Tenslotte wil ik Ann bedanken. Niet alleen voor haar onbeschrijfelijke motivatie om alle taalfouten uit mijn tekst te halen, maar evenzeer voor haar lach die de meest sombere zwartkijker doet smelten. Zij was het ook die perfect het evenwicht aanvoelde tussen ontspanning en heel hard werken. Ann, bedankt voor heel veel!

Ik wil afsluiten met de mensen die het dichtst bij mij staan: mijn ouders en mijn drie broers en schoonzussen. Met een simpele 'tzal wel lukken, ge zult da wel zien' heeft ons ma mij door het examen Analyse in de eerste kandidatuur geduwd. Wie weet waar stond ik nu zonder haar? Zij weet hoe ik in mekaar zit, al houdt ze haar hart dikwijls vast als ik weer eens een uitdaging gevonden heb die op het eerste zicht niet echt haalbaar lijkt... Gelukkig zorgt onze pa dan voor de relativerende en motiverende noot door een simpele 'als gij dat ziet zitten moet ge niet twijfelen'. Zowel ons ma als onze pa hebben zich waarschijnlijk 10.000 keer afgevraagd waar ik toch al die tijd mee bezig was... Wel, dit is het resultaat, het resultaat van gezond boerenverstand en veel doorzettingsvermogen... Ook mijn broers en schoonzussen staan altijd voor mij klaar... Mijn grote broers, die onder andere zorgden voor bluesmuziek, vorstelijk lekker eten, voetbal en een waterdichte verzekering. Bert, Lydia, Jos, Greet, Raf, Carole, Melissa, Stefanie, Valérie, Dafne, Zjef en Kobe, ik ben er zeker van dat we nog veel toffe familiefeestjes gaan meemaken. Aan die laatste twee – mijn petekindjes – draag ik dit werk op. Ik hoop dat ze ooit begrijpen wat er in staat, maar eigenlijk is dat niet zo belangrijk...

Sam Michiels, 2 Oktober 2003.

*Aan Zjef en Kobe...*

# Contents


**1 Introduction** **1**
- 1.1 Protocol stacks . . . 1
- 1.2 Context . . . 3
- 1.3 Towards flexible protocol stacks . . . 4
- 1.4 Contributions . . . 6
- 1.5 Overview . . . 7

**2 Background** **9**
- 2.1 Key problems and characteristics of protocol stack software . . . 9
  - 2.1.1 Software design and architecture . . . 10
  - 2.1.2 Extreme requirements . . . 10
- 2.2 Software engineering background . . . 11
  - 2.2.1 From programming to engineering . . . 12
  - 2.2.2 Software architectures and components . . . 17
- 2.3 Research challenges . . . 20
  - 2.3.1 Adaptability . . . 21
  - 2.3.2 Testing . . . 28
  - 2.3.3 Framework optimization . . . 32
- 2.4 Conclusion . . . 36

**3 The DiPS+ Component Framework** **37**
- 3.1 From DiPS to DiPS+ . . . 38
  - 3.1.1 Objectives . . . 38
  - 3.1.2 Shortcomings in DiPS . . . 38
- 3.2 The DiPS+ architecture . . . 39
  - 3.2.1 Data and management plane . . . 39
  - 3.2.2 Inside the data plane . . . 40
  - 3.2.3 Concurrency control . . . 45
- 3.3 Design of the data plane . . . 47
  - 3.3.1 Packet and anonymous communication . . . 47
  - 3.3.2 Component entry and exit point . . . 48

3.3.3 Unit types . . . 48
3.3.4 Layers . . . 51
3.4 Design of the management plane . . . 55
3.4.1 Intercepting the packet flow . . . 55
3.4.2 Concurrency management . . . 57
3.5 The DiPS+ prototype . . . 59
3.5.1 Building a protocol stack . . . 59
3.5.2 Architecture description . . . 61
3.5.3 Developer roles . . . 64
3.6 Conclusion . . . 66
3.6.1 Contributions of DiPS+ . . . 66
3.6.2 Summary . . . 67

**4 Self-adaptive system behavior 69**
4.1 The DMonA approach . . . 71
4.1.1 The relationship with DiPS+ . . . 71
4.1.2 The DMonA design . . . 72
4.2 A case study: load management . . . 74
4.2.1 Thread allocation models . . . 74
4.2.2 Locally and dynamically assigning threads . . . 78
4.2.3 Admission control . . . 83
4.3 The DMonA prototype . . . 84
4.3.1 Handling peak loads . . . 85
4.3.2 Handling bottlenecks . . . 86
4.4 Conclusion . . . 89

**5 DiPS+ testing 91**
5.1 Testing issues from a DiPS+ perspective . . . 92
5.1.1 Component controllability . . . 92
5.1.2 Component observability . . . 93
5.1.3 A uniform test approach . . . 94
5.1.4 Incremental testing . . . 95
5.1.5 Environment control . . . 96
5.2 The DiPSUnit test framework . . . 98
5.2.1 Uniform test infrastructure . . . 98
5.2.2 Dealing with asynchronous behavior . . . 103
5.2.3 Event control . . . 104
5.2.4 Environment control . . . 107
5.3 Examples . . . 108
5.3.1 A component test case . . . 109
5.3.2 A protocol layer test case . . . 110
5.4 Lessons learned . . . 113
5.4.1 Software fault detection . . . 113

5.4.2 Testing experiences . . . . . 114
5.5 Conclusion and future work . . . . . 114

**6 DiPS+ framework optimization 117**
6.1 DiPS+ performance analysis . . . . . 118
6.1.1 System level analysis . . . . . 118
6.1.2 Unit level analysis . . . . . 120
6.2 Limitations of optimizing compilers . . . . . 122
6.3 The DiPS+ optimization approach . . . . . 124
6.3.1 Optimization granularity . . . . . 124
6.3.2 Language support for framework optimization . . . . . 126
6.3.3 An optimizing builder . . . . . 126
6.4 Validation . . . . . 127
6.5 Conclusion . . . . . 128

**7 Related work 131**
7.1 Protocol stack frameworks . . . . . 131
7.1.1 Scout . . . . . 131
7.1.2 Click modular router . . . . . 132
7.1.3 Cactus . . . . . 132
7.1.4 Staged Event-Driven Architecture . . . . . 133
7.2 Provided support during software development . . . . . 133
7.2.1 Adaptability . . . . . 133
7.2.2 Testability . . . . . 134
7.2.3 Framework optimization . . . . . 134

**8 Conclusion 135**
8.1 Major achievements . . . . . 135
8.2 Validation of the DiPS+ framework . . . . . 136
8.2.1 Admission control in the RADIUS protocol . . . . . 136
8.2.2 On demand protocol stacks . . . . . 140
8.2.3 DiPS+ Master's theses . . . . . 140
8.3 Applicability of the DiPS+ approach . . . . . 141
8.3.1 USB device drivers . . . . . 141
8.3.2 File systems . . . . . 143
8.4 Future Work . . . . . 143
8.4.1 Component hot-swapping . . . . . 144
8.4.2 Distributed system management . . . . . 144
8.4.3 Safe adaptations . . . . . 144

# List of Figures


1.1 Sending a TCP/IP packet via an intermediate router, which strips the Ethernet and the IP header, selects the next node to deliver the packet to (i.e. the receiver), and forwards the packet via the network and datalink layer, which both attach a new header to the packet. . . . . 2

2.1 An intuitive illustration of an object-oriented framework, which is no longer present at execution-time. . . . . 15
2.2 Inversion of control: in the conventional approach (left), application code calls generic library and system code; in the framework approach (right), application code is called by the generic framework, which controls when and how application code is executed. . 16
2.3 Intuitive illustration of a component framework, which is not only present at design- and instantiation-time, but also at execution-time. 20
2.4 General scope of the dissertation: a well-designed component framework with an associated software architecture that supports three crucial challenges of software development – self-adaptability, testing and framework optimization. . . . . 21

3.1 The data and the management plane in the DiPS+ architecture. . 40
3.2 Example of a DiPS+ component pipeline with a dispatcher that splits the pipeline in two parallel component areas. More processing resources have been assigned to the upper concurrency component. 42
3.3 Anonymous communication via a blackboard architectural style: a blackboard data structure has been coupled to each message to carry meta-information from one component to another. . . . . 45
3.4 The design of two core DiPS+ abstractions: component and packet. 47
3.5 A DiPS+ component with a packet receiver, a unit, and a packet forwarder. . . . . 48
3.6 The design of the unit class hierarchy in DiPS+. . . . . 49
3.7 The design of the DiPS+ concurrency unit. . . . . 50

3.8 The design of the DiPS+ dispatching unit. . . . . 51

3.9 The design of a DiPS+ layer with identification of upper/lower entry and exit points, and layer dispatchers. . . . . 53

3.10 A DiPS+ component with a policy object at its packet receiver. The policy delegates incoming packets to a pipeline of management modules. . . . . 57

3.11 Two DiPS+ concurrency units with three interfaces: the functional interface is used to deliver packets to a unit; the management interface to set buffer overflow and underflow strategies; and the scheduler interface to associate a packet handler with a concurrency unit. 59

3.12 The design of the DiPS+ builder with four types of stack properties, a generic parser, a specific builder for each type of property, and the protocol stack that collects all property instances. . . . . 60

3.13 Architectural and functional view on a DiPS+ system. . . . . 64

4.1 DMonA (DiPS+ Monitoring Architecture) as the management plane on top of the DiPS+ data plane. . . . . 72

4.2 The design of DMonA. . . . . 73

4.3 Illustration of three thread allocation models: the thread-per-request, the bounded thread pool, and the DiPS+ model. The component pipeline consists of three components, with increasing load. This creates a bottleneck at the end of the pipeline for the DiPS+ model, which inserts a concurrency unit in front of each component (resulting in three component areas). The thread-per-request model is performing well until 300 concurrent requests. As load increases, throughput degrades exponentionally. The throughput of the other two models remains constant for high loads. The overhead of concurrency components inside the component pipeline, and the imperfect balance of processing resources explain the throughput difference between the bounded thread pool model and DiPS+. . . . 75

4.4 Illustration of DMonA attached to DiPS+ via two policies, one associated with the packet receiver and one with the packet forwarder. Processing resources are retrieved from a pool of free resources and allocated to a concurrency unit via its scheduler interface. . . . . 79

4.5 Intuitive illustration of balancing processing resources over multiple component areas to resolve bottlenecks. Area A requires 20% of the processing-time, area B 30%, and area C 50%. In the top figure, processing resources are equally distributed over the three component areas (i.e. 33% of the available resources for each area). Since area C is more processing-intensive than area A and B, it becomes the bottleneck (i.e. the packet queue before the area fills). The middle figure shows how DMonA migrates one packet handler to area C. Since this creates a new bottleneck in area B, DMonA moves a packet handler from area A to area B. This results in the optimal balance of processing resources according to the load of each area. 82
4.6 Using DMonA to control the admission of new packets in a component area. The management module associated with the packet receiver acts as a leaky bucket and drops packets when the buffer is full. The management module at the packet forwarder measures response-time, which is used to steer the packet delivery rate of the leaky bucket. . . . . 84
4.7 Comparison of four thread assignment methods handling increasing processing load. The component pipeline consists of three components, with increasing load. This creates a bottleneck at the end of the pipeline for the DiPS+ and the DMonA model, which insert a concurrency unit in front of each component (resulting in three component areas). The thread-per-request model is performing better until 300 concurrent requests. As load increases, throughput quickly drops below that of the bounded thread pool, the DiPS+ and the DMonA model. By balancing the processing resources more optimally, DMonA clearly outperforms the DiPS+ model. . . . . 86
4.8 Measured throughput in three areas with changing load. The DiPS+ model suffers from unbalanced thread distribution. The DMonA model clearly outperforms the other two, although throughput varies due to a non-stable thread distribution. . . . . 88
4.9 Dynamic thread re-allocation over three component areas. . . . . 90

5.1 The design of the DiPSUnit test framework. . . . . 97
5.2 The DiPSUnit test framework connected to a component with one packet receiver and one packet forwarder. . . . . 99
5.3 Some intuitive examples of layer test case scenarios. On the left, a normal packet is sent through the layer. In the middle, a packet is received from the network, forwarded through the bottom exit point, and (virtually) delivered to the local stack through the upper exit point. The right most scenario is the same, but limited to packet forwarding. The upper entry and exit point are not used in this case. . . . . 100

5.4 The DiPSUnit test framework connected to a DiPS+ layer (two packet receivers, two packet forwarders). . . . . 101
5.5 The DiPSUnit test framework with monitor support connected to a DiPS+ layer with a concurrency component. . . . . 104
5.6 The DiPSUnit test framework connected to a DiPS+ layer with an event throwing component. . . . . 105
5.7 Packet filtering inside the protocol stack. The two filters in the down going flow delete and delay packets at respectively the transport and the network layer. The filter in the up going flow duplicates incoming packets at the datalink level. . . . . 107
5.8 The DiPSUnit framework coupled with JUnit. The detailed DiPSUnit design can be found in Figure 5.1 on page 97. . . . . 108
5.9 Graphical representation of the Ethernet layer test case. . . . . 111

6.1 UDP/IP/loopback protocol stack performance tests for a standard DiPS+ version and for one with units merged. 100.000 packets sent, ran on an Intel Pentium 4 CPU 1.70GHz with 512 MB RAM running Linux 2.4.20-pre11. All software written in Java, compiled and executed with JVMs from SUN (1.4.0) [Sunc], IBM (1.4) [IBM03] and Blackdown (1.4.1) [Bla03]. Tests on SUN and Blackdown executed with the -server option for full hotspot optimization [Sun03b]. 119
6.2 Connector overhead relevance, given "load simulator" components with increasing load. The measured load of existing unit implementations varies between 4 and 13. . . . . 121
6.3 Two relevant AOP shortcomings: aspect relation between two or more objects (left), and two or more aspect instances related to the same object (right). . . . . 124
6.4 Intuitive illustration of the standard DiPS+ IPv4 layer (left), and the fully optimized version (right). . . . . 125
6.5 Intuitive illustration of the standard DiPS+ IPv4 layer (left), and the partially optimized version (right). . . . . 126
6.6 Scenario for building a standard and an optimized DiPS+ protocol stack. . . . . 128
6.7 UDP/IP/loopback protocol stack performance tests for a standard DiPS+ version, an optimized DiPS+ version, a simulated AspectJ version and one with units merged. 100.000 packets sent, ran on an Intel Pentium 4 CPU 1.70GHz with 512 MB RAM running Linux 2.4.20-pre11. All software written in Java, compiled and executed with JVMs from SUN (1.4.0) [Sunc], IBM (1.4) [IBM03] and Blackdown (1.4.1) [Bla03]. Tests on SUN and Blackdown executed with the -server option for full hotspot optimization [Sun03b]. . . . . 129

8.1 Design of RADIUS authentication and accounting in DiPS+. The top figure illustrates a scheduler strategy that distinguishes gold, silver, and bronze users. The bottom figure, demonstrates a strategy that prioritizes accounting requests over authentication requests. . 138
8.2 Two high-level views on the USB stack: traditional layered architecture (left) and DiPS+ manageable architecture (right). . . . . 142

# List of Tables

4.1 Dynamically changing the load in three component areas. The interval between two changes of load is 400 seconds. . . . . 87

6.1 Processing-time of the pipe connector and the dispatching component for sending 10.000 packets. . . . . 120
6.2 Processing-time and load indicator of standard DiPS+ components. Time measurements are for processing 10.000 packets. The load indicator positions each component on a virtual load scale, which is also used in Table 6.1 . . . . . 122

# List of Listings


3.1 The interfaces for a policy and its associated management modules. 56
3.2 The management interface of a concurrency unit. . . . . 58
3.3 Building a DiPS+ protocol stack. . . . . 61
3.4 An example of a structure property description of a DiPS+ protocol stack with four layers (Loopback, Ethernet, IP, and TCP), the glue that connects these layers (IPonLoopback, IPonEthernet, and TCPonIP), two datalink interfaces (lo and veth0), and a TCP socket. . . . . 63
3.5 An example of the layer property description for the DiPS+ IPv4 layer. . . . . 65
5.1 The setup and tear down code of a DiPS+ Ethernet header constructor test case in DiPSUnit. . . . . 109
5.2 The code of a test case for the DiPS+ Ethernet header constructor in DiPSUnit. . . . . 110
5.3 The setup and tear down code of a DiPS+ Ethernet layer test case in DiPSUnit. . . . . 112
5.4 The code of a test case for the DiPS+ Ethernet layer in DiPSUnit. 113
6.1 The optimization description for a DiPS+ UDP/IP protocol stack. 127

*"The price of success is hard work, dedication to the job at hand, and the determination that whether we win or lose, we have applied the best of ourselves to the task at hand."*

*– From Vince Lombardi.*

# Chapter 1

# Introduction

The objectives of this chapter are to present the context of the dissertation and to outline the main challenges that will be tackled. Since protocol stacks play a central role throughout the text, their main characteristics are sketched in Section 1.1. The chapter then presents three trends in network services and their execution environments, and identifies support for flexibility as the major challenge for protocol stack software. Section 1.3 proposes our view on the development of flexible protocol stack software, which is based on three major software engineering tracks: self-adaptation, testing and optimization. The chapter ends with a summary of the main contributions in Section 1.4, and an overview of the chapters to come in Section 1.5.

## 1.1 Protocol stacks

The main goal of a protocol stack is to enable communication between devices (also referred to as nodes) that are attached to a network [Tan96]. Doing so, a protocol stack hides all network transfer details from applications running on top of it. The protocol stack may, for instance, be responsible for delivering data correctly at the receiver, even if data can get lost during transfer. In addition, the protocol stack is responsible for finding a route from the sender of the data to the receiver.

The major tasks of a protocol stack are conceptually separated in protocol layers stacked on top of each other (which explains its name). The most common

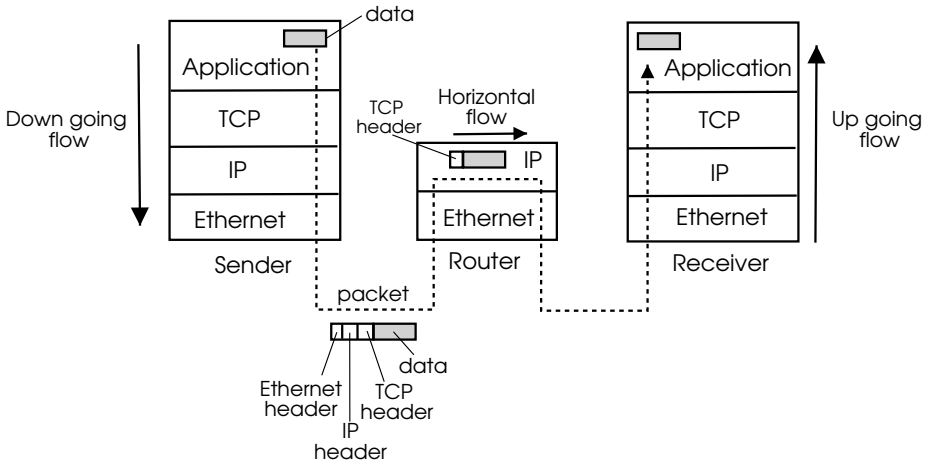



Figure 1.1: Sending a TCP/IP packet via an intermediate router, which strips the Ethernet and the IP header, selects the next node to deliver the packet to (i.c. the receiver), and forwards the packet via the network and datalink layer, which both attach a new header to the packet.

model for protocol stacks consists of four layers[1] : (1) the application layer, which encapsulates end user services (e.g. e-mail, file transfer, remote login, etc.); (2) the transport layer, which is responsible for end-to-end delivery of data between a sender and a receiver. Sender and receiver nodes may not be adjacent (i.e. directly connected); (3) the network layer of each node, therefore, selects the next node towards the receiver that is directly reachable from the sender; (4) the datalink layer, finally, is responsible for physically transferring a data packet between adjacent nodes. This four-layered model is used to connect devices on the Internet. It is referred to as the TCP/IP model, as TCP (Transmission Control Protocol [ISI81b]) and IP (Internet Protocol [ISI81a]) are respectively the most used transport and network protocol on the Internet. Yet, many other protocols exist at each layer of the protocol stack. Figure 1.1 illustrates the TCP/IP protocol stack model.

Each layer in a protocol stack relies on the services offered by the layer underneath. The transport layer, for instance, relies on the network layer of each intermediate node to find the next step in the route to the receiver. As such, the chain of network nodes selected in the network layer of each intermediate node

[1]The OSI (Open System Interconnection) protocol stack model [Tan96] consists of seven layers, and is often used to explain the ideas behind protocol stacks. As such, it serves as a reference model but, in contrast to the TCP/IP model, it lacks wide-spread implementations. Most of the functionality in the OSI model exists in all protocol stacks, although two or three OSI layers may be incorporated into one.

forms the end-to-end route used by the transport layer. The network layer, for its part, relies on the datalink layer to transfer the data to the network driver, which puts the data on the physical network.

Multiple flows can be detected inside a protocol stack. First, a down going flow represents a data packet that is sent to the network. The packet starts at the top of the stack and traverses each protocol layer until it reaches the bottom. At this point, the data is transferred to the network driver. Secondly, when a packet arrives from the network, it is processed in an up going flow from the bottom of the protocol stack to the top. Finally, a horizontal flow may connect the up and the down going flow. For example, at the network layer of a router, incoming packets may be forwarded from the up going to the down going flow. In this case, the packet is not delivered locally but forwarded to the next (intermediate) node on the end-to-end route. Figure 1.1 illustrates the three flows (down, up, and horizontal) that transfer a packet to its receiver via one intermediate node (the router).

Peer protocol layers can send information to each other by attaching a protocol-specific header (and/or trailer) to the data that is transferred. Each protocol may add such a header to a down going packet, resulting in a chain of headers attached to a data packet. In the up going flow, each protocol strips its header from the packet and interprets the available information for processing. Figure 1.1 shows how the protocols at the sender add headers (respectively a TCP, IP, and Ethernet header). The router only strips the Ethernet and the IP header, since it forwards the packet at the network (IP) layer. The router adds new IP and Ethernet headers to the packet in the down going flow.

## 1.2 Context

The context of this dissertation and the challenges in it are formed by network services and their requirements to the underlying system software, i.c. protocol stacks. Network services include web page access on the Internet, e-mail hosting, offering the latest news facts, information retrieval, etc. The most wide-spread collection of network services is formed by the Internet, with millions of servers spread out over the world, and a multitude of clients connected to them. Yet, also other types of network environments are subject to network service requirements: local area networks, in-home networks [HAV03], or wireless ad-hoc networks [Met99], which are created dynamically when network nodes come in each other's neighborhood.

System software is a generic term referring to any computer program or library code intended to support the execution of the computer system and the application software, which in turn helps solving user problems directly. Specific kinds of system software include such sub-systems as the core operating system, protocol stacks, file systems, and device drivers.

The relevance of this dissertation is confirmed by three trends in network services and their execution environment. First of all, network (Internet) services have become critical both for driving large businesses as well as for personal productivity. Global enterprises are increasingly dependent upon network-based applications (e.g. for e-commerce, supply chain management, or financial accounting). Yet, many individuals consider e-mail and Internet access as indispensable too. This growing reliance on network connectivity emphasizes the need for robust software, also at the level of system software.

Secondly, the Internet has become a highly dynamic environment with respect to the available network bandwidth, the number of parallel requests to process, the type of services required and also with respect to the service quality provided. It becomes more and more difficult to develop a single type of protocol stack that can be fine-tuned to all these dimensions of variability (e.g. by setting some parameters) [KMC+00]. Moreover, even if the system might be customizable by setting parameters, network variances are often short-term, which obsoletes manual intervention [MR96]. By consequence, it would be more appropriate to develop a flexible protocol stack that is able to detect various changes in the environment and adapt itself to handle them.

Thirdly, more and more client devices on the network can be of any type as opposed to the traditional personal computer. Already a wide variety of networked client devices can be found, such as personal digital assistants (PDAs), mobile telephones, or vehicles equipped with wireless communication channels [KMD+99]. Yet, these types of clients typically have less processing resources available than, for instance, a personal computer [IYT95]. By consequence, it is important to be able to customize the system – for instance by providing no more functionality than required by the services running on the device. In addition, the available resources should be utilized as efficient as possible, by minimizing overhead and by scheduling processing resources intelligently.

> *These three trends – growing reliance on network connectivity, highly dynamic environment characteristics, and a wide range of client devices – reveal a common challenge for protocol stack software: offering support for flexibility at design-, development-, and execution-time, tailored to application- and/or network-specific characteristics.*

## 1.3 Towards flexible protocol stacks

The challenge to develop flexible protocol stacks originates from two main shortcomings in traditional protocol stack designs. First of all, protocol stacks are often designed as a composition of coarse-grained blocks of code (e.g. protocol layers) with ambiguous responsibilities. Its monolithic character makes it very difficult to customize a protocol stack (e.g. by changing specific functionality inside a particular layer). Also, testing the behavior of a protocol layer is difficult if the internal

code is not sub-divided into independent modules. A second major shortcoming is that the behavior of a protocol stack is often fixed at design-time and fine-tuned to handle a pre-determined load, parallelism, service type or provided service quality. Unfortunately, it is very difficult for protocol stack software to handle major deviations from the average behavior.

It is our belief that **an appropriate design method and an elegant software architecture** are required to manage the complexity inherent to protocol stack software, and to deal with dynamic network behavior. Although state-of-the-art software engineering principles (such as object- [Boo94] and component-oriented programming [Szy98], or open implementation [KLL+97]) provide guidelines to develop flexible software, all too often these guidelines are neglected during development [LH89]. By consequence, the resulting code becomes inflexible for adaptations, which increases the chance for errors if code is adapted anyway [Yok92]. Protocol stack developers therefore should have at their disposal an infrastructure that enforces such flexible software design.

Furthermore, we focus on three crucial challenges in the development process of flexible protocol stacks: software (self-)adaptation, testing and optimization. In addition, the infrastructure that enforces flexible software design should be embedded in the overall software development process.

The first challenge lies in software **adaptability and management**, both at design- and run-time and both concerning functionality and concurrency. Concurrency control and management, for instance, are crucial to optimally allocate and distribute processing resources. Moreover, since environment changes (e.g. network traffic bursts) are typically short-term and unpredictable, the protocol stack must be self-managing, i.e. able to detect and handle critical situations.

Secondly, the protocol stack is a critical sub-system, which should be of high quality to prevent the system from going down due to software errors. Although self-adaptability helps in handling crisis situations gracefully, **software testing** is a key factor in accomplishing robustness. This requires a test infrastructure that facilitates test development and execution, and that encourages software developers to effectively write the test code along the development process.

Thirdly, **optimization** of the overhead inherent to flexibility support is crucial, particularly for system software on which many applications rely. While run-time adaptability can improve performance in specific situations, it often leads to overhead. The third challenge thus lies in finding an acceptable balance between this overhead and the benefit of adaptability support, especially in times of normal operation when such support is not needed.

Obviously, protocol stack development faces many other challenges, which fall, however, outside the scope of this dissertation. First of all, adaptability can also be addressed from a functional point of view, i.e. changing the functionality inside the protocol stack to handle environment changes. This may be done by adding, removing, or replacing specific code at well-defined places in the protocol stack.

Secondly, adaptability and management are not only relevant to a single node, but also to a network of connected nodes. This involves, for instance, detecting bottlenecks in the network and adapting one or more protocol stacks to handle the problem. Such distributed self-healing of protocol stacks is a key challenge for ad-hoc networks where nodes may be added and removed frequently, thus obsoleting manual management. Distributed management and adaptability is also linked with functional protocol stack changes, which often must be deployed at multiple nodes atomically. For example, when packet fragmenting is added in the down going protocol stack flow of the sender, reassembling functionality must be added in the up going flow of the receiver. A third key challenge for protocol stack management is safety. Obviously, the system must be able to protect itself from malicious parties, which may try to adapt the protocol stack in a negative way. But even for trusted parties the system should assure that run-time changes make a protocol stack perform better without hampering its correct execution.

## 1.4 Contributions

The main contribution of this dissertation is that it provides a consistent answer to three software engineering challenges in protocol stack development: self-adaptability, testability, and optimizability. We refer to this solution as DiPS+ (Distrinet Protocol Stack+), since it extends and changes the original DiPS system. We summarize the contribution of DiPS+ in each of the three software engineering tracks as follows:

- **Self-adaptability.** It is crucial for a protocol stack to be able to adapt itself, given the unpredictable and varying circumstances it must handle. The dissertation proposes DMonA, the DiPS+ Monitoring Architecture, which enables to (1) monitor the system's run-time behavior, (2) suggest an appropriate solution to adapt the system when a problem has been detected, and (3) deploy this solution in the system.

- **Testing.** The second research track focuses on unit, integration, and system testing, and proposes a test framework, called DiPSUnit. DiPSUnit allows testing software modules in isolation from the rest of the protocol stack or even from the physical network. In addition, specific circumstances can be simulated concerning not only execution scenarios, but also particular environment conditions (e.g. packet loss, delay, or duplication).

- **Framework optimization.** Finally, DiPS+ must be able to intelligently master the overhead that is inherent to run-time flexibility. This dissertation proposes a way to identify those places in the design where run-time flexibility – as offered by DMonA – is not needed. With this information, the protocol stack can be optimized to offer flexibility only where necessary,

while at all other places support for flexibility is removed. This provides a best-of-both-worlds solution.

The strength of this dissertation is that these three challenges are not proposed as individual solutions, but share a well-defined software architecture [SG96] and component framework [SN99] underneath. We will refer to them as the DiPS+ architecture and component framework. DiPS+ enables to apply the three proposed solutions not only at software design-time, but throughout the complete software life-cycle, i.e. also at build-, run- and management-time.

Indeed, the design of a DiPS+ system consists of independent components, which allows to develop and easily restructure protocol stacks in a highly incremental fashion. From day one of programming, all these components can be tested uniformly in varying circumstances using DiPSUnit. Having a test framework may increase the confidence to carry out software changes by enabling to re-execute all available tests after each adaptation. Doing so, a test framework also supports an incremental style of software development. At protocol stack build-time, optimization hot-spots can be identified and described, allowing the DiPS+ framework to build an optimally performing version. Finally, once the software runs, the DiPS+ framework relies on DMonA for monitoring and fine-tuning the protocol stack.

## 1.5 Overview

The dissertation is structured as follows. Chapter 2 discusses three key problems in today's system software and provides the relevant and necessary software engineering background to understand the research challenges we propose to address. In addition, the chapter provides the state-of-the-art in research related to each of these challenges, and situates how this dissertation fits into that picture.

Chapter 3 presents the DiPS+ component framework, which forms the basis for the three following chapters. It presents the DiPS+ architecture and design, while showing how the framework enforces this design and allows to build protocol stacks from an architectural description.

Chapter 4 first explains how we have developed a self-adaptation architecture on top of the DiPS+ basis. This architecture (DMonA) can monitor the protocol stack's behavior and deploy solutions in the DiPS+ core. The second part of the chapter presents a case study of processing load management and shows the DMonA benefits in terms of throughput.

Chapter 5 addresses common software testability problems from a DiPS+ perspective. With that in mind it presents DiPSUnit, the unit test framework for DiPS+, and shows its benefits for testing a protocol (stack) in isolation.

Chapter 6 analyzes the overhead introduced by the DiPS+ framework, and explains how this overhead can be controlled while preserving the flexibility offered by DiPS+ only where needed.

Chapter 7 situates our approach with regard to related research.

Finally, Chapter 8 summarizes the main achievements presented in this dissertation, and explains how DiPS+ has been validated in realistic cases. In addition, it demonstrates that the DiPS+ approach is applicable to other application domains than protocol stacks (i.c. device drivers and file systems). The chapter ends by identifying current and future research tracks that spin off from this dissertation.

*"If we knew what it was we were doing, it would not be called research, would it?"*
*– From Albert Einstein.*

# Chapter 2

# Background

This chapter starts by identifying important problems in traditional development of protocol stacks, concerning the software architecture and design as well as the considerable requirements posed by network services. By way of background, Section 2.2 introduces relevant software engineering concepts and defines the terminology that is essential to understand the following chapters. Finally, Section 2.3 describes the research context for three key software development challenges and delimits how the research in this dissertation fits into that picture.

Although some of the problems and characteristics presented are applicable to (system) software in general, this chapter describes them consistently from a protocol stack point of view. Nevertheless, we are convinced that our approach is applicable to a broader context than protocol stacks. As Chapter 8[1] shows, it may be applied to two other types of system software with highly similar problems and characteristics: device drivers and file systems.

## 2.1 Key problems and characteristics of protocol stack software

Protocol stack software is generally accepted to be highly complex. This complexity derives from the fact that this software is used in very different circumstances depending on the load of the system, the applications running on top of it, or the hardware that is used. Software characteristics such as adaptability, extensibility, reusability and robustness are highly relevant to (system) software. This dissertation aims to address some common problems of traditional protocol stack software

[1] More specifically, Section 8.3 (page 141).

and its characteristics. In order to focus discussions later on, this section provides a short overview of these problems and characteristics.

### 2.1.1 Software design and architecture

#### Poor separation of concerns

Software programmers obviously have to implement the system's functional properties, i.e. *what* the system does. Additionally, they sometimes have to cope with non-functional behavior, i.e. *how* the system does it. Especially protocol stack software typically incorporates multiple non-functional concerns such as concurrency, throughput, testability or robustness. Unfortunately, the code for such non-functional concerns is often scattered across the functional code [KLM+97], which complicates their control and implementation, and distracts programmers from programming functionality.

#### No hierarchical view

All too often, protocol stack developers do not have at their disposal a well-defined software architecture that helps them understand the overall system structure, and that matches the software design close enough to switch back and forth between the architectural and the design perspective [GP95]. By consequence, programmers cannot zoom in or out to an appropriate level of detail. On the one hand, too much detail diverts attention from the overall semantics and structure. On the other hand, an architecture description that only identifies coarse-grained components is ineffective because essential details are lost.

#### Limited reusability

The lack of attention for generic patterns or abstractions gives rise to ad-hoc software, which is hardly reusable in a different context or in the next version of the system with slightly different functionality [Joh97]. Modularization, which is typically seen as an advantage towards reusability [JR91], is less effective when modules are coarse-grained, since it prevents subtle adaptations. By consequence, high-level patterns and abstractions are continuously rediscovered and re-implemented, which hampers productivity.

### 2.1.2 Extreme requirements

#### Poor customizability

Protocol stacks are black-box entities, which makes them very difficult to customize to application-specific requirements or device-specific characteristics. This becomes a serious problem as the Internet becomes accessible from a wide range of

client devices with highly varying resources available. For mobile devices such as personal digital assistants (PDAs) or wireless telephones, it is crucial to customize the protocol stack. To save memory resources, for instance, one should be able to remove specific functionality from the protocol stack when it is not used by any application. Such flexibility is also highly relevant to Internet terminals[2] [ITE99], which offer a user personalized access to subscribed Internet services (e.g. Internet telephony, file transfer, web access, etc.). In this case, a protocol stack should not only be customized to the resource characteristics of the terminal device it will be running on, but also to the services required by the user. Providing all terminals with all potentially needed protocols is simply impossible. Yet, Internet telephony, for instance, requires totally different protocols and network service quality than file transfer.

#### Tuned for performance

Protocol stacks are typically developed having in mind a specific application service domain with specific performance requirements and characteristics. The first problem of this performance ambition is that defining the average performance requirements of an application domain is far from trivial. Statically configured protocol stacks often lead to well-performing systems for the average type of applications. Yet, for applications with non-average characteristics, such systems perform poorly. Secondly, performance requirements often call for code adjustments across the functional code, leading to ad-hoc, difficult to understand and error-prone software [KLM+97].

#### Unpredictable load conditions

Protocol stacks should be able to cope with highly variable load conditions and yet perform optimally in all cases. Most protocol stack software, however, is designed to handle a limited range of supported load circumstances. Typically for highly loaded servers, incoming requests are distributed over a pool of server machines to increase parallelism. Such network parallelism is not powerful enough to handle extreme load conditions, since this would require too many parallel machines to be cost-effective [Wel02].

## 2.2 Software engineering background

To understand our perspective on software design and architecture, this section presents an overview of relevant software engineering techniques. First of all, it

[2]Internet terminals are placed at public places (airports, train stations, cinemas, etc.) and are activated by inserting a chip card that has stored all user-specific information, such as the subscribed services.

describes the evolution of software development support, from programming to engineering, by presenting five state-of-the-art paradigms: procedural programming, abstract data types, object-oriented programming, design patterns, and object-oriented frameworks. After that, the section focuses on software architectures and components, as these two paradigms form the foundation of this dissertation.

### 2.2.1 From programming to engineering

Throughout the years, software development has gradually evolved from programming towards engineering. Programming techniques mainly concentrate on source code reuse to support software maintainability, adaptability and extensibility. Software engineering augments these techniques by helping programmers to reuse the structure of the system in addition to the source code.

#### Procedural programming

During the 1950s, several procedural programming languages have been developed [KR88, Wel79]. Procedural programming is a software programming paradigm based upon two main concepts: unit and scope (i.e. the data viewing range of an executable code statement). A procedural program consists of one or more units or modules – either user coded or provided in a code library. Each module is composed of one or more procedures, also called functions, routines, subroutines, or methods, depending on the programming language. It is possible for a procedural program to have multiple levels or scopes, where each level may contain local variables which are inaccessible from outer scopes.

An often-used procedural programming language for writing system software is the C programming language [KR88]. Both its modularized style of programming and its openness to low level hardware features (such as the use of pointers to refer to locations in memory) made the C language very useful for system programming. Around 1970, for example, it was used to write the UNIX[3] operating system [RT78].

#### Abstract data types

Progress in language design provided more and more flexibility at design-time with the introduction of modules that provide access protection for related procedures and data structures. To this end, the interface – i.e. the specification – was separated from the underlying implementation and abstract data types (ADT) were introduced. ADTs are data types described in terms of the operations they

[3] Unix consists of an operating system and numerous applications. Since its original inception, Unix has been extended with new applications and system interfaces, and adapted to new and evolving requirements concerning security and networking. The availability of UNIX source code (e.g. the Berkeley TCP/IP [Ste00] protocol stack) has been one of the driving forces for the birth of the Internet.

support (their interface) rather than how they are implemented. For example, arrays, linked lists and binary trees – among other data structures – can all support add/remove/select operations, but their implementations can be highly different (e.g. with specific performance characteristics). ADTs raised the design level of certain elements above the level of programming language statements or individual algorithms.

Despite this progress, traditional procedural languages show some major limitations, such as difficulty in extending and specializing functionality, difficulty in factoring out common functionality for reuse, inter-operability, and maintenance overhead. These limitations have forced the software community to look for new approaches to software programming.

#### Object-oriented programming

The object-oriented programming (OOP) paradigm presented a new approach to facing the challenge of building large-scale applications. OOP extends the idea of data encapsulation and presents a software system as a collaboration of modules (objects) that each encapsulate their internal state and that execute a common task by invoking each other's interface. This paradigm forms the basis for many of today's software systems and paradigms. An object is similar to a value in an abstract data type; it encapsulates both data and operations on these data [JF88]. In this way, object-oriented languages provide modularity and information-hiding, like conventional data abstraction languages. However, two features concerning software reuse distinguish an object-oriented from an ADT-based language, and are the main reasons for the success of OOP: class inheritance and polymorphism [JR91, Opd92].

Class inheritance stands for the sharing of attributes and operations between a class and its sub-classes [AC96]. It allows to create a *class hierarchy* in which each sub-class inherits the operations and the internal structure of its super-class(es). A sub-class can define new operations or it can redefine existing ones. Inheritance promotes code reuse, since code that is shared by several classes can be placed into a common super-class, which sub-classes can inherit from. The programmer can define a new class by choosing a closely related super-class and simply describing the differences with that super-class while reusing the similarities. This style of programming is called programming-by-difference [JF88].

Polymorphism allows an object to interact with other objects of different types as long as they have the same interface [Boo94]. In other words, a method can be invoked on an object without knowing its exact sub-type. Because a polymorphic method works with a wider range of attributes, it is easier to reuse than a non-polymorphic one. A method that visualizes some type of elements in an array, for instance, will work correctly whenever all elements understand some "visualize" message, irrespective of the specific sub-type of each element.

All these features (information-hiding, inheritance and polymorphism) have

introduced a true revolution in software programming techniques. Key benefits of OOP are enhanced reusability, maintainability, adaptability and extensibility of software [Rus90]. Yet, even if objects and classes allow programming at a higher level of abstraction, the programmer is still responsible for providing the structure and the flow of control of the application. In other words, OOP benefits – and especially reusability – are mainly limited to the class level. Another important approach to software reuse involves reusing the architecture or the functional decomposition of software (within a particular application domain) [Deu89]. This idea of *design reuse* has lead to design patterns and object-oriented frameworks.

### Design patterns

A design pattern refers to a commonly recurring problem to be solved, along with a solution and the context in which that solution works. By publishing such patterns, one tries to prevent that designers re-invent the wheel over and over again. The current use of the term pattern in OOP is derived from the writings of the architect Christopher Alexander who used it in the context of urban planning and building architecture [Gab96, AIS77]. Yet, the idea of applying generic patterns to solve recurring problems is applicable to many other disciplines than urban planning and building architecture, including software development [BMR+96, GHJV95, Cop96].

Within the OOP community, the use of design patterns has become state-of-the-art. The book *Design Patterns* [GHJV95] defines a design pattern as follows:

> *A design pattern is a description of communicating objects and classes that is customized to solve a general design problem in a particular context.*

Although this definition focuses on object-oriented design, it could be readily adjusted to describe software design patterns in general (by simply replacing object and class with module or sub-system). A design pattern is an abstract description of a combination of objects (a design) that can be reused and combined with other design patterns to create an application (code). For example, a GUI (Graphical User Interface) application can be designed as a combination of three major design patterns (and several less important ones) [GHJV95]) often referred to as Model/View/Controller (MVC) [BMR+96]. In this way, the core model (application logic) is separated from different views on it by introducing a controller that translates user-specific actions to the model. In addition, it uses the "Observer" pattern to ensure that the view on the model is consistent, the "Composite" pattern to nest different views, and the "Strategy" pattern to make views delegate responsibility for handling user events to their controller.

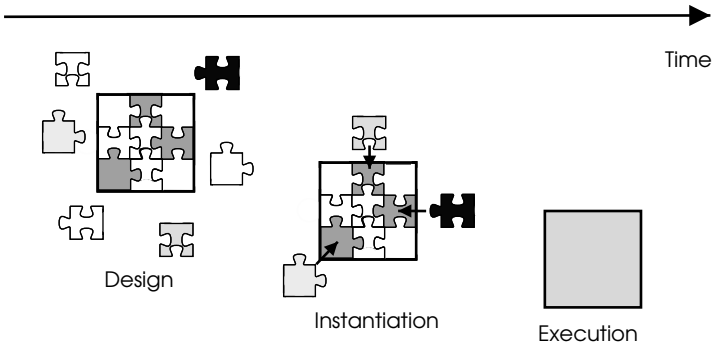



Figure 2.1: An intuitive illustration of an object-oriented framework, which is no longer present at execution-time.

### Object-oriented frameworks

Object-oriented frameworks describe how a system is decomposed [Deu89] and how its "components" interact. They allow the reuse not only of the design (as design patterns do), but also of a major part of the source code [WBJ90]. A framework is a collection of abstract and concrete classes (code) along with a design that describes how to combine objects to build a working system [JR91].

In this dissertation we use the definition from [Joh97]:

> *A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact.*

The purpose of a framework is to present a skeleton of an application that can be customized by an application developer. A framework enhances extensibility by providing explicit "hook methods" [Pre95] that allow applications to extend its stable interfaces. The framework life-cycle is illustrated in Figure 2.1. A framework is domain-specific [JF88] by nature because it represents part of the application (domain) logic. As a result, frameworks are less generic than design patterns, but offer more support to the application programmer.

The idea to identify common properties is similar to that of program families, already defined in the 1970s by Parnas as "sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members" [Par76]. Finding common abstractions, however, is far from trivial. In general, abstraction seems to be discovered by generalization from a number of concrete examples [JF88]. The same applies to frameworks, which often result from concrete implementations instead of being designed and developed from scratch [RJ97].

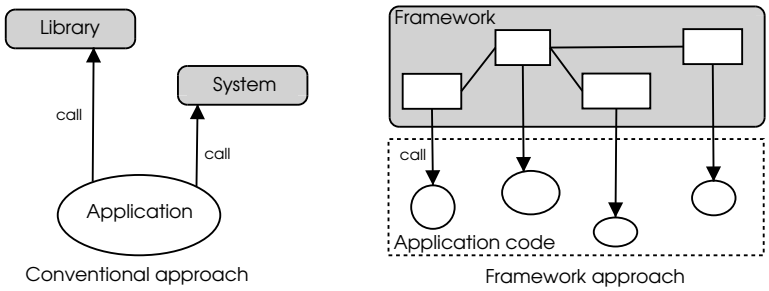



Figure 2.2: Inversion of control: in the conventional approach (left), application code calls generic library and system code; in the framework approach (right), application code is called by the generic framework, which controls when and how application code is executed.

The application-specific code to customize the framework is often called from within the framework itself rather than from application code. This *inversion of control* gives frameworks the power to serve as extensible skeletons [FS97]. The framework often plays the role of a coordinator that sequences application activity (comparable to a program's main method). However, this inversion of control makes applications written with frameworks hard to debug since the flow of control oscillates between framework and application code [FS97]. Inversion of control between the conventional and the framework software development approach is illustrated in Figure 2.2.

Frameworks can be classified based on how they can be customized or extended. The first class of frameworks (also called *white-box frameworks*) provide abstract classes to be sub-classed (inherited) by an application programmer [JF88, FS97]. They are called white-box because the application programmer must know and fully understand the internal design and architecture of the framework to be able to extend it. Yet, it is difficult to extract the implicit architecture from an object-oriented design since it only exposes the class hierarchy, not the object interactions [SN99]. This implies a long learning curve and, as a result, a higher risk for introducing errors (e.g. because one misunderstands how to properly extend the framework) [BMM+99].

Another way to customize a framework is by providing application-specific code as well-separated software modules ("components") and hiding the internal framework structure to developers. Accordingly, this type of frameworks is called *black-box framework* [JF88, FS97]. Each provided "component" must understand a specific protocol that defines how components can be connected. To this end, the protocol often specifies the communication interface of the "components" to be able to plug them into the framework. Black-box frameworks replace "reuse by inheritance" with "reuse by composition". Application programmers only have

to learn the protocol (i.e. the external interface of the components) and not the internal framework structure, which makes black-box frameworks much easier to understand and learn than the white-box type. Moreover, generic "components" can be reused in different applications or application configurations as long as the protocol is followed. However, black-box frameworks are more difficult to develop since they require framework developers to define interfaces and hooks that anticipate a wider range of potential use cases [HJE95].

Few *pure* white-box or black-box frameworks exist in practice [BMM+99]. A framework is often inheritance-based in the beginning of its life-cycle, when the application domain is not sufficiently understood to allow parameterizing the behavior (as black-box frameworks do) [Joh97]. White-box inheritance frameworks should be seen as a natural stage in the evolution of a system. Ideally, each white-box framework eventually evolves into a black-box framework [JF88].

### 2.2.2 Software architectures and components

Another maturization process, similar to the evolution from white-box to black-box frameworks, has occurred in the area of both structure descriptions and programming paradigms. This evolution has lead to software architectures and components. The combination of these two has recently resulted in component frameworks.

#### Software architectures

Software architectures [SG96] describe the overall structure of a software system. Software architectures are very similar to object-oriented frameworks. Especially black-box frameworks impose a kind of domain-specific architecture [Joh97, VK89], yet not every software architecture implies a (black-box) framework. Moreover, a framework is an object-oriented design, while a domain-specific architecture might not be. A software architecture can be defined as follows [GP95]:

> *The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.*

The main purpose of software architectures is to make a clear separation between computational elements (components) and their relationships (connectors) [SG96]. An architecture description can be thought of as a graph in which the nodes stand for components and the edges for the connectors. Architectural components represent computational entities such as (Java) objects, Java Beans [Sund], Enterprise JavaBeans [Sun03a], or COM technologies [Mic02]. Connectors represent interactions as varied as procedure call, event broadcast, indirect blackboard communication, etc.

An architectural style (sometimes called component model [BBB+00]) is an abstraction over a *family* of systems in terms of a pattern of structural organization [GP95]. More specifically, an architectural style determines the vocabulary of

components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined (for instance, no cycles allowed) [PW92]. Examples of architectural styles are pipes and filters, data abstraction and object-oriented organization, event-based implicit invocation, blackboard (repository-based), layers, or a combination of these [GS93].

An architecture description language (ADL) is a notation that allows for a precise description of the externally visible properties of a software architecture, supporting different architectural styles at different levels of abstraction. Externally visible properties refer to the assumptions other components can make about a component, such as its provided services, performance characteristics, error handling, and shared resource usage [BCK98].

The architectural view on software zooms out from all details, which is important to software development [BMR+96] for three reasons. First of all, it simplifies our ability to comprehend large systems by abstracting away code and focusing on the big picture [OMT98]. Secondly, it clearly encourages reuse of both components and structure. Finally, it promotes software evolution by separating functionality in (architectural) components from the way components are interconnected. This allows changing the connection mechanism to handle evolving concerns about performance, inter-operability, prototyping and reuse [GP95].

To be effective, however, the architectural view needs to be mapped onto the functional view (the actual code). Only when this mapping is clear, developers are able to zoom in and translate the architecture into a functional design.

At the structural level, evolution of OOP has shown a shift from white-box inheritance-based frameworks towards black-box composition-based frameworks and software architectures in general. At the functional (programming) level, we can see a similar shift from objects to software "components". A software "component" is the programming equivalent of an architectural component, which represents a computational entity in the global structure of the system [Mat99]. In this way, a clear mapping between the functional and the architectural view has been realized.

### Component-oriented programming

Although the ideas of modularization and reuse behind component-oriented programming (COP) [Szy98] are certainly not new (Parnas already presented modularization in the 1970s [Par72]), they do confirm the next step in the maturization of OOP [MN97].

Ultimately, COP aims at building a software system through composition of black-box software packages. A text processor, for example, can be composed of a text processing package, a spell-check package and a thesaurus package. Extending the text processor should be limited to integrating another package into the existing system. Such black-box packages are usually referred to as *components*. However, many component definitions exist and terminology differs depending on

the domain (level) it is used in. While a component at the architectural level is simply defined as a functional entity, a more stringent definition is needed when zooming in to the functional level.

To avoid any confusion we will use the well-established Szyperski definition of a component in the domain of COP [Szy98]:

> *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.*

This definition implies that components are designed for composition [ND95, Szy98]. However, it is clear that additional support is needed for component composition. On the one hand, the definition mentions that a component must offer well-defined interfaces in order to enable communication with it. On the other hand, the definition implies that a component should not be used in isolation (cfr. "unit of composition"). Yet, components should be employed according to a software architecture that determines *how* components should be composed into a software system [SN99].

Having a software architecture (or component model) that describes the structure of a software system is not enough. Architectures do not determine how to connect components or how to configure connectors in order to deal with system-wide issues. What is needed, therefore, is some infrastructure at run-time that enforces how the system is instantiated and that controls and manages its run-time behavior. This need inspired the idea of component frameworks.

#### Component frameworks

A component framework in itself is not modified by the components, but accepts component instances as plug-ins [BSW98]. The definition of [SN99] matches best our view on component frameworks:

> *A component framework is a collection of software components and architectural styles that determines the interfaces that components may have and the rules governing their composition.*

A component framework is typically used in situations where strict rules and policies must be enforced in order for the system to work correctly. In that sense, component frameworks are similar to object-oriented frameworks. In contrast to white-box frameworks where applications are generally built by sub-classing specific framework classes, component frameworks primarily focus on component composition, similar to black-box frameworks. The main difference between a black-box and a component framework, however, is that the latter is present both at design- and at run-time, while (black-box) frameworks are design-time entities

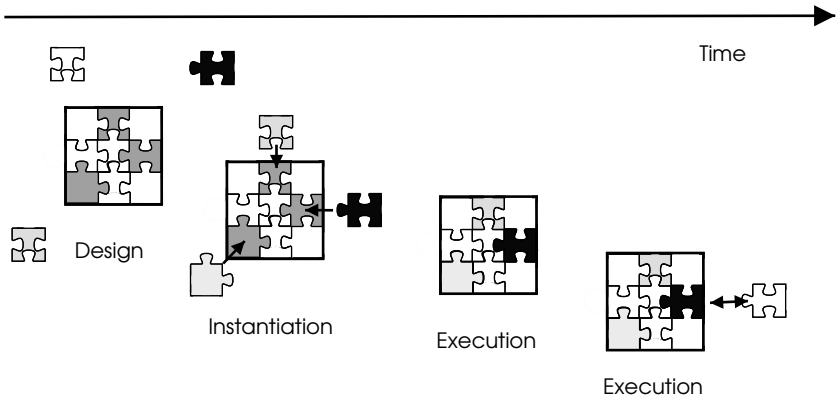



Figure 2.3: Intuitive illustration of a component framework, which is not only present at design- and instantiation-time, but also at execution-time.

that disappear at run-time. As a result, a component framework can easily deal with changes at run-time, as is illustrated in Figure 2.3.

After its conception in the late 1980s, object-oriented frameworks have attracted many researchers and software engineers. Frameworks have been designed and implemented in several application domains such as user interfaces [LVC89, WGM89], but also in system software such as operating systems [Rus90, Mad96] in general, and later they have been applied to specific operating system sub-domains such as protocol stacks [Mat99, Koh01, HP91] and file systems [Mad92].

## 2.3 Research challenges

Having sketched the background for our research, this section zooms in on three challenges in the software development process – self-adaptability, testing and framework optimization – and presents our view on each of them in the context of this dissertation. Figure 2.4 illustrates the overall view of the dissertation: a well-designed software architecture and component framework, which supports and enforces a style of programming that facilitates adaptability, testing and framework optimization.

**Self-adaptability.** From the literature overview we can conclude that component frameworks can be a means to support both design- and run-time adaptability. The ideal component framework should enforce a clear separation between functional and concurrency concerns, enable to inspect the running system and allow to change its behavior when problems are detected. In this way, the component framework itself can manage the running system and adapt it on the fly.

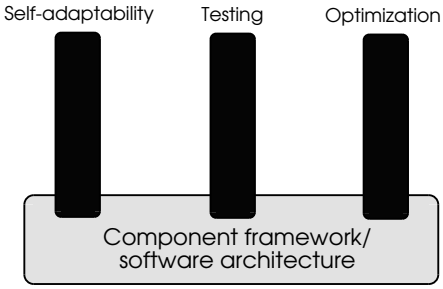



Figure 2.4: General scope of the dissertation: a well-designed component framework with an associated software architecture that supports three crucial challenges of software development – self-adaptability, testing and framework optimization.

**Testing.** A recent trend in software development is to search for effective testing approaches [Amb02]. Research in this area tends to shift testing efforts from the end of the development cycle towards the very beginning, i.e. day one of software development [Bec99, BG98]. Since this trend places testing in the middle of the day-by-day software development process, it must be highly iterative – to follow the evolution of the design – and relatively lightweight – not to overload developers with testing their code.

However, iterative and lightweight testing is not feasible without a component framework that enforces a "testable" component design and a test framework that facilitates the testing process. Testability means that components, whether fine- or coarse-grained, can easily be tested in isolation using a uniform approach. A test framework must cope with internal active (concurrent) behavior and allow for environment control to simulate specific situations.

**Framework optimization.** Obviously, system performance is, and will always be, a key characteristic, especially for protocol stacks. More than ever, developers must find a balance between two contradicting concerns: the overhead introduced by the component framework that is used to develop software (e.g. overhead to support run-time adaptability), and the performance of that framework – and by consequence of the system.

### 2.3.1 Adaptability

We define software adaptability as "the ease by which a system or parts of it can be tailored to changing requirements" [Tek96]. In particular, we concentrate on adaptability in the context of protocol stacks [KLM+93], as this allows us to present our objectives of providing adaptability more clearly. Obviously, some adaptability concerns are also applicable to software in general.

First of all, we point out the need to have adaptable protocol stack software in both the functional and the non-functional (behavior) domain. Secondly, four relevant software adaptability dimensions are presented: adaptation concern, time of adaptation, adaptation predictability, and adaptation initiator. Thirdly, next to these adaptability dimensions, five state-of-the-art adaptation mechanisms are presented, which taken together provide an overview of the most relevant adaptation approaches today. Finally, we identify the challenges concerning adaptability this dissertation will handle by positioning our research along each of the four adaptability dimensions.

#### Need for adaptable protocol stacks

The authors of [Cah96] identify a number of relevant problems that can be addressed by making (operating system) software more flexible, i.e. adaptable to varying circumstances. The problems listed here are divided into two categories: functional (feature) mismatch and non-functional (behavior) mismatch. The dissertation focuses on behavioral mismatch (i.c. performance, and concurrent behavior).

**Feature mismatch.** A feature is a functional property offered by a software component (or system). A feature mismatch is an imbalance between the effective and the expected functionality. Three kinds of feature mismatch are relevant: feature deficiency, feature abundance, and version mismatch. In the first case, the system does not provide a feature that is required by an application. The second case occurs when the system provides superfluous services that, although not used by any application, result in (avoidable) overhead. The third case, version mismatch, occurs when the system provides a different version of some service than is expected or preferred by an application. This has become increasingly relevant in the fast evolving software development industry. Indeed, software errors are often the rule rather than the exception, partly because the "time to market" – i.e. the time between the start of design and the product release – has become very short. As a result, software updates are published, and even automatically downloaded and installed, soon after the main release [Mic03b]. Obviously, this increases the possibility of a software version mismatch.

**Behavior mismatch.** The system's behavior reflects its non-functional properties [RMJ+98], i.e. *how* it executes rather than *what*. Behavior mismatch occurs when the system does not behave as expected or desired by an application. Expected behavior is typically expressed in terms of response-time or service quality. On the one hand, behavior mismatch can be caused by a user, for instance by changing its requirements. For example, the user of a multimedia application may vary the expected quality level of an audio/video stream (such as the number of bits per pixel, screen size, compression level, etc.). Obviously, this impacts the service quality level expected from the system (for instance, the application should receive more processor-time). On the other hand, internal system conditions can

cause behavior mismatch too. The system's processor load or available memory, for instance, do influence its behavior, even when applications do not change their expectations [WCB01].

Performance mismatch is an important example of behavior mismatch that is highly sensitive to the internal system state (processing load and available memory). Performance mismatch occurs when the system as a whole or a specific sub-system offers performance that is unacceptable for applications or users. Systems should be designed to solve performance bottlenecks. Often, however, performance mismatch cannot be solved by replacing one or more individual services (components), because the problem is spread throughout the system and needs to be handled by system-wide measures.

### Adaptability dimensions

Software adaptability has many different faces. We propose an adaptability taxonomy based on four dimensions: adaptation concern, the time of adaptation, adaptation predictability (anticipated or non-anticipated), and the adaptation initiator.

**Adaptation concern.** Separation of concerns (SoC) is a well-accepted software development principle [KLL+97, KLM+97]. The idea behind SoC is that different non-functional concerns (such as concurrency, distribution, fault-tolerance, security, etc.) should be represented as well-identifiable entities that are clearly separated from functional code. This allows software developers to concentrate on one specific concern at a time, instead of having to address all of them simultaneously.

Having non-functional concerns separated from the functional code, makes it easier to adapt a single concern. In view of that, we differentiate between functional adaptations and non-functional adaptations. Functional adaptations involve changes in *what* the system does. Examples of functional adaptations are instantiating a framework with application-specific code, replacing, adding or deleting computational code, replacing a decision strategy, etc.

On the other hand, non-functional adaptations change *how* the system behaves. This includes changes in concurrency level, communication patterns, self-healing behavior, load balancing or quality of service. Non-functional adaptations typically address two or more concerns. For example, changing concurrent behavior obviously involves the concurrency concern, i.e. how much parallelism is appropriate at each moment in time. At the same time, it affects the functional concern, i.e. where in the code should parallelism be introduced. Similarly, non-functional concerns may influence each other. The security concern, for instance, may be influenced by the distribution concern. Indeed, it is likely that security policies differ between a stand-alone and a distributed system setup. This "multi-concern effect" (also referred to as "feature interaction" [PSJ+01]) makes non-functional adaptations highly complex and requires a well-defined software design and architecture

[KLL+97, OMT98].

**Time of adaptation.** Another important dimension of adaptability concerns the time of adaptation. The adaptation time-line goes from static adaptations at design-time to dynamic changes at run-time. In between these two extremes, we identify adaptation at system build-time. The further on the time-line adaptations occur the more powerful they become, since more knowledge is available about the execution context. However, run-time adaptations are more difficult because the system may be processing data while it is changed. For example, it is far from trivial to replace an executing component when its internal state must be transferred to the new component [JMMV02].

Design-time adaptability requires a structure-reuse aware infrastructure, such as design patterns or (component) framework technology, in combination with a programming language that allows code reuse as well. A system can be customized at design-time, for instance, by choosing between specific sub-system implementations, or by integrating new sub-systems that extend its functionality.

Build-time adaptations rely on information that is only available when the system is being built and all parts are loaded into the system. These parts may, for instance, be selected based on application-, user- or environment-specific requirements, downloaded from a network, and integrated in the system at build-time.

Dynamic (i.e. run-time) adaptability allows protocol stacks to be customized to the current needs of applications or users during execution. Note that run-time adaptations cannot occur arbitrarily at any time, since they may hamper the system's correct functioning. By consequence, the system transformation that is associated with a run-time adaptation is postponed until the system has reached a safe state [GK96].

**Adaptation predictability.** Adaptation predictability defines to which degree one can predict what changes are allowed and possible. We differentiate between anticipated and unanticipated changes [GK96]. The former type assumes that what can be adapted is known in advance. Typically, adaptability can be anticipated in the design. For example, an object-oriented framework defines hot-spots for applications to extend or customize the framework, a class can present setter methods for specific parameters, or the design may allow to install different strategies for changing specific behavior (cfr. the "Strategy" design pattern [GHJV95]).

Unanticipated changes, on the other hand, are not taken into account at design-time explicitly. Although the system must be designed to cope with such unanticipated changes, it does not have further knowledge about the specific type of adaptation, its location, or the time it will occur. Integrating a new component at run-time, for instance, is an unanticipated change since both the component and where in the system architecture it would be added were unknown at system design-, load- and build-time. This is in contrast to the Strategy design pattern, where the type of the class that is added and the place in the design are known at

design-time.

**Adaptation initiator.** The trigger to change can be external or internal. External adaptation initiators may be system administrators or application users who are allowed to tune the system. Internal triggers come from the system itself. In this case, the system is responsible for detecting the need to change, for choosing an appropriate solution and for deploying it. Such systems are often referred to as self-adaptive [SS97] or self-healing [GKW02].

### Adaptation mechanisms

Research concerning adaptable operating systems has been dominated by the transition from so-called monolithic operating systems to systems composed of separate modules. We provide an overview of the most relevant adaptation mechanisms, which helps understanding the decisions we make and the solutions we propose in the chapters to come.

**Micro kernels.** Compared to traditional monolithic operating system designs, micro kernels offer increased support for adaptability by allowing large parts of the system to be implemented by (user level) servers. This improves flexibility since modifying features of an operating system implemented as servers is simpler than modifying integrated features of a monolithic kernel. However, the granularity of such adaptations is quite coarse, since the unit of replacement is typically a server. As a result, all clients are affected by such adaptation. Moreover, in the absence of development support, the effort necessary to specialize or change a service may be high compared to the amount of change required. Often, changes have to be programmed from scratch [Cah96]. Nevertheless, micro kernel technology was a major step towards flexibility, which resulted in well-known operating systems such as Mach [BGJ+91, ABB+86] and Chorus [ZBC+81].

**Object-oriented operating systems.** Russo [Rus90] characterizes an object-oriented operating system as one that both is implemented using object-oriented techniques and provides its services via invocations on system objects. Object-oriented operating systems research tries to apply OOP in order to benefit from its well-known advantages such as code reuse, portability, maintainability and incremental extensibility [Boo94]. For developing operating systems, however, the most important gain of using OOP lies in design reuse, as offered by object-oriented frameworks. Frameworks provide the opportunity of building highly customizable systems. The Choices operating system pioneered the use of frameworks as the basis for the design of an operating system family [Rus90].

**Open implementation.** Kiczales has proposed a model of abstraction for software engineering known as the open implementation model [KLL+97, KTW92]. This model is based on the argument that, while the traditional information-hiding/black-box model of abstraction shields clients from having to know how an abstraction is implemented, it also prevents them from altering those details when desirable [WC01]. Preferably, clients are able to exercise control over the

way the abstraction is implemented. It should therefore be possible to install an application-specific policy, which describes how the system should behave. Such a policy may, for instance, prioritize incoming network packets in a protocol stack, based on their data length. In this way, the system can be customized based on application-specific requirements, without having to expose internal system details. To this end, Kiczales distinguishes the base-interface (i.e. the API[4]) of an entity, which typically offers access to its functionality and internal state, from the meta-interface. The latter provides an abstract view that allows clients to adjust the implementation behavior in well-defined ways (e.g. by installing a policy). Many existing approaches to provide flexibility can be mapped onto the open implementation model and its separation between base- and meta-interfaces. For example, the Mach external paging interface [MA90], which allows to involve applications in page replacement policy decisions, can be seen as a meta-interface to the Mach virtual memory system.

**Meta-object protocols.** The model of open implementation is closely related to meta-object protocols (MOPs) [KdRB91]. A meta-object protocol is basically a meta-interface to the implementation of a language's object model that allows application programmers to control the implementation of certain aspects of the object model (e.g. object invocation, dispatching or inheritance) [Rob99]. Typically, a MOP is implemented using reflection. In essence, a reflective system is implemented in terms of itself. A reflective programming language, for instance, is one where the implementation of the language (e.g. a method or class construct) makes use of the language itself (e.g. by using objects, inheritance or polymorphism). One of the best-known reflective object-oriented operating systems is Apertos [Yok92]. Reflection and MOPs offer the potential of building highly run-time adaptable systems. However, MOPs tend to introduce performance overhead due to message interception. Moreover, lack of development support makes meta-level programming difficult and error-prone.

**Aspect-oriented programming.** The ideas of open implementation, separation of concerns and MOPs have lead to a new paradigm called aspect-oriented programming (AOP) [KLM+97]. Reflection and MOPs can be seen as specific implementation techniques of the AOP paradigm [Sul01]. AOP builds on the strengths of existing technologies (such as OOP) and provides additional mechanisms to support separation of cross-cutting concerns, which OOP handles poorly [EAK+01]. A concern is cross-cutting when its implementation is scattered across the functional code and cannot be modularized into a single modular unit using standard OOP techniques. Examples of such concerns are, among others, concurrency [Lop97], distribution [Lop97], security [VBC01, WVD02], performance [GSPS01, ERFL01], and data pre-fetching[5] [CKF+01]. Separated development of

[4] API stands for Application Programming Interface, which is the set of operations that a system exposes for use by (application) programmers.

[5] Pre-fetching is a performance optimization technique that amortizes the cost of going to disk

non-functional concerns allows combining specific concerns at build-time in order to customize individual run-time instances of a system to specific situations.

The goal of AOP is to make designs and code more modular, meaning that concerns are localized rather than scattered and have well-defined interfaces with the rest of the system [EAK+01]. Because a cross-cutting concern is implemented as a modular unit, it is easier to think about it as a single design unit.

Currently, various techniques exist that implement the ideas of AOP (e.g. AspectJ [KHH+01], Hyper/J [OT01], and composition filters [BA01]). On the one hand, techniques like AspectJ and Hyper/J focus on build-time adaptations, since they merge[6] non-functional concerns into the functional code at pre-compilation-time. To this end, an aspect language supports programmers to express concerns more easily. This is a considerable advantage compared to MOPs, which only provide infrastructure support to intercept incoming requests without offering the programmer an easy way to express what should be done at the meta-level. Another advantage of merging is that it omits the performance overhead introduced by intercepting and reifying incoming requests, as is done by a MOP. However, one should be aware of the other side of the picture [MJV+98]: merging code highly complicates run-time adaptability since non-functional behavior becomes intermingled with the functional code. As a result, it becomes very difficult (if not impossible) to change this behavior at run-time. Various approaches that try to combine merging and run-time flexibility have been published recently [PGA02, BH02].

On the other hand, composition filters do not change any source code, but introduce adaptation code – so-called input and output filters – around it instead. This is similar to the MOP approach, except that composition filters do provide developer support to express how to treat incoming requests. For example, a "wait filter" [Ber94] is used to block incoming requests until a set of conditions has been fulfilled. As a result, such a wait filter may be used to enforce concurrency control transparent to the functional component. A concatenation of filters can be installed to apply additional non-functional behavior. Since filters are clearly separate entities, both at design- and at run-time, it becomes much easier to change non-functional behavior compared to merging approaches.

### Research focus

The research on adaptability conducted within the scope of this dissertation focuses on solving behavioral mismatch, in particular on performance mismatch,

by predicting future data requests and retrieving additional data with each explicit disk request. In this way, chances are that future requests for data can be handled by accessing the data cache, which is much faster than the disk device. Code for pre-fetching is typically spread throughout the system, which makes it difficult to control.

[6] "Aspect weaving" is a term commonly used in AspectJ to denote the idea of merging concerns into the functional code, yet the term "merging" is more general.

and fits into the four presented software adaptability dimensions as follows. First of all, if we are to handle behavioral mismatches, research should focus on adaptability of non-functional *concerns*. In particular, adaptive concurrent behavior will be proposed as a solution. Secondly, since system performance is extremely sensitive to varying circumstances at run-time (e.g. processing load, or input/output (I/O) request patterns), and since these variances cannot be predicted at design-time, protocol stacks should be made adaptable *at run-time*. Thirdly, because one cannot predict at system design-time where or when adaptations will be needed, and performance overhead should be minimized at all times, the system must *anticipate* adding and removing management extensions when appropriate. Fourthly, since variations in run-time circumstances are highly unpredictable and short-term, only the *system itself* can appropriately detect problems and deploy corresponding solutions.

Our focus on software adaptability thus can be summarized as follows:

> Concerning software adaptability, research in this dissertation focuses on providing a software architecture to deal with behavior mismatch. More specifically, we study anticipated, run-time adaptability of concurrent behavior, initiated and controlled by the system itself. Concurrent behavior is clearly a non-functional concern that is cross-cutting the functional code.

### 2.3.2 Testing

Comparable to the previous section, this section first motivates the need for testing. After that, it proposes four testing dimensions: the test objective, the considered unit for testing, test strategies, and the test methodology. At the end, the dissertation's research focus is positioned along the four testing dimensions.

#### Need for testing

It goes without saying that testing is a crucial element in the software development process, and that its potential benefit for robustness increases as user expectations rise and software systems become larger, more complex and thus more error-prone.

Protocol stack software emphasizes the need for testing even more. First and foremost, a protocol stack is generally accepted to be a highly *complex* piece of software. Moreover, code is often tuned for performance and efficiency, which makes it difficult to understand, maintain or adapt. As a result, testing such complex protocol stack software is a far from trivial.

Secondly, internal *concurrent behavior*, which is crucial to the performance and scalability of a protocol stack, makes the software extra sensitive to software faults. Indeed, because concurrency code is typically spread throughout the functional code, concurrency becomes difficult to understand and control. Traditional

software abstractions, such as semaphores and monitors, try to support concurrency control. Yet, unfortunately, these techniques do not allow programmers to concentrate on one concern at a time [KLM+97] (i.e. functionality versus concurrency).

A third crucial issue in testing protocol stack software is *creating (or simulating) a real-life execution context*. This requires the environment and the system itself to be controllable. In order to test a protocol stack, for example, one has to simulate both network behavior (such as packet loss, delay or duplication) and application behavior (such as bursty or peak traffic). However, parameters like processing load or network errors are not always easy to control.

Fourthly, protocol stack software serves as a *central point of failure* for all applications, whether running locally on the system or distributed over the network as clients. A software fault in an application can (or at least should) only harm the process executing that application without directly affecting other applications. Yet, a protocol stack error often results in the whole system going down, which may affect many local and remote users, especially in case of a server fault.

### Testing dimensions

Software testing can be viewed from different angles. This section presents a basic classification along four dimensions that are relevant to this dissertation: test objective, unit for testing, test strategy, and test methodology.

**Test objective.** The most common goal for software testing is to verify whether the unit under test complies with its functional specification. We will refer to this kind of testing as functional testing. Software testing can, however, address non-functional (behavioral) objectives as well. First of all, stress testing verifies whether the system handles various load conditions gracefully (i.e. without crashing). The system may handle high load conditions, for instance, by removing low priority requests, queuing requests it cannot handle immediately, using more appropriate (caching) strategies to change its behavior.

Another example of non-functional testing is performance testing. This type of testing verifies whether the system's throughput remains within pre-defined boundaries during various load conditions. These boundaries can relate, for instance, to response time, overall throughput or processing delay. The difference between stress and performance testing is that the former checks the system's correct behavior under stress conditions, while the latter is purely interested in performance results.

**Unit for testing.** Dependent on the adopted view of a system, different units for testing can be identified. What matters is not so much the size of the pieces tested, but the unit testing perspective [McG97a], i.e. whether or not a unit encapsulates functionality that is worth being tested. McGregor [McG97a] identifies such an entity (unit) as follows: "a component is worth testing when the penalty for the component not working is greater than the effort required to

test it". Not every unit will be sufficiently large, important or complex to meet this condition, thus not every unit will be tested independently. Typically, the basic unit for testing in object-oriented software is a class. Methods are simply meaningless separated from their class [Bin94b].

Zooming out from the basic unit level usually reveals several testing levels, each concentrating on a particular level of detail. One can identify class tests, sub-system (or class cluster) tests and system tests. Different terminology is used in the testing community to denote each of these test levels. Henceforth, we will refer to the lowest level of testing as *unit testing*, to the testing of sub-systems as *integration testing* and to testing of complete systems as *system testing*.

Test scenarios are described in so-called test cases (sometimes referred to as test designs). Hetzel [Het88] observes that "a document or statement that defines what we have selected to be tested and that describes the expected results is called a test design". Related test cases (for the same class, cluster or system) are typically grouped into so-called test suites.

**Test strategy.** Numerous test strategies have been described in the literature [Har00, Bin94b], varying from purely code-based approaches to formal ones, both for object-oriented [Bin99] and component-oriented systems [Gao00]. Of these strategies, the following three are well-known and often used in practice [Bin99].

*Black-box testing* tests a piece of software regardless of its underlying implementation. More specifically, it dictates that test cases for a software component be purely based on examination of its specification. The goal of black-box testing is to demonstrate that the software being tested does not conflict with its external specification.

*White-box testing* tests the implementation of a piece of software, irrespective of its specification. The goal of white-box testing is to identify such issues as infinite loops or unreachable code statements.

*Gray-box testing* is based on examination of both the specification for a piece of software and its underlying implementation. A typical goal of gray-box testing is to identify situations in which the software's behavior does not correspond to the expectations.

Most testing efforts are carefully planned combinations of black-, white- and gray-box testing techniques. Fine-grained testing, i.e. testing small pieces of software such as a single method or class, usually involves a significant amount of white-box testing. Coarse-grained tests for class clusters or sub-systems are typically black-box in nature. Moreover, component-oriented software development emphasizes the importance of black-box test approaches, since source code is rarely available to the component user [Szy98].

**Test methodology.** Traditional software development methodologies place testing at the end of the development cycle. An extreme case in this respect is the waterfall development model, which delays testing until after analysis, design and implementation [JBR99]. Current methodologies are obviously more iterative

in the sense that they propose a sequence of shorter analysis/design/implementation/test cycles. Yet, also iterative development approaches delay testing until the end of one (albeit smaller) cycle.

Common to traditional development approaches is that they all put the development process in the center of attention. Software developers must follow that process as closely as possible with specific emphasis on analysis, design, implementation and testing. The sequential nature of such development approaches and the rather strict separation between software analysis, design, implementation and testing, often produces software that does not (or only partially) fulfill the end user's original functionality and/or quality requirements.

Recent software development approaches are drastically different and place the developer and the product aimed-for in the center. This often results in a more lightweight and "developer-friendly" methodology, which explains why the community refers to it as *agile* [Amb02, NSN+97, Fow00]. In the eXtreme Programming (XP) approach [Bec99], for example, testing plays a central and crucial role in code development. It is part of the development process from the very start. Moreover, test code is written *before* the computational code and as such can be used as an up-to-date and executable specification for the system.

#### Research focus

During the past decades, several testing techniques have been developed. Originally, object-oriented programming was expected to reduce testing efforts because of increased reuse. Unfortunately, experience has shown this reduction to be an illusion [Bin94b]. To make matters worse, inheritance, polymorphism and encapsulation pose new problems for testing [Bin94b]. Migration from object-oriented towards component-oriented programming does not completely solve these issues [Gao00]. Despite all attention in software engineering research, testing is and still remains difficult, time-consuming and error-prone.

Although numerous research papers and books have been published concerning software testing [McG97b, BG98, HMF92, Bin99], an overall accepted technique that is easy and effective enough for programmers has yet to come. Three major reasons explain why such important software development aspect as testing is still not well-integrated in the software development process.

First of all, software is often designed without testing in mind, which results in software that is difficult to test. "Software testability" is a crucial concern to software development [Bin94a]. By consequence, the definition of a software component should not only identify a component as an executable and deployable entity, but also as a "testable" entity [Gao00]. There are several well-accepted object-oriented software engineering rules to develop extensible, adaptable and reusable software [LH89, Bin94a], and the same is true for component software [Gao00]. Four main criteria for developing testable components can be summarized as follows [LH89, Bin94a, Gao00]. First of all, low class coupling facilitates testing a

class in isolation. Secondly, delegating tasks transparently from one component to another (e.g. `componentX.doSomething()`) prevents a long method call chain provided by the caller (e.g. `componentX.getFoo().getBar().doSomething()`). This allows a component to change the sequence of sub-tasks to execute, without the need to adapt the caller. Thirdly, exposing short class interfaces facilitates testing, because less interface methods have to be tested and because long interfaces are more likely to change than short ones. Finally, interface consistency, i.e. all classes (or components) should present similar tasks through the same interface, allows to reuse test cases over different classes.

A second reason for testing being so difficult is that software engineering rules are *not enforced*. Even if it is known for years how to develop testable software, programmers often lack infrastructure support to guide them in the proper direction. Thus, if we are to develop more robust and error-free software in the future, one of the key challenges is to enforce these software engineering rules by a well-defined framework.

The third reason concerns the lack of *test infrastructure* that enables software programmers to write tests easily and efficiently [Har00]. What is needed, therefore, is a test framework to test the software that is implemented using the software development framework (which enforces the testability rules). The test framework allows reusing typical code to setup and tear down test cases and suites, and to execute them sequentially in a fail safe manner. In addition, the test framework should be reusable for both fine-grained unit tests and coarse-grained integration and system tests. Relevant examples of such test frameworks are PACT [MK96], Mock [TM00], and JUnit [BG98].

Given these problems in current software development and testing, our position regarding the second challenge (testing) can be summarized as follows:

> A major challenge is to provide a software architecture and component framework that supports the development of testable software. To this end, software components should be designed as testable units, and design principles that lead to the development of testable components should be enforced by the framework. In addition, developers should be offered a separate test framework, which allows efficient and effective functional testing of components, protocol layers, and protocol stacks as black-box entities. Therefore, the test framework should support an iterative style of software testing, starting from the first day of programming.

### 2.3.3 Framework optimization

This section positions the dissertation in the research domain of software optimization. The need for software optimization – and framework optimization in par-

ticular – is motivated first, after which three relevant dimensions are presented: the optimization target, the time of optimization, and optimization granularity. The section finally positions the dissertation's research focus along the three optimization dimensions.

#### Need for framework optimization

Performance, i.e. the number of processed requests over a period of time, is an important characteristic of protocol stack software. Controlling performance, however, is far from trivial since users and applications may pose non-trivial quality requirements affecting the underlying system. On top of that, these quality requirements often counteract performance by involving extra flexibility, which in turn often results in processing overhead. Finding an acceptable balance between user-level requirements and system performance forms a major challenge to protocol stack development, for several reasons.

A first reason lies in the contradiction between two important concerns in the development of protocol stack software: performance and run-time adaptability. On the one hand, protocol stacks must support complex and varying requirements of applications and users. On the other hand, performance becomes crucial because of increased usage, extended functionality (what is offered) and required quality of service (how it is offered). Although run-time adaptability can improve performance effectively, it often causes overhead. A first portion of overhead is introduced by the adaptive behavior itself. Adapting internal parallelism, for instance, requires monitoring to enable bottleneck detection in the first place, and to deploy a solution in the system accordingly. Needless to say, there is a performance price to pay. An extra portion of overhead may be introduced by the software design, which must take into consideration run-time adaptability. For instance, one approach to allow monitoring in the system in fine-grained places is to provide "hooks" where passing requests can be intercepted by a higher (meta-)level of control. In addition, this solution requires the code to be modularized in fine-grained units, concurrency to be represented as well-manageable and therefore separate entities, and the system to support plugging in and out monitoring code when needed.

Secondly, support for framework optimizations should be confined to specific and appropriate places in the system. Monitoring support, for example, is only needed in those parts of the system where bottlenecks are likely to occur. In addition, not all module instances of a particular type should necessarily be optimized. One instance, for example, may require monitoring while this is not needed for an other.

A third reason for performance control being so difficult is that optimizations should only be applied when needed at run-time. The highly dynamic operation context of a network (e.g. the Internet) may influence system performance temporarily. As a result, it is difficult to select and deploy appropriate optimizations

at build-time that will remain relevant during the system's life-cycle. In fact, optimizations should be reconsiderable at run-time. System monitoring, for instance, only becomes really appropriate when performance drops below a certain threshold. Instead of equipping the system with monitor support at design-time, it would be better to extend the system at run-time when needed.

### Optimization dimensions

Optimization techniques can be categorized based on optimization target (*what?*), time of optimization (*when?*) and optimization granularity (*where?*).

**Optimization target.** A wide range of software concerns can be optimized. The major target for optimization in protocol stack software is undoubtedly throughput, i.e. the number of finished requests per time unit, as it directly affects feedback delay to the user. A second important target for optimization is memory usage [MPBO96], especially in embedded systems with limited resources available. Throughput optimization concentrates on minimizing memory access, while memory optimization techniques focus on reducing the total amount of memory required.

Next to throughput and memory optimization, several other aspects are subject to optimization. A case in point are human-computer interfaces, especially for applications that are in direct contact with human users. Their look-and-feel must be suggesting what can be done and how, rather than requiring to study a user manual. A lot of research has been done in the context of typical PC applications, resulting in well-known paradigms and mechanisms like windows, buttons, drag-and-drop, true-type fonts (What-You-See-Is-What-You-Get), etc. The current challenge is to offer the same ease-of-use for hand held devices with small screens.

**Time of optimization.** Along the software development life-cycle, optimization can be scheduled somewhere between implementation and execution of the system. In general, one can identify three optimization moments.

First of all, framework optimizations may take place at compile- and/or link-time. Today nearly all compilers are able to optimize code for specific processors by using techniques such as loop unrolling, dead code elimination, function inlining, etc. Basically this boils down to optimizing the translation of source code into machine code for a specific hardware architecture. Apart from optimizing compilers, other optimization techniques include systems like Scout [MMOP95] and Click [KCK+00], which rely on "little languages" to build and optimize component configurations. Click optimization provides performance improvement by manipulating component source code. Knit [RFS+00] for its part is a general-purpose component definition and linking language for operating systems code that helps reduce performance overhead of "componentization". It can be used in combination with existing component systems such as Click. Research on separ-

ation of concerns and component models suggests that the optimization concern [SGC02] should be described separately in a specialized language and integrated in the system at compile- (or weave-)time. AspectJ [Asp03], for example, provides a language to describe non-functional concerns (aspects) along with a weaver tool that weaves aspects into functional Java code at compile-time.

Secondly, modern compiler technology allows extending the optimization process from compile-time to load-time, which is often referred to as just-in-time (JIT) compilation. In its pure sense, JIT compilation translates virtual machine code into hardware specific code in order to gain performance.

Finally, more intelligent optimizing systems monitor and optimize code during execution. This is particularly important for dynamically adaptable systems. Continuously optimizing systems [Suna, Sun03b, KF01] can re-optimize code, for example, when a new class is dynamically loaded into the system, which would result in code being no longer optimal. For this reason, they de-optimize the previously optimal code and replaces it with an optimized version that takes into account the new class.

**Optimization granularity.** The representation of the code to be optimized is coupled with the execution time and the optimization approach. The time of optimization decides whether the target is the system's architecture, plain source code or binary code. Optimization techniques are divided into white- and black-box approaches. White-box approaches optimize code "from inside", i.e. they typically focus on manipulating source code (e.g. loop unrolling, dead code elimination or function in-lining) in order to keep the processor pipeline filled, minimize memory footprint, etc. Black-box approaches rather optimize "from outside" and concentrate on higher-level entities such as classes, components or the system's software architecture. In such approaches, optimizations are often based on application, domain or environment information.

An example of a black-box optimization is restructuring the software architecture of a network router [Mat99, SHHP00], such that it only reassembles the incoming fragments intended for local delivery and not these that are forwarded. Such optimization is allowed only if the router connects networks with the same maximum fragment size. It only changes the software architecture, but not the functional code (as a white-box approach would do).

### Research focus

This dissertation focuses on framework optimization following a black-box approach and adaptable at run-time. Throughput does not only depend on individual objects or components, but also on how they interact. The latter type of optimization, which deals with overhead introduced by an architectural style and/or component framework, will be the center of attention. Our first goal is to develop a software architecture that allows "end-to-end optimization" [SSC98], i.e. optimization based on component relationships rather than on the compon-

ents themselves. In other words, optimizations focus on the way components communicate with each other. The second aim is to allow to undo a black-box optimization at run-time, and return to the original software architecture.

From a software engineering perspective, this requires maximally reducing the overhead that is introduced by an architectural style or a component framework. In addition, the reduction should concentrate only on those places in the architecture where framework support is not required at run-time. Finally, whenever removed framework support code becomes relevant again, the optimization must be undone. For example, all support for installing monitoring code should be removed at places where monitoring is not required. However, when such support becomes relevant again, it must be possible to reintroduce this code. This results in heterogeneous software systems with pieces of optimized as well as unoptimized code, i.e. a best of both worlds solution: run-time overhead only exists in places where run-time adaptability is required, yet this support is removed or optimized in places where it is not needed or used.

> Coping with contradictory goals of run-time adaptability and performance requires an elegant framework design and a well-defined software architecture to allow end-to-end framework optimization, using a black-box approach. In addition, optimizations must be localized in the framework (i.e. only at places where it is needed) and reversible at run-time.

## 2.4 Conclusion

The chapter has started by identifying the major problems in today's protocol stack software, concerning software architecture and design as well as the considerable requirements network services pose on the underlying protocol stack. The chapter then provided necessary background and discussed the relevant mile stones in software engineering research. In addition, it has proposed three key challenges in software development – self-adaptability, testing and framework optimization – and motivated why they help in solving some of the major problems in the development of protocol stacks.

Several existing systems focus on these challenges, but they often handle them individually. Many research results have been published concerning self-adaptability, testing or framework optimization, yet few of them take into account the overall view. This software engineering view should support and enforce design principles that enable and facilitate addressing these challenges.

The following chapters of this dissertation present a component framework with an associated software architecture that simultaneously supports self-adaptability, testing and framework optimization.

*"Make everything as simple as possible, but not simpler."*
*– From Albert Einstein.*

# Chapter 3

# The DiPS+ Component Framework

This chapter presents the basis of our research, the DiPS+ (Distrinet Protocol Stack+) component framework. Following a top-down approach, the chapter zooms in from a high-level architectural view, over the core design behind this architecture, to the DiPS+ prototype for building protocol stacks.

The DiPS+ architecture identifies two separate planes: one for data processing and the other for management. The data plane represents the core structure of a DiPS+ protocol stack, which identifies packets, components and protocol layers. The management plane is conceived as an independent layer on top of the data plane that enables monitoring the system's behavior and changing it in case of problems. The DiPS+ design proposes each of the core building blocks of the DiPS+ component framework, it shows how they cooperate and indicates their relevance in the overall picture. In addition, the design represents the management plane as an independent extension to the data plane.

The foundation for this work is the DiPS component framework for building protocol stacks [Mat99, MMWV01]. This chapter extends the architecture and changes the design in crucial places. These changes lead to DiPS+, which supports not only software programming, but the overall software development process, especially with respect to adaptability and control, testing, and framework optimization. To avoid any confusion between the original DiPS model and the one presented in this dissertation, we refer to the latter as DiPS+.

The chapter starts by identifying the main requirements the DiPS+ framework should fulfill and motivating why the original DiPS design and architecture are not sufficient. Section 3.2 presents the software architecture that is used as the foundation of the DiPS+ component framework. Sections 3.3 and 3.4 zoom in on each

plane and explain how DiPS+ satisfies its prerequisites. After this presentation of the core architecture and design, Section 3.5 provides the background necessary to understand how these ideas have been applied in the DiPS+ prototype. It shows how a TCP/IP protocol stack architecture is described, and how such description can be transformed automatically into a running protocol stack. The chapter ends by summarizing the main contributions and formulating conclusions in Section 3.6.

## 3.1 From DiPS to DiPS+

### 3.1.1 Objectives

In essence, the goal of DiPS+ is to provide an answer to some major shortcomings and issues protocol stack software development is facing. To this end, this section identifies the main objectives the DiPS+ platform must tackle to support and facilitate the three major tracks of the dissertation: adaptability, testing and software optimization.

**Uniformity.** Systems should be composable of fine-grained entities, which may be grouped to treat them as a whole. Grouping entities enables focusing on the appropriate level of detail. Yet, different granularities should be usable in a consistent and uniform manner. In other words, it should be possible to manage, test, or employ entities irrespective of their granularity.

**Clear mapping between architecture and design.** A clear mapping between architecture and design makes the system easier to understand and manage. All concerns, both functional and non-functional, must be represented as well-identifiable modules. This allows developers to concentrate on one specific concern at a time.

**Controllability.** Two important concerns, concurrency and monitoring the data flow, should be controllable as separate entities. Concurrency control involves searching for an optimal balance between local parallelism and processing load in order to maximize global throughput. The data flow for its part may be controlled by intercepting messages and interpreting their characteristics, which in turn allows to fine-tune the system's behavior accordingly. This requires both a management sub-system that decides what changes must be deployed, and a system architecture and design that is open to adaptation. In addition, components should be completely unaware of the management sub-system to allow them to be reused in combination with different types of control.

### 3.1.2 Shortcomings in DiPS

In order to reach these objectives, the DiPS framework design and architecture should be changed or extended in three well-defined areas. First of all, the design

of the core DiPS abstractions should be *unified* to enable generalization. This is crucial to allow a test framework to offer a generic and uniform test approach for unit, integration and system testing. Secondly, the entry and exit of a DiPS component should not only be *explicit in the design but also in the architecture and the running code.* The entry and exit of a component are the ideal place to intercept information that is flowing through the system. In addition, intercepting information becomes completely transparent for the functionality inside the components. Finally, the DiPS architecture must be extended with a *management and control plane* that is responsible for monitoring the underlying protocol stack and adapting it when needed. Since all DiPS+ components share a uniform design, they all offer the same type of entry and exit point. By consequence, the management plane can be attached to any component in the system. Furthermore, the management plane is transparent for the components in the protocol stack, since it can be attached to explicit entry and exit points in DiPS+.

## 3.2 The DiPS+ architecture

Taken as a whole, the DiPS+ architecture is layered: it represents data processing and management as two planes on top of each other. Each of these planes is in turn represented by a software architecture that combines multiple architectural styles such as the pipe-and-filter, the blackboard and, again, the layered style [GS93]. In addition, the combination of the pipe-and-filter and the blackboard style of communication enables concurrency control, i.e. customizing how request handlers are spread throughout the system in order to optimize the overall throughput.

### 3.2.1 Data and management plane

Figure 3.1 illustrates intuitively the DiPS+ architecture, which identifies two clearly separated planes. The data plane houses the functional part of the system, i.c. the protocol stack. Without going into detail, this plane identifies components and how they are connected on the one hand, and offers layers as a composition of basic components on the other hand. Section 3.2.2 describes the data plane architecture in detail.

On top of this data plane, DiPS+ offers the management plane, which acts as a meta-level to extract information from the data plane and control its behavior. Two main tasks have been identified at this level: packet filtering and concurrency control. Packet filtering retrieves and collects information from the data plane by intercepting component communication. Concurrency control for its part involves exploiting processing resources optimally, and customizing their scheduling. This enables an application or a system administrator to prioritize the processing of particular tasks or requests in order to apply optimally the available processing

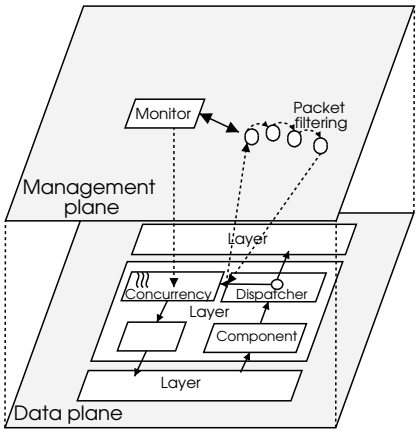



Figure 3.1: The data and the management plane in the DiPS+ architecture.

resources. Section 3.2.3 describes how concurrency control is supported in the data plane architecture.

Although the dissertation clearly limits the scope to packet filtering and concurrency control, many other management concerns can be identified. Replacing components at run-time, for instance, implies major management activities [JMMV02, KM90]. First of all, data flowing through the system must be blocked while the component is being replaced. Secondly, one must be sure that the component to be replaced is not processing data, and is not involved in any transaction during replacement. For example, a packet reassembler component queues incoming fragments until all of them are available to reassemble the packet. Consequently, without extra control during component replacement, the fragments that are queued in the reassembler get lost. Depending on the provided reliability level of the system (guaranteed packet delivery, or not), losing packets may be unacceptable. Therefore, the management plane has to control how many packets enter and leave a component. With this information, it can calculate whether or not a component can be replaced safely. A third management concern is to drain a component to be replaced, i.e. block fragments of new packets and let other fragments through until all pending packets are reassembled. Finally, to omit the draining overhead, the management plane could decide to transfer the internal state (e.g. the fragment queue) from the replaced component to the one that is inserted.

### 3.2.2 Inside the data plane

When we take a closer look at the architecture of the data plane we can identify three main architectural styles – the pipe-and-filter, the blackboard, and the

layered style.

### Pipe-and-filter style

The DiPS+ data plane architecture can be compared with the assembly line in a factory. Such a system consists of a collection of independent entities (robots), each of which is responsible for a well-defined sub-task (e.g. installing the dashboard, a seat or the engine). These entities are interconnected via a conveyor belt, which transports "products" from one robot to the next one. Each robot has a different processing-time, dependent on the complexity of its task. In case of a car factory, for instance, painting a car is obviously more time-consuming than installing a seat. Processing intensive entities slow down the conveyor belt to the tempo of the slowest entity attached. To overcome this problem, complex tasks are executed in parallel as much as possible and products that cannot be processed immediately are buffered until one of the robots is ready.

The pipe-and-filter style is very convenient for developing various types of system software such as protocol stacks, file systems and web services. A protocol stack can be thought of as a down going and an up going packet flow. The downward flow of out going packets identifies such components as header constructors, routing, packet fragmenters and serialization. The incoming packet flow involves processing steps such as de-serialization, packet forwarding and reassembly, and header parsing. File systems for their part identify a similar flow of request parsing, cache lookups, device driver request construction, etc. [WCB01]. Web services are also typically flow-oriented [Sun03a].

The core abstractions of a pipe-and-filter software architecture are components (or filters) and connectors (pipes). Our view, however, distinguishes additional connection abstractions for dispatching, concurrency and event handling. These are not only very relevant abstractions for protocol stack software, identifying them as separate entities also facilitates their control. Figure 3.2 shows an example of a simple DiPS+ architecture.

**Component.** Each DiPS+ component represents a functional entity with a well-defined and fine-grained task. Each component processes network packets, for instance by constructing or parsing a network header, fragmenting a packet or reassembling its fragments, or encrypting or decrypting packet data. Components are independent entities, unaware of the other components in the system. As such, DiPS+ components are comparable to the robots or machines in the factory metaphor.

**Connector.** Connectors serve as architectural pipes, i.e. they provide a means to glue components together into a flow. The connector concept maps to the conveyor belt of the assembly line, as it transports packets from one component to another.

**Dispatching.** The dispatcher serves as a demultiplexer, allowing to split a single flow into two or more sub-flows. Sub-flows are used to differentiate the

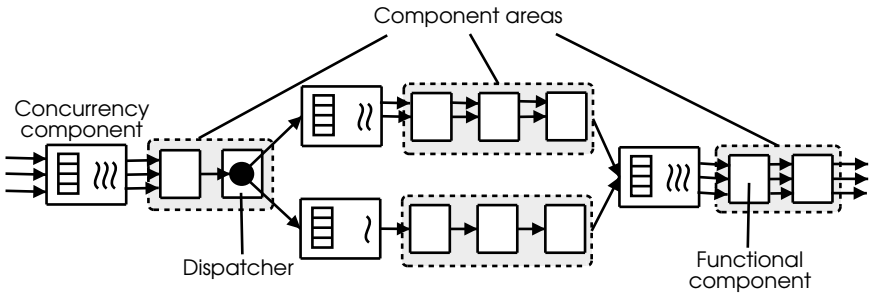



Figure 3.2: Example of a DiPS+ component pipeline with a dispatcher that splits the pipeline in two parallel component areas. More processing resources have been assigned to the upper concurrency component.

treatment of incoming messages. For example, the incoming packet flow in the network layer of a host with routing capabilities is typically split into one flow for local delivery and one for forwarding packets. In addition, packet dispatching (or demultiplexing) usually takes place at each protocol layer-crossing in the up going path. Between the network and transport layer, for instance, the up going packet flow of a typical UDP-TCP/IP protocol stack is demultiplexed into a TCP [ISI81b] and a UDP [Pos80] sub-flow. Again, there is a comparable concept in the factory metaphor, namely the branches of the assembly line that allow to differentiate between products (e.g. cars) with different sets of optional features to be installed.

**Events.** The DiPS+ architecture combines the pipe-and-filter based style with an event-based interaction mechanism. This allows components to send out control signals to anywhere in the system. The DiPS+ architecture offers the event connector, which allows components to register interest in a certain event. To some extent, this is comparable to an alarm signal in the assembly line, which triggers all interested workers when someone pushes an alarm button.

### Blackboard style

Referring to the factory metaphor, we can identify another important architectural style. Each product is typically labeled with meta-information (e.g. a bar code) that specifies how it should be treated along its way through the factory. Robots can interpret and use this information when processing a product. Moreover, they can attach additional meta-information to inform other robots “down the stream”. Obviously, robots should not remove meta-information, since they have no clue which other robots might need it. This kind of management task is reserved to specific factory administrators that have an overall view on the factory.

In terms of software architecture, this style of communication corresponds to

the blackboard style. This communication style is characterized by indirect message exchange, which results in an anonymous component interaction model.

The blackboard interaction style is very convenient in combination with the pipe-and-filter style. Since each message visits all relevant components following the pipe-and-filter style, messages are the ideal means to serve as central data source. We now zoom in on the blackboard model and explain how it is mapped onto DiPS+.

**Message.** In order to finish a common task, components forward a message from the source to the sink of the component flow. A message is the analogue of a product that is conveyed on the factory's assembly line. In case of protocol stacks, a message represents a network packet. Messages can be annotated with meta-information in order to influence how they are processed. Such meta-information can be provided by an application or by components inside the system. The former is relevant, for instance, when application-level parameters (e.g. user priority) may impact the internal behavior of the protocol stack. The major benefit of having components attach meta-information to incoming messages is anonymous component interaction.

Two examples will clarify the benefit and power of attaching application-specific preferences as meta-information to a message. First of all, it allows application-, or even user-based prioritization of messages. The attached priority may be used for selecting a message to process from a message queue. For prioritizing incoming messages, a special "prioritizer" component could be introduced in the up going component pipe of the protocol stack. Another, more challenging case is to use application-specific communication preferences to influence how network packets are transferred over the physical network. Imagine a vehicle with telematica capabilities installed, and with multiple wireless communication channels available [KMD+99]. Such communication channels each may use a specific wireless communication protocol with specific characteristics concerning communication cost, speed, or reliability. Each of these characteristics may be highly relevant to the user. In such case, data transfer may be influenced by attaching user-based meta-information to out going packets. Based on this meta-information, the most appropriate wireless communication device can be selected at the bottom of the protocol stack. Without meta-information, it would be very difficult to make such application-specific decision at the bottom of the stack.

**Anonymous communication.** Anonymous component interaction enables composing DiPS+ components without affecting their source code. This independency allows for reuse and adaptation of individual units in different protocol stack configurations.

The idea behind the blackboard architectural style is that a collection of independent entities (producers and consumers of information) interacts with a common data structure (the so-called blackboard), which represents the current state.

The blackboard model is usually represented by three major parts [GS93]. First of all, the *knowledge sources* represent application-specific knowledge and computation. They can interact directly with the blackboard, but never with each other. Secondly, the *blackboard data structure* itself encapsulates state data and uncouples knowledge sources from each other. Knowledge sources make changes to state information in the blackboard, thus incrementally producing a solution to the problem at stake. The third part of the model is the *controller*, driven entirely by the state of the blackboard. Control may be located in the knowledge sources, the blackboard, a separate entity or a combination of these.

The blackboard model is mapped onto the DiPS+ architecture as follows. The knowledge sources correspond to the DiPS+ components. Each component can manipulate or attach information to an incoming message, and this information can be anonymously interpreted by any other component in the stack. Obviously, components have to agree on the type of information that is published via the blackboard. Yet, they are unaware of exactly which component has produced that information. Secondly, the blackboard data structure is represented by the meta-information that is attached to the message traversing the protocol stack from one DiPS+ component to another. In this way, there is a separate blackboard for each message and it traverses the protocol stack along with its message. Finally, control of the blackboard is entirely distributed across the knowledge sources. Each DiPS+ component controls if and when meta-information is retrieved from an incoming message. Similarly, each component may decide to add or change meta-information.

#### Layered style

Introducing a separate layer abstraction is highly relevant for several reasons. First and foremost, it is very natural to have a design entity that directly represents a key element of a protocol stack. Often, a protocol layer is implemented, integrated or reused in its entirety instead of as a group of independent components. This does not obsolete the relevance of fine-grained units, rather it augments the level of support towards protocol (stack) programmers. Similarly, a factory's assembly line is not structured as a straight connection of processing entities. Rather, these entities are grouped into coarse-grained logical sub-systems that serve a higher-level goal. Such sub-systems may be, in case of a car factory, metal engineering (pressing or welding), interior processing or body-work painting. The hierarchy of fine-grained entities and more coarse-grained sub-systems allows to zoom in and out to the appropriate level of detail.

Secondly, each layer offers an encapsulation boundary. Every protocol layer encapsulates data received from an upper layer by putting a header in front of it. Each layer corresponds to one header that contains the information to be shared with the peer layer at the remote host. For incoming packets, each layer parses the protocol header and attaches relevant information to the packet as

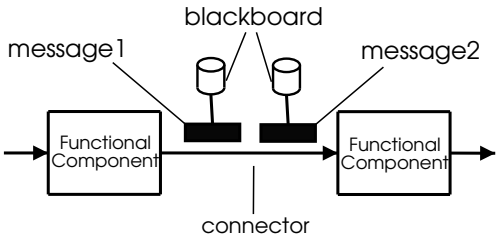



Figure 3.3: Anonymous communication via a blackboard architectural style: a blackboard data structure has been coupled to each message to carry meta-information from one component to another.

meta-information.

Finally, from a protocol stack point of view, layers provide a unit of dispatching. Layers can be stacked in many ways. The simplest way – one layer on top of another – does not require layer dispatching. Yet, in cases of multiple layers on top of one (N on 1), one layer on top of multiple layers (1 on N), or a combination of multiple layers on top of multiple other layers, dispatching is definitely required. Once more, this confirms the relevance of offering layers as a separate abstraction.

### 3.2.3 Concurrency control

Processing resources (i.e. threads) are spread over the DiPS+ architecture by introducing concurrency components in specific places. Concurrency components sub-divide the system into independent component groups, which will be referred to as component areas[1] (as illustrated in Figure 3.2).

The concurrency component is responsible for two tasks: injecting active behavior into the system and request scheduling. Active behavior means that one or more threads each guide a request through the component area. On a mono-processor, these threads are interleaved by the scheduler of the operating system, thus resulting in virtual parallelism. Yet on multi-processors,[2] each concurrency component may be associated with a different processor, which enables true parallelism.

Request scheduling in a concurrency component involves the selection of the next packet to process. The selection criterion may be based on request-specific information such as the size of the request, its source or destination, or its type (e.g. connection setup versus data transfer).

[1] Welsh [Wel02] uses the term “stage” to denote a similar abstraction.

[2] Multi-processor systems or mono-processors that execute multiple parallel threads on a single chip. Examples of such on-chip parallelism are the Intel®Pentium®4 processor with hyper-threading technology [Int03], and Sun’s UltraSPARC “throughput computing” strategy [Sun03c].

When mapped to the assembly line metaphor, the concurrency component may be compared to the combination of a storage space and one or more workers. The storage space buffers incoming products that cannot be processed immediately. The workers correspond to the processing resources.

There are two major advantages of having explicit concurrency components. First of all, it allows for fine-grained and distributed control of scheduling requests. Each concurrency component may incorporate a customized scheduling strategy, using all meta-information that is attached to the request by upstream components. This information may not be available at the beginning of the component pipeline. In case of protocol stacks, for instance, an incoming network datagram only reveals its encapsulated information by parsing protocol headers. The more headers are parsed, the more information is available, for instance to prioritize between multiple source addresses.

A second advantage of having concurrency components spread throughout the system, is that it allows to prioritize not only between incoming packets, but also between component areas. DiPS+ concurrency components allow, for instance, to associate additional threads with those component areas that incorporate crucial sub-tasks. The component pipeline as depicted in Figure 3.2 identifies two parallel component areas. This setup (i.e. two parallel tracks) may benefit the most from spreading concurrency components since it demultiplexes the packet flow and, by consequence, allows to differentiate between the possible sub-flows.

Suppose the setup of Figure 3.2 represents one area that encrypts the data in each request while the other area only encrypts the header information, which is clearly less processing-intensive. In this case, one relevant approach to optimize the overall system's throughput would be to add processing power to the data encryption area, since this is the most processing intensive. However, the opposite approach might also be relevant, when the majority of packets follows the header-encryption path. In this case, the average system throughput can be increased by prioritizing the header-encryption area, and thus by adding threads to the concurrency component associated with it.

Concurrency components exploit the benefits of both the pipe-and-filter and the blackboard architectural style. The pipe-and-filter style divides the system into independent components, each with their specific task. The DiPS+ dispatcher allows to split the component pipeline into parallel sub-pipes, which enables differentiating their processing. Thanks to the blackboard style of packet forwarding, component tasks are typically packet-based, i.e. each component handles incoming packets by processing packet-specific data and/or meta-information. This allows to increase parallelism since most components have no local state that is shared by multiple threads in parallel.

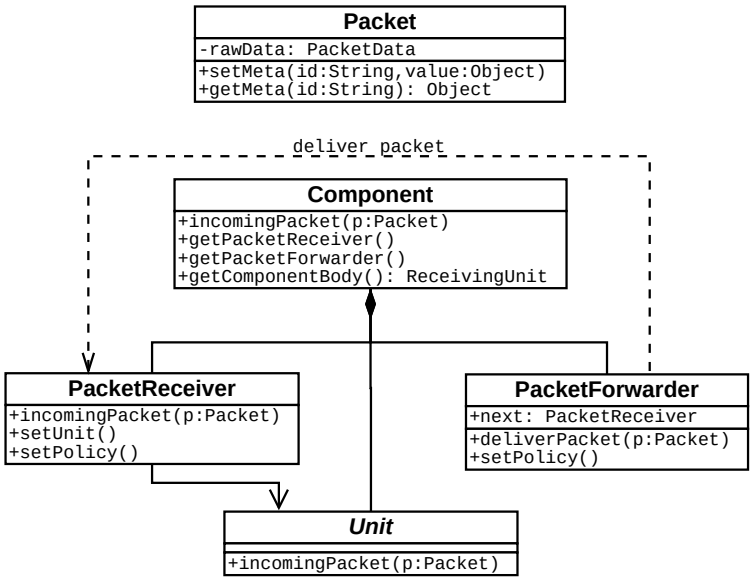



Figure 3.4: The design of two core DiPS+ abstractions: component and packet.

## 3.3 Design of the data plane

Zooming in to a more detailed level, this section proposes the software design behind the DiPS+ architecture. The design is divided into two orthogonal planes. The data plane represents all functional entities such as components and layers, whereas the management plane embodies management aspects.

We present the basic building blocks of the DiPS+ design: `Packet`, `Unit`, unit wrapping by a `PacketReceiver` and `PacketForwarder`, and `Component`. The design is shown in Figure 3.4. The three unit types offered in DiPS+ are also presented in detail and illustrated in Figure 3.6.

### 3.3.1 Packet and anonymous communication

A DiPS+ packet object represents a network packet that traverses the protocol stack. However, it is not just an encapsulation of data and protocol headers, but a key framework abstraction that supports anonymous communication in the protocol stack. It provides a way to store and retrieve *meta-information* without any further control of the specific type of information stored. By consequence, components must agree on the type of the meta-information that is shared through the blackboard. Yet, in this way, however, new types of meta-information can be supported easily since no type checking is done.

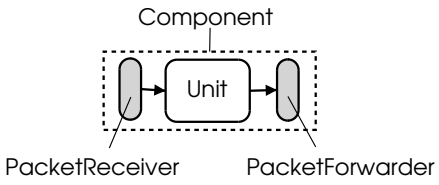



Figure 3.5: A DiPS+ component with a packet receiver, a unit, and a packet forwarder.

### 3.3.2 Component entry and exit point

#### DiPS+ component

Component activity is split into three concerns: packet acceptance, packet processing, and packet delivery. The DiPS+ framework controls packet acceptance and delivery by means of explicit component entry and exit points (the packet receiver and forwarder). Packet processing is taken care of by a DiPS+ `Unit` class, which forms the core of a component. The packet receiver and forwarder uncouple processing units. The DiPS+ `Component` class is purely a framework entity that is transparent to programmers. A component encapsulates and connects a unit together with its packet receiver and forwarder. Figure 3.5 illustrates intuitively the structure of a DiPS+ component.

#### Packet receiver and forwarder

There are two reasons why DiPS+ offers the `PacketForwarder` (PF) and `PacketReceiver` (PR) class as separate entities. First of all, the PF and PR together represent an architectural connector. As such, they uncouple DiPS+ units from each other. A PR object represents the entry of a unit, while the PF represents the exit. Eventually, the PF delegates packets to the PR associated with the next unit in the path.

The second reason for having an explicit PR and PF has to do with enabling fine-grained and unit-specific management and control. Both the PR and the PF serve as packet interception points to the management meta-layer. In order to do so, they are equipped with a `Policy` object that allows to inspect and manipulate the packet flow. More details about linking the data and management plane via a unit's PR and PF will be described in Section 3.4.

### 3.3.3 Unit types

The `Unit` class is the basic packet processing building block in DiPS+. It is the heart of an architectural component and must be either provided by a programmer or reused from a unit repository. Each unit offers a narrow and fixed interface to

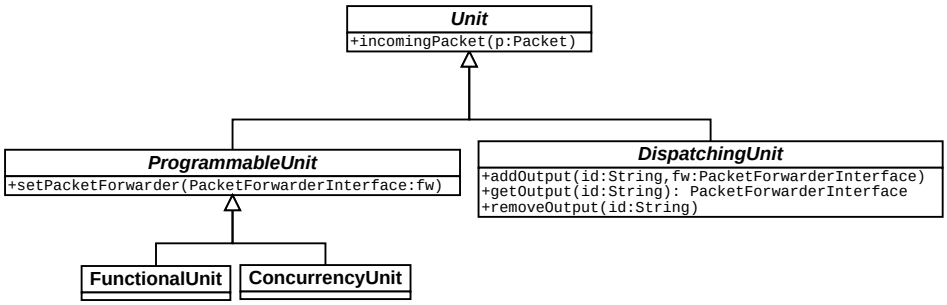



Figure 3.6: The design of the unit class hierarchy in DiPS+.

receive incoming packet objects: `incomingPacket(Packet p)`. This interface is the same for all DiPS+ units, resulting in high reusability.

Packet processing can be viewed from different angles, each with different goals and characteristics. First, from a functional perspective, packet processing entails applying some specific activity that can be identified in the architectural component pipeline. Secondly, from a concurrency perspective, packet processing comes down to selecting a packet from a queue (using some scheduling strategy) and associating with that packet a processing resource that guides the packet through a component area. Finally, from a dispatching perspective, packet processing selects an appropriate delivery point based on information that is available in the packet.

These three concerns can be generalized, since units serve a very generic and consistent task: accepting incoming packets and, after processing, delivering them to the next component. To this end, units offer a generic and uniform unit interface. This allows to zoom out and treat all units in a uniform manner. For example, many aspects of testing, throughput monitoring or optimization can be generalized for all units.

In order to stress the existence of different concerns, however, each packet processing perspective is reflected in the design by a different unit type. Programmers must understand and follow this unit hierarchy in order to enable concern-specific control at build- or at run-time. Yet, experience learns that this is a minor obstacle, especially compared to the learning curve for white-box frameworks in general. Figure 3.6 shows the `Unit` class tree in DiPS+ with functional, concurrency and dispatching units. Dispatching units typically have more than one packet forwarder, in contrast to programmable units. Programmable units are further sub-divided into functional and concurrency units, to stress their different behavior in the design.

**Functionality.** The code in a `FunctionalUnit` class describes a well-defined piece of functionality, for example, header construction or parsing, data encryption or decryption, packet fragmentation or reassembly, etc. This code is typically

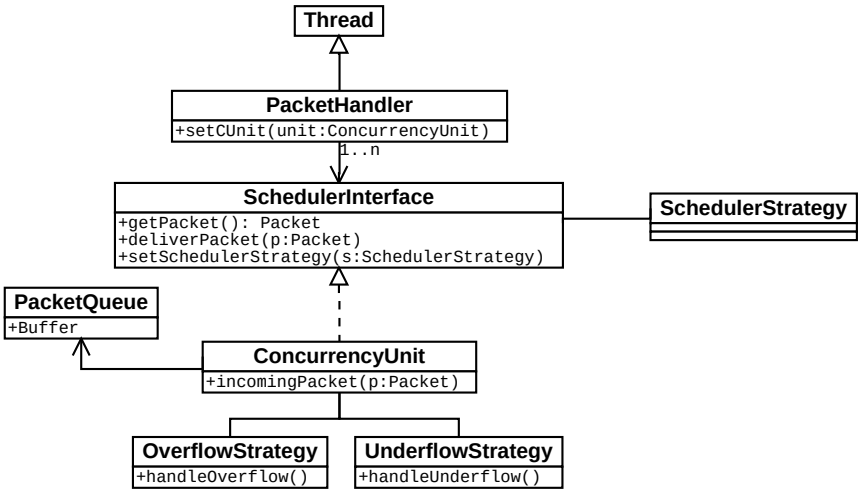



Figure 3.7: The design of the DiPS+ concurrency unit.

written by a protocol (stack) programmer.

**Concurrency.** A concurrency unit allows to increase or decrease the level of parallelism in the component area behind it. In addition, it controls when and which requests are scheduled. These tasks are offered by four major abstractions: `PacketQueue`, strategies for overflow or underflow of the packet queue, `PacketHandlers`, and the `SchedulerStrategy`. Figure 3.7 displays the overall picture of the concurrency unit design.

The `PacketQueue` object stores incoming packets that cannot be processed immediately. Its capacity can be increased (to cope with request bursts) or decreased (to release unused memory). Concurrency behavior to deal with overload and underload can be manipulated by registering specific strategies (`OverflowStrategy` and `UnderflowStrategy`). This offers an open design, i.e. it ensures control of the concurrency unit without exposing its internal attributes (such as the packet queue).

The `PacketHandler` is a thread that guides a packet through the component area behind its concurrency unit. To minimize thread creation overhead, packet handlers can be fetched from a thread pool and are reused to execute on different concurrency units. More than one packet handler may be associated with a single concurrency unit to increase the level of parallelism in a particular component area. The packet handler communicates with its associated unit through the scheduler interface (see Figure 3.7).

A `SchedulerStrategy` class is used to decide which packet will be selected next from the packet queue. This allows to differentiate between incoming packets, based on the packet-specific information (e.g. the packet's source or destination,

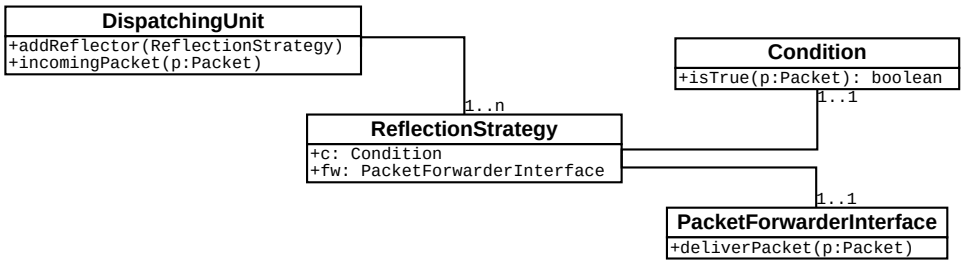



Figure 3.8: The design of the DiPS+ dispatching unit.

data size, type (data or control packet), etc). Again, the scheduler strategy may be customized to fine-tune the concurrent behavior of the concurrency unit.

**Dispatching.** Packet dispatching is offered by a separate DiPS+ abstraction: the `DispatchingUnit`. A dispatching unit delegates incoming packets to the appropriate destination, based on meta-information retrieved from them.

The design of the DiPS+ dispatching unit is illustrated in Figure 3.8. A reflection strategy is used to find the appropriate packet forwarder for delegation. Each strategy encapsulates a `Condition` object that tests whether or not an incoming packet matches its criteria. To do so, the condition object can interpret any meta-information attached to DiPS+ packets. If a packet matches the condition, it is delegated to the packet forwarder associated with the strategy.

### 3.3.4 Layers

#### Design

Five main abstractions are relevant in the layer design: entry and exit point identification, layer dispatching, layer types, layer connection, and encapsulation.

**Entry and exit points.** Since each protocol layer identifies a down going and an up going packet flow, it offers at least two entry points and two exit points. This is shown in Figure 3.9. An entry point accepts incoming packets from an upper or lower layer in the protocol stack. It represents the packet receiver of the component that accepts incoming packets associated with a flow. Similarly, an exit point represents the packet forwarder of the component that delivers out going packets to an upper or lower layer.

Entry points are made accessible via the `getUpperEntrypoint()` and `getLowerEntrypoint()` methods in the layer interface. These are shortcuts to the more generic method `getEntry(entryId)`, where *entryId* is a string that identifies an entry point. Similarly, exit points are accessible via the `getDownDelivery()` and `getUpDelivery()` methods, or via the more generic `getExit(exitId)`.

**Layer dispatching.** Protocol stacks can be configured in many ways, which

clearly influences layer dispatching. The simplest configuration is a tower that stacks each layer on top of exactly one other. In this case all packets that leave a layer via its exit point can directly be forwarded to the entry point of the next layer in the packet flow. In reality, however, configurations are typically more complex and multiple protocols can co-exist at the same level of the protocol stack. At the transport level, for instance, TCP and UDP typically exist next to each other, which requires dispatching packets that arrive from the IP layer below. At the datalink level, a network router with multiple network interfaces must dispatch down going packets to the appropriate network interface driver. For such configurations, layer dispatching is required to select which flow a packet must follow.

To this end, a layer can be extended with a DiPS+ dispatcher. The advantage of offering dispatching separately is that a dispatcher is only introduced when it is needed. Figure 3.9 shows a protocol layer with a dispatcher in both the down going and the up going flow.

**Layer types.** Apart from "normal" protocol layers (e.g. TCP, IP, Ethernet), we identify two special layer types that serve as the boundary between a DiPS+ protocol stack and the environment. The environment on top is formed by the applications using the stack. For example, the application boundary at the top of a protocol stack is typically represented by a socket. In DiPS+ a socket is represented by the `UpperLayer` class. On the one hand, an upper layer provides an interface to the applications above, which allows typical application behavior such as opening and closing a network socket, and sending or receiving data. On the other hand, it provides a DiPS+ layer interface at the bottom, allowing to forward and receive DiPS+ packets.

The environment underneath a DiPS+ protocol stack is typically the host's operating system, represented as network device drivers or non-DiPS+ protocols offered by the host itself. Similar to an upper boundary layer, this lower boundary is designed as a `LowerLayer` class. A lower layer offers a DiPS+ layer interface at the top that allows sending and receiving DiPS+ packets, along with an environment-specific interface at the bottom. The hardware interface layer, for instance, couples the datalink protocol layer in DiPS+ with the network device driver of the operating system.

**Connecting layers.** Like for components, there is a similar connection mechanism for layers represented by the `LayerGlue` class. A layer glue connects layers by coupling the exit point of one layer with the entry of another, and installing the selection condition in the layer dispatcher.

To this end, the layer glue defines two identification keys (`UpperLayerKey` and `LowerLayerKey`) respectively for the upper and the lower layer it connects. The idea is that a component attaches as meta-information a layer key, which indicates the path to follow from one layer to another. For each incoming packet the layer dispatcher matches this key with the key associated with a layer glue.

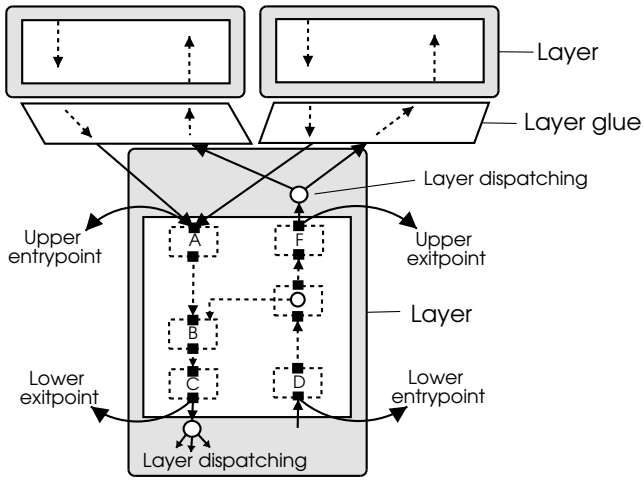



Figure 3.9: The design of a DiPS+ layer with identification of upper/lower entry and exit points, and layer dispatchers.

For example, a typical field found in the protocol header of an incoming packet is the identification of the encapsulated protocol. This identification is standardized, but not necessarily equal for each protocol combination. Indeed, the Ethernet specification [DIX80] identifies the Internet Protocol (IP) as 0x0800, whereas the Point-to-Point protocol (PPP) [Sim94] refers to it as 0x002F. For that reason, the protocol header parser component translates this protocol identifier to an upper layer key object and attaches this as meta-information to each incoming packet.

**Encapsulation.** Besides connecting layers, the layer glue also tags each down going packet with the protocol layer it comes from. This is done by attaching the upper layer key of the layer glue as meta-information to each down going packet.

Protocol tagging is necessary because it enables a packet to be dispatched correctly between layers when traversing the protocol stack at the packet's receiver. In other words, it provides the necessary information that allows traversing the same sequence of protocols both at the sender and the receiver but in opposite direction. To this end, each protocol header typically identifies the next protocol that is encapsulated in the packet, in case multiple alternatives are possible. By consequence, the encapsulated protocol identifier must be available at header construction-time, which is the case since it has been attached to the packet in the layer glue.

It is a considerable advantage to delegate to the DiPS+ framework the task of providing encapsulation information, and this for two main reasons. First of all, if the framework were not responsible for this task, it would need to be programmed explicitly in the components. This prevents layer reuse in multiple configurations,

because the encapsulation identifier depends on both the upper and the lower layer (cfr. the Ethernet protocol in combination with PPP or IP). By consequence, encapsulation in the framework prevents that components in a layer must know about the upper layer protocol.

A second reason for migrating encapsulation to the framework is that it is a common task that is relevant for most protocol stack configurations. Indeed, whenever multiple layers are on top of another, a selection must be made at the boundary (i.e. the layer glue) between the upper and lower layer. This makes it no more than logical to generalize this task at framework level.

#### Layer resources

A layer resource encapsulates information that is relevant to more than one component inside a specific layer. Think for instance of a central data repository that stores information about the status of a connection-oriented transport protocol, a routing table in a network layer, or an address cache in an address resolving protocol (e.g. the Address Resolution Protocol (ARP) [Plu82]). Moreover, a layer resource represents information that can be updated and controlled by an external party. A system administrator, for instance, may add or remove entries from the routing table, or reset the cache. Adaptations may as well originate from the protocol stack environment itself. For example, configuring a network interface card with a network address causes a trigger from the protocol stack that results in adding an entry to the routing table resource.

A resource *manager* provides the link between the protocol stack or a management application on the one hand, and a specific type of layer resource on the other hand. This link may be realized when a resource manager registers itself to receive protocol stack related events – for instance, the configuration of a network interface. Yet, applications may as well directly call a resource manager, which can be retrieved via the `PStack` object.

Having layer resources available as separate entities has two major advantages. It focuses attention for concurrency control on well-identifiable entities in the system. Layer resources must be thread safe, since multiple components and/or applications may access them in parallel. Extracting layer resources from components facilitates access control and, by consequence, makes the code more robust.

The second advantage is that it improves configuration flexibility, as components can be configured with various resource implementations and vice versa. Resources can be reused for different component implementations and different resource implementations can be associated with a single component.

## 3.4 Design of the management plane

Two important concerns for management and control are addressed in DiPS+. Packet interception, the first concern, allows interpreting the packet flow through the component pipe, for example by counting how many packets arrive per time unit, or by measuring the average packet size. The second concern is concurrency management, which aims at balancing processing resources – threads – throughout the system in order to maximize global throughput.

An extra objective has to do with "pluggability" of the management plane. This implies that the management plane can be transparently added to or removed from DiPS+ units at the data plane.

### 3.4.1 Intercepting the packet flow

The following three examples emphasize the need for fine-grained and dynamic management. First of all, throughput measuring involves calculation of arrival and departure rates at specific places in the system and of the ratio between them. Bottlenecks are these places in the system where the arrival rate of packets is much higher than the departure rate. Since each concurrency unit identifies the start of an independent sub-flow, they are the ideal place for bottleneck detection. Yet, this requires run-time monitoring right before and after each concurrency unit.

Secondly, replacing components in a multi-threaded environment requires blocking packets. To be efficient, packets should be blocked as close to the replaced component as possible to minimize the impact on performance. Indeed, in this way only those threads are blocked that actually visit the component to be replaced, while threads in independent component areas are excluded. Again, fine-grained control of the packet flow is absolutely necessary.

Finally, the need for fine-grained control and management is also applicable to other domains than protocol stacks. I/O (Input/Output) intensive applications that work on large data sets usually have well-identifiable I/O patterns during their execution. A typical pattern involves, for instance, a read phase to fetch data from persistent storage, a processing phase to compute new data and finally a write phase to store these new data. Each of these phases has specific characteristics that may influence caching strategies. Read-ahead caching is appropriate for handling large data sets during the read phase. During the write phase, however, when reads are more scattered, caching is better turned off [MER96]. In order to detect specific I/O patterns and manipulate caching strategies accordingly, fine-grained monitoring is needed.

In order to enable fine-grained management, the system must be open to administrators or administrative tools in a well-defined way. Exposing specific implementation issues in such a way that important strategies can be customized to system and application preferences is a well-known technique known as open im-

```
public interface Policy {
    void policyIncomingPacket(Packet packet);
    void setStartModule(ManagementModule m);
    void setEndModule(ManagementModule m);
}

public interface ManagementModule {
    void setNextModule(ManagementModule m);
    void deliverPacketToNextModule(Packet packet);
}
```

Listing 3.1: The interfaces for a policy and its associated management modules.

plementation[3] [KLL+97]. In addition, control must be customizable at run-time to particular circumstances, since a protocol stack often cannot be brought down to install extra management control features.

In view of that, DiPS+ offers a management meta-level on top of the basic data plane of components and connectors. Packets are transparently intercepted and delegated to this higher level, where they can be controlled, interpreted, manipulated, etc.

As already mentioned in the previous section, architectural connectors are represented in the DiPS+ design by an explicit packet receiver and forwarder. Offering explicit packet receivers and forwarders does not only uncouple units from each other, but also provides attachment hooks for the management layer. Such hooks are designed as policies, which are responsible for the handling of incoming and out going packets.

**Policies.** The typical solution for open systems that are customizable at run-time is a meta-object protocol (MOP) [KdRB91]. Basically, a MOP specifies a generic interface to a higher (meta-)level, used to add message processing that typically does not belong to the purely functional behavior of the system (e.g. monitoring, fault handling, concurrency control, etc.).

The DiPS+ design allows transparent packet interception at the packet receivers and forwarders via the associated `Policy` object. The policy describes the interface of the management meta-level as shown in Listing 3.1. Via this interface, a packet object is intercepted at the packet receiver (or forwarder) and delegated to the meta-level.

A policy describes the strategy to handle each incoming packet. The strategy delegates each packet to a number of `ManagementModule` objects (see also Listing 3.1), which may be registered by an administration tool at application level. Each individual management module describes the processing of a packet, for instance, increasing a generic packet counter, interpreting specific information from

[3] Open implementation is discussed in Chapter 2 (page 25).

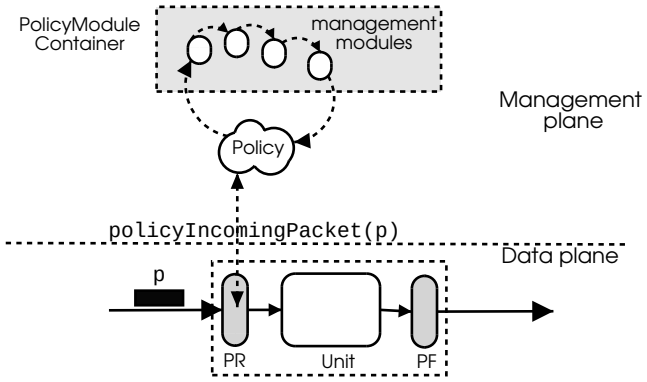



Figure 3.10: A DiPS+ component with a policy object at its packet receiver. The policy delegates incoming packets to a pipeline of management modules.

the packet, or even attaching to the packet extra meta-information based on information accumulated over time.

Generic management modules can be reused from a repository. Yet, the policy design also allows application-specific modules to be registered. To this end, multiple management modules are connected into a pipeline. Figure 3.10 illustrates a packet receiver policy and shows how incoming packets are bypassed to a pipeline of management modules at the meta-level. The last module delivers the packet back to the packet receiver (or forwarder).

**Customizing policies.** To hide DiPS+ meta-level details, application level tools can provide their specific management modules in a so-called `PolicyModuleContainer` class. This allows applications to customize in a controlled manner how packets are handled at the management plane.

### 3.4.2 Concurrency management

If multiple processing resources are concurrently active, their allocation needs to be balanced throughout the system to avoid bottlenecks in the component pipeline. The goal is to maximize the processing capacity of each component area and, by consequence, the overall performance. In terms of the assembly line metaphor, concurrency management means deciding how many of the available workers are best allocated with each machine to obtain the maximum throughput.[4]

Two flavors of management can be identified. The first type called parameter tuning allows manipulation of certain functional parameters. An object or component that is managed in this way must offer an interface to set and read the

[4]Throughput in the assembly line metaphor means the number of finished products that are delivered per time unit.

```
public interface ManagementMetaInterface {
  void setOverflowStrategy(OverflowStrategy strategy);
  void setUnderflowStrategy(UnderflowStrategy strategy);
}
```

Listing 3.2: The management interface of a concurrency unit.

attributes under control ("getters" and "setters"), comparable to well-known management systems such as the SNMP protocol [CFSD], Java management extensions (JMX) [Sunb] or Java Beans [Sund].

This type of parameter management, however, does not allow to radically change the internal behavior of the managed concurrency unit in a controlled way. For example, in order to change the unit's behavior in case of buffer overflow, relevant internal attributes – such as the packet queue – must be exposed. Possible queue manipulations to deal with overflow are increasing its size or removing low-priority packets. Yet, exposing internal attributes releases control on how the packet queue is used and manipulated. Obviously, such details should remain hidden behind the component's interface to avoid side-effects.

To this end, a more specific type of management, dealing with behavioral rather than functional parameters, is more appropriate. Behavior management allows changing internal strategies that describe how the system should behave. A major advantage of this approach is that it allows to change a system while all implementation details are kept internal. Again, this type of management refers to the open implementation principle.

**Management meta-interface.** In view of behavior management, DiPS+ offers applications a management interface to concurrency unit strategies (see Listing 3.2), which enables, for instance, to install an appropriate buffer overflow strategy. For example, when the buffer of a concurrency unit gets filled, its overflow strategy may decide to increase the buffer size, remove low priority packets, or remove all packets arriving after the queue has been completely filled. Similarly, an underflow strategy can decide to decrease queue length when only a fraction of its capacity gets filled.

Figure 3.11 sketches the global concurrency management picture. In the example shown, two concurrency units are customized. On the left side of the figure, extra packet handlers are associated with a concurrency unit to increase the level of parallelism (`handler.setCUnit(u1)`). Similarly, a packet handler can be removed and migrated to another concurrency unit. On the right side, the overflow strategy of the concurrency unit is customized by calling the management interface (`u2.setOverflowStrategy(...)`) The underflow strategy may be changed in a similar way.

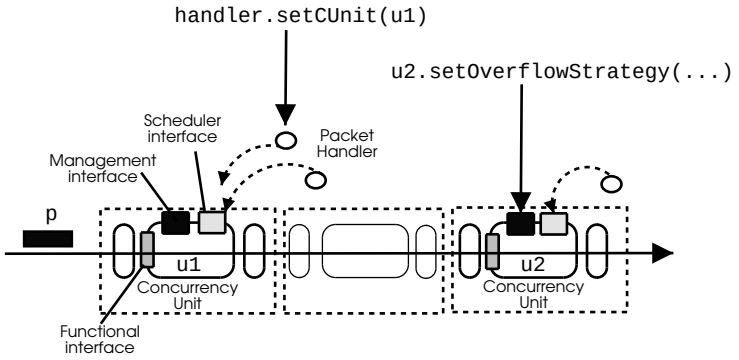



Figure 3.11: Two DiPS+ concurrency units with three interfaces: the functional interface is used to deliver packets to a unit; the management interface to set buffer overflow and underflow strategies; and the scheduler interface to associate a packet handler with a concurrency unit.

## 3.5 The DiPS+ prototype

This section introduces the DiPS+ prototype. It provides the background necessary to understand the chapters to come, especially those concerning testing (Chapter 5) and framework optimization (Chapter 6). For this reason, we pay special attention to the DiPS+ builder, which generates a running system out of the architectural description of a DiPS+ protocol stack.

### 3.5.1 Building a protocol stack

The DiPS+ framework offers tool support to facilitate protocol stack building. The overall DiPS+ builder design is illustrated in Figure 3.12. The design distinguishes four properties, a generic parser, property-dependent builders and the protocol stack, which encapsulates all property objects.

**Properties.** The `StackProperty` class represents the architecture of a protocol stack, a protocol layer, a datalink interface, or a component. DiPS+ identifies a concrete property type for each of them. First, the protocol stack structure property describes the "helicopter view" of the system. This view represents the layers in the stack, the kind of socket that is offered to applications and the datalink interface to the network driver at the bottom of the stack. Secondly, the layer structure property zooms in on components and connectors to provide an architectural component view. Thirdly, the interface property describes how a datalink interface must be configured. Finally, the component property describes a component as a composition of a packet receiver, a unit and one or more packet forwarders. This property is optional since most components are very simple to create anyway.

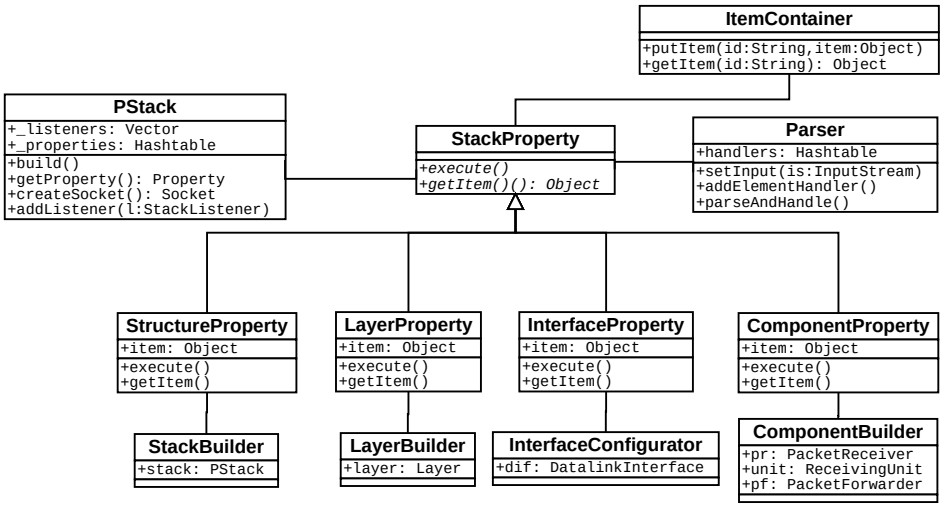



Figure 3.12: The design of the DiPS+ builder with four types of stack properties, a generic parser, a specific builder for each type of property, and the protocol stack that collects all property instances.

Each property class is associated with an `ItemContainer` and a `Parser`. The item container collects all instantiated items (e.g. components, connectors, layers, sockets, etc.) associated with a particular property. The parser is able to interpret a property description. In addition, a property object links the DiPS+ parser with a property-specific builder, i.e. the object that knows how to create (build) a property from the parsed entities.

**Parsing.** The DiPS+ parser translates a property description into a manageable format for the builder. The generic DiPS+ parser can be customized by registering specific handler objects for every architectural entity (layer, component, connector and interface). Each handler delegates the creation of an entity to the builder associated with this property.

The parser can be customized for handling a new entity type, simply by providing a specific `ElementHandler` object for it. The handler delegates each parsed element to a specific builder, which is associated with the property being parsed. For example, a layer parser is able to interpret and handle each element inside a layer description (e.g. a component, connector, dispatcher, etc.), and delegates the creation of each of these elements to a layer builder.

**Building.** Each property has an associated builder object, which is responsible for creating, connecting and configuring the property items that have been parsed by the DiPS+ parser. Multiple builders cooperate in building a protocol stack. The stack builder, for instance, creates and configures a protocol stack by parsing

```
1  public buildStack(){
2     // ...
3    PStack pstack = new PStack();
4    pstack.addProperty(new StructureProperty(pstack, "tcpipethlo_structure.xml''));
5    pstack.addProperty(new IfconfigProperty(pstack, " ethlo_ifconfig .xml''));
6    pstack.build ();
7
8    UDPDatagramSocket dsock =
9         (UDPDatagramSocket) pstack.createSocket(socketType);
10    // ...
11 }
```

Listing 3.3: Building a DiPS+ protocol stack.

the stack structure property and instantiating layers, layer glues, the socket layer and the datalink interface layers. To this end, the stack builder delegates the building of the layers themselves to the layer builder, which knows how to create, connect and configure components.

In order to facilitate the component configuration process, DiPS+ offers a generic resource configurator, which associates a DiPS+ unit with the resource to be configured. Separating resources from the unit implementation makes changing the implementation of either resource or unit very easy. For example, the layer architecture description in Listing 3.5 shows a configurator for the `outrouter` component (line 10), which is responsible for packet routing from one network to another. This component is configured with two resources (lines 4 and 7), for routing and addressing respectively. The former encapsulates a routing table, which identifies the next network node to reach a given destination. The latter encapsulates knowledge on address comparison (e.g. to match a given IP address to a network mask). These two resources are configured by a configurator (lines 20 and 21), which links the outrouter component with the two resources.

**PStack.** The `PStack` object represents a protocol stack and encapsulates all its properties. As such, it forms a central access point for managing internal resources via the properties. As Listing 3.3 illustrates, the `PStack` object offers a simple way to `build()` a protocol stack (line 6) after all properties have been added (lines 4 and 5).

### 3.5.2 Architecture description

The DiPS+ architecture is represented as an XML [HM02] description. Such description identifies the core architecture entities like components and protocol layers as well as how these entities are interconnected. To this end, descriptions for component connectors and layer glues, dispatchers and concurrency units are provided as well.

The DiPS+ architecture description is decomposed into an overall protocol stack description and a layer description for each protocol in the stack. The protocol stack description identifies the protocol layers, the socket type, and one or more datalink interfaces. The socket represents the endpoint applications use to communicate with the protocol stack. A datalink interface provides the access point that communicates with the network interface hardware. Layers are described separately, since they are in turn a composition of cooperating components. Components themselves are very fine-grained and for this reason not described separately.

Listing 3.4 shows an architecture description for a TCP/IP protocol stack. It clearly describes the essential elements of a protocol stack: layers, layer glues, datalink interfaces and sockets. The example shows four protocol layers, i.e. `TCP`, `IP`, `Ethernet` and `Loopback`. Subsequently it describes how these layers are connected using layer glue, i.e. `IPonLoopback`, `IPOnEthernet` and `TCPonIP`. Apart from the "normal" protocol layers and their associated glues, the boundary layers, i.e. the upper and lower layers of the stack, are described as well. Two lower layers provide access to the datalink interfaces (lines 24 and 27): `DatalinkInterface-Loopback`, an interface that re-launches incoming packets back into the bottom layer of the protocol stack, and `DatalinkInterfaceUDPTunnel`, which "tunnels" incoming packets over a standard UDP/IP connection. Finally, line 31 defines the upper layer of the protocol stack, which provides the application endpoint (i.c. a TCP socket).

Listing 3.5 zooms in on the IP layer of this protocol stack. It lists all essential elements layers can be composed of: components (lines 10, 10–16, and 35), connectors (lines 25–31, and 44–46), a dispatcher (lines 39–42), a concurrency unit (line 37), layer resources (lines 4 and 7), and the upper and lower entry point (lines 24 and 43). Each of these items is represented as such in the architecture description, which makes the listing self-explaining.

Having an architecture description apart from the implementation has major advantages. First of all, it allows different compositions to be configured without having to write programming code or to change or recompile the source code of individual units.

A second advantage is that it allows specific (optimizing or test) builders to be applied on the same architecture description. Testing a protocol layer in isolation, for instance, reuses the architecture description, but the layer is created in a different context (i.e. a test case instead of a protocol stack).

Finally, a separate architecture description allows for optimization, in the sense that an optimizer can analyze the architecture and change it in order to become more efficient. When, for instance, a network router is known to be connected to two networks with the same maximum segment size, it can be reconfigured to omit reassembly and re-fragmentation of forwarded packets, since they are fragmented in sufficiently small pieces already. Only packets for local delivery must

```
<stack name="TCP/IP Stack">
  <!-- Layer definitions -->
  <layer name="Loopback"
         class="dips.protocol.loopback.LoopbackCore"
          configfile ="loopbacklayer.xml"/>
  <layer name="Ethernet"
         class="dips.protocol.ethernet.EthernetCore"
          configfile ="ethernetlayer.xml"/>
  <layer name="IP"
         class="dips.protocol.ip.ipv4.IPv4Core" configfile="ipv4layer.xml"/>
  <layer name="TCP"
         class="dips.protocol.tcp.TCPCore" configfile="tcplayer.xml"/>
  <!-- Glue definitions to connect the layers-->
  <glue  name=''IPonLoopback''
         class="dips.protocol.glue.IPonLoopback"
         upper_layer="IP" lower_layer="Loopback"/>
  <glue  name=''IPonEthernet''
         class="dips.protocol.glue.IPonEthernet"
         upper_layer="IP" lower_layer="Ethernet"/>
  <glue  name=''TCPonIP''
         class="dips.protocol.glue.TCPonIP"
         upper_layer="TCP" lower_layer="IP"/>
  <!-- Datalink interface definitions -->
  <interface name="lo"
         class="dips.iface.loopback.DatalinkInterfaceLoopback"
         connect_to_layer="Loopback"/>
  <interface name="veth0"
         class="dips.iface.udp.DatalinkInterfaceUDPTunnel"
         connect_to_layer="Ethernet"/>
  <!-- Socket definition -->
  <sockettype class="dips.protocol.tcp.TCPSocket"
         connect_to_layer="TCP"/>
</stack>
```

Listing 3.4: An example of a structure property description of a DiPS+ protocol stack with four layers (Loopback, Ethernet, IP, and TCP), the glue that connects these layers (IPonLoopback, IPonEthernet, and TCPonIP), two datalink interfaces (lo and veth0), and a TCP socket.

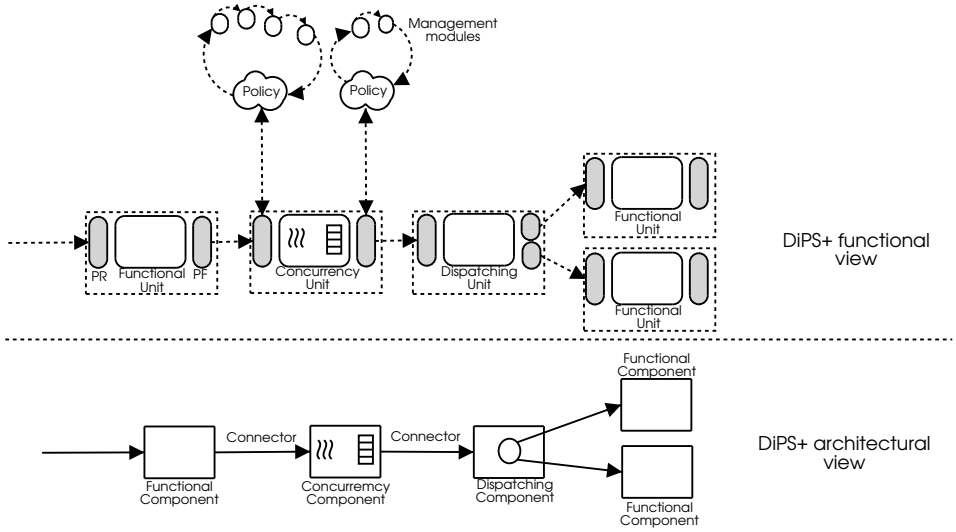



Figure 3.13: Architectural and functional view on a DiPS+ system.

be reassembled in this case.

Once a running system has been created from the architecture description, each packet forwarder knows the packet receiver of the next component in the pipeline. Registering the forwarder of the source component with the packet receiver of the destination is done by a DiPS+ builder.

### 3.5.3 Developer roles

The architectural and the functional view on a DiPS+ system are displayed in Figure 3.13. The clear mapping between adjacent levels makes it easier for developers to learn how the system works while allowing specialists to zoom in on the appropriate level of detail.

For each view, DiPS+ identifies one or more developer roles. The *DiPS+ architect role* designs the overall system structure without being distracted by low-level implementation details (as every architect does). This structure is basically a configuration of connected components (see for example Listing 3.5). To emphasize the global picture, however, a DiPS+ architect can zoom out to higher level abstractions such as protocol layer, datalink interface or socket (e.g. Listing 3.4). DiPS+ architects, like all software architects, can best be compared with architects of a building. They too zoom out to a "helicopter view" of the system being designed, thereby identifying abstractions such as bedroom, bathroom, kitchen, toilet, etc.

```
<layer name="ipv4layer">

<!-- down going path -->
  <resource factory="dips.generic.RoutingTable"
      name="ipRoutingTable"
      class="dips.protocol.ip.ipv4.IPv4SimpleRoutingTableManager"/>
  <resource factory="dips.protocol.ip.IPAddressTable"
      name="ipAddressTable"
      class="dips.protocol.ip.ipv4.IPv4SimpleAddressTableManager"/>
  <component class="dips.protocol.ip.ipv4.IPv4OutRouter"
      name="outrouter"/>
  <component class="dips.protocol.ip.ipv4.IPv4HeaderConstructor"
      name="hdrcons"/>
  <component class="dips.protocol.arp.ARP"
      name="arp"/>
  <component class="dips.generic.DatalinkLowerLayerKeyTagger"
      name="lowerlayerkeytagger"/>

  <configurator class="dips.protocol.ip.IPOutRouterConf" name="outrouter">
      <item name="" type="java.lang.String" value="ipRoutingTable"/>
      <item name="" type="java.lang.String" value="ipAddressTable"/>
  </configurator>

  <upperentrypoint name="outrouter"/>
  <connector class="dips.repository.connector.SimplePipeConnector"
      name="cd1"    from="outrouter"     to="hdrcons"/>
  <connector class="dips.repository.connector.SimplePipeConnector"
      name="cd2"    from="hdrcons"       to="dummyarp"/>
  <connector class="dips.repository.connector.SimplePipeConnector"
      name="cd3"    from="dummyarp"      to="lowerlayerkeytagger"/>
  <connector class="dips.repository.connector.SimplePipeConnector"
      name="cd4"    from="lowerlayerkeytagger"  to="downdelivery"/>

<!-- Up going path -->
  <component class="dips.protocol.ip.ipv4.IPv4HeaderParser"
      name="hdrpars"/>
  <component class="dips.repository.concurrencyunit.ActiveUnit"
      name="active_comp"/>
  <dispatcher class="dips.protocol.ip.ipv4.IPv4InRouterReflectionStrategy"
      name="inrouter"/>
  <dispatcher output="local"  name="inrouter" to="updelivery"/>
  <dispatcher output="forward" name="inrouter" to="outrouter"/>
  <lowerentrypoint name="hdrpars"/>
  <connector class="dips.repository.connector.SimplePipeConnector"
      name="cu1"    from="hdrpars" to="active_comp"/>
  <connector class="dips.repository.connector.SimplePipeConnector"
      name="cu2"    from="active_comp" to="inrouter"/>
  <configurator class="dips.repository.configurator.ActiveUnitStarter"
      name="active_comp">
</layer>
```

Listing 3.5: An example of the layer property description for the DiPS+ IPv4 layer.

The *DiPS+ builder role* links units with their packet receiver and forwarder(s), and creates corresponding components. The builder connects components in order to instantiate a running DiPS+ protocol stack. This task can easily be automated by a building tool, as is the case in DiPS+ (see also Listing 3.3). Just as different aspects in building architecture are treated by different "builders" (say an electrician, a kitchen installer or a plumber), DiPS+ identifies different builders to focus on the datalink interface, socket, protocol layer or stack structure.

The *DiPS+ developer role* translates each architectural component into a DiPS+ unit design, and implements this design. A DiPS+ developer is comparable to a kitchen or bathroom designer, who designs and creates specific elements of a house. Similarly, a DiPS+ system is not implemented by one programmer, but each functional aspect can be handled by a different domain specialist. Each of these developers must follow strict rules enforced by the DiPS+ component framework.

Finally, the *DiPS+ administrator role* concentrates on the system's behavior rather than on its functionality. To this end, a DiPS+ administrator develops specific packet handling policies and attaches those to a packet receiver and/or forwarder. Administrators may have different goals such as testing, throughput monitoring, component hot-swapping, etc. They can be compared with interior designers or landscape architects who concentrate on the overall view, decoration and concept of the in- or outside of a building.

## 3.6 Conclusion

### 3.6.1 Contributions of DiPS+

Research in the context of this dissertation has contributed to the evolution of the original DiPS framework [Mat99] to DiPS+. This dissertation transforms DiPS from a programming framework and architecture, which focuses on providing support for protocol stack programming, to a manageable platform that offers support throughout the software development process, both at design- and at runtime.

The first contribution lies in the uniform design of different unit types, which allows treating units (and components) of various types consistently. This has two major advantages. It allows functional, concurrency and dispatching units as well as composed units (e.g. layers) to be treated equally in case unit characteristics are irrelevant. For example, testing a single fine-grained component (i.e. unit testing) does not differ from testing a component composition (integration testing). The second advantage is that the unit design identifies different types for separate concerns (functionality, dispatching and concurrency). This allows concentrating on a single concern without being distracted by other concerns scattered across the same functional code. Without such concern separation, management and control

become very difficult.

The second contribution is that a DiPS+ component is composed of a packet receiver, a unit and a packet forwarder, which again has two advantages. First of all, the packet forwarder and receiver uncouple adjacent units, similar to an architectural connector. This unit independence allows developing, controlling and testing units isolated from other units or the environment. To this end, units can only communicate indirectly via the packet forwarder and receiver, which are controlled by the DiPS+ framework. The second advantage is that splitting architectural connectors into a forwarder and a receiver permits fine-grained control of incoming and out going unit traffic.

The third major contribution consists of the management plane, which enables intercepting packets from the data plane and fine-tuning concurrency units by a system administrator. Since the management plane is orthogonal to the data plane, it can be removed in case management and control are not needed. Moreover, since units are unaware of the management plane being present, they can be reused in both cases, i.e. with or without management and control. Removing management overhead can be crucial to systems with very stringent performance requirements.

### 3.6.2 Summary

This chapter has presented the DiPS+ architecture and design of both management and data plane. We can conclude that the objectives of DiPS+ have been met, for several reasons.

**Uniformity.** Central to the DiPS+ design is the component, and especially the different unit types that identify specific unit concerns (e.g. functionality, concurrency and dispatching). Units are very fine-grained entities designed to be easily composable in multiple configurations. On top, the framework itself offers the layer abstraction, which is basically a composition of fine-grained components.

**Clear mapping between architecture and design.** The data and management plane are well-recognizable in the DiPS+ design as well as the architecture. Moreover, the design of the data plane provides a protocol layer as a separate entity, which corresponds to the architectural layered view of a typical protocol stack.

**Controllability.** Components are equipped with explicit entry and exit points that can be customized with specific packet flow control policies. As such, they enable controlling the packet flow, since they can intercept incoming and out going packets. The management interface offered by concurrency units allows balancing packet handlers over the system in order to optimize throughput. The management plane is clearly separated from the data plane, and can easily be removed when not needed. Packet receivers and forwarders may provide a packet intercepting policy, which forms the hook for attaching the management plane. By consequence, functionality (which resides in the units) is not affected by the presence or absence of the management plane.

*"640K ought to be enough for anybody."*
*– From Bill Gates, in 1981.*

# Chapter 4

# Self-adaptive system behavior

The development of robust and adaptable system software is motivated by three concerns: strong dependence on Internet services, the variable character of network traffic, and the wide variety of client devices that are connected to the network. First of all, Internet services have become critical both for driving large businesses and for personal productivity. Global enterprises are increasingly dependent upon Internet-based applications (e.g. for e-commerce, supply chain management, or financial accounting). Yet, many individuals consider e-mail and Internet access as indispensable too. This growing dependence upon Internet services underlines the importance of their availability, scalability and capability to deal with varying loads.

Secondly, Internet servers (such as web, e-mail or file servers) must be highly flexible to handle the burstiness inherent to network load. Popular services must stand huge variations in demand, with request bursts at times when they are most valuable. As the slash-dot effect[1] illustrates, more than 100-fold increases in demand are not uncommon when a site becomes popular [WS00]. Another example of Internet burstiness that can cause similar effects, relates to denial-of-service attacks. Such attacks involve simultaneous requests being sent from thousands of sites across the Internet. They have had a major impact on the performance of sites such as Buy.com, Yahoo!, and whitehouse.gov [Lem01b], which inspired the startup of several companies to combat this problem [Lem01a].

[1] This term is often used to denote sudden increases in demand when a web site becomes popular. The famous portal site `http://slashdot.org` often brings down resource-limited web servers when publishing a link to them on its main page.

Thirdly, more and more client devices on the network can be of any type. Already a wide variety of networked client devices can be found, such as personal digital assistants (PDAs), mobile telephones, low-end Internet terminals [ITE99], or vehicles equipped with wireless communication channels [KMD+99]. Yet, these types of clients typically have less processing resources available than, say, a personal computer [IYT95]. By consequence, the available processing resources should be employed as efficiently as possible, for instance, by allocating resources to those component areas that need them the most, or to packets with the highest priority.

This chapter presents DMonA, the DiPS+ Monitor Architecture. DMonA is a concrete implementation of the DiPS+ management plane and allows to detect performance bottlenecks, propose solutions to solve them and deploy these solutions at run-time [MDJ+02]. Moreover, it upgrades DiPS+ from a *manually adaptable* system to a *self-adaptable* system that can adjust itself to drastically changing load circumstances. In other words, via DMonA a protocol stack can assign packet handlers to concurrency components itself, rather than doing this under human control.

DMonA is designed as a generic framework that can be customized to various domains of system management and control. As proof of concept, this chapter applies DMonA to two complementary types of overload management. On the one hand, overload is (partially) handled by dynamically re-balancing internal processing threads over DiPS+ component areas. For example, component areas that consume few processing resources could be given priority over resource intensive areas. Or, the other way around, processing power can be moved from low to high-priority areas. On the other hand, overload conditions are handled by controlling and prioritizing incoming requests. For example, one may favor un-encrypted or cached requests, since they require less processing power. Another example is to delay scheduling of incoming packets until enough work has been accumulated to amortize the startup cost.

DMonA is also applicable to other domains than load management, and even to other sub-systems than protocol stacks. On the one hand, intrusion detection[2] [Nor99], for instance, may be done by intercepting incoming packets in the protocol stack and trying to recognize suspicious patterns. A possible solution is to mark suspicious packets so that they can be removed by a packet filter in the component pipeline. On the other hand, if DiPS+ would be used to develop file systems (instead of protocol stacks), the DMonA approach could be used to optimize I/O, for instance, by using adaptive file system policies for caching and pre-fetching. Such adaptive file system behavior requires I/O monitoring, i.e. recognizing and classifying I/O requests [MR97b, MR97a], and selecting the most appropriate caching and pre-fetching policy based on this information [MR96]. In addition, perform-

[2] An intrusion detection system (IDS) attempts to detect an intruder breaking into the system or a legitimate user misusing system resources. An IDS must monitor the system and observe deviations from its normal behavior. In addition, it must cope with changing system behavior over time as new applications are being added.

ance sensors may be used to tune policy parameters for the system environment [MER96].

The chapter is structured as follows. Section 4.1 proposes the design of the DMonA framework for self-management and explains how it relates to DiPS+. Section 4.2 presents a DMonA case study for load management. Section 4.3 presents the performance benefit of DMonA, followed by a short summary in Section 4.4.

## 4.1 The DMonA approach

DMonA, the Dips+ Monitoring Architecture, supports self-management of DiPS+ systems. The DMonA self-management approach is responsible for three subtasks: detecting (upcoming) problems (*self-monitoring*), proposing a solution to the problem and deploying this solution in the DiPS+ data plane (*self-healing*).

Before describing the design of the DMonA framework and how it can be customized to specific management domains, we briefly repeat the essential management elements in the DiPS+ design, since DMonA relies on them.

### 4.1.1 The relationship with DiPS+

With respect to self-monitoring and self-healing, DMonA relies on the architecture and design of the DiPS+ data plane. As such, DMonA is a concrete instance of the generic management plane presented in the previous chapter.[3] To situate DMonA in the global DiPS+ picture, Figure 4.1 shows how DiPS+ and DMonA are related, i.e. how the DMonA management plane may extract information from DiPS+, and how DMonA decisions for change can be deployed in the DiPS+ data plane.

On the one hand, self-monitoring is enabled in DiPS+ by associating a policy with a component's packet receiver and/or forwarder. Such a policy decides how packets are processed before or after being handled by a DiPS+ unit. In case of DMonA, a policy may for instance intercept packets and subsequently delegate them to a pipeline of management modules in the management plane, each of which extracts from them specific (meta-)information. Policies have been introduced and explained in Section 3.4.1. Figure 3.10 shows a packet receiver policy with a single management module.

On the other hand, self-healing[4] in the DiPS+ data plane is enabled via the management interface that is exposed by concurrency units. The management interface opens the concurrency unit black-box to customize its buffer overflow or underflow policy, or its packet scheduling policy. Section 3.4.2 describes in detail how concurrency behavior can be managed.

[3] For more information about the DiPS+ management layer, see also Section 3.4 (page 55).

[4] Although self-healing is a much broader research domain, this dissertation focuses on its use in the context of concurrency control. See also 3.2.1 (page 39).

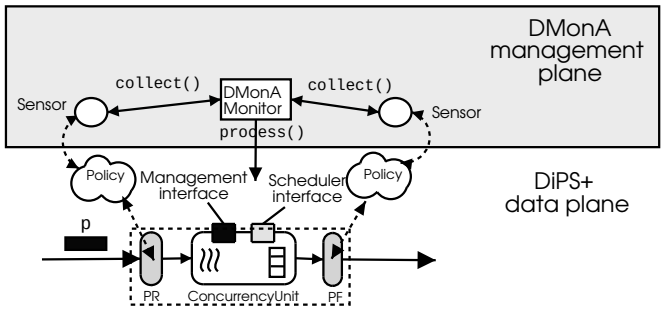



Figure 4.1: DMonA (DiPS+ Monitoring Architecture) as the management plane on top of the DiPS+ data plane.

### 4.1.2 The DMonA design

The DMonA management plane should be thought of as a generic framework on top of DiPS+. DMonA itself does not provide specific monitoring or adaptation algorithms. Instead, it is designed as an open platform that allows specific policies or algorithms to be plugged in. DMonA focuses on automating management tasks, i.e. processing the information that has been collected in the management module(s), proposing a solution in case a problem has been detected, and deploying this solution.

The `Monitor` class is the driving entity in the DMonA design. Figure 4.2 shows how the monitor is associated with the other classes in DMonA: the `Collector`, the `MonitorPolicy`, and the management `TargetId`. DMonA provides a management framework that allows collectors and policies to be customized as framework hooks.

A collector object collects the information stored in a `Collectable` object, for instance, a management module. In addition, it processes this information, for example by calculating the packet throughput (based on the number of intercepted packets during some time interval or the average packet data size), or even by neural network calculations to detect particular patterns in the packet stream [MR97b]).

The monitor describes the DMonA behavior as a generic template (cfr. the Template design pattern [GHJV95]). Its main task is to trigger its associated collectors for information retrieval, and deliver the collected data to the monitor policy for further processing. As such, it determines the flow of control by deciding when the collector and the policy are called[5].

The monitor policy decides whether or not adaptation is necessary, for instance, to improve system performance. Such a policy may compare, for example, the arrival rate at a concurrency unit with its departure rate. In case the arrival

[5]This inversion of control is a typical framework characteristic, cfr. Section 2.2.1 (page 15).

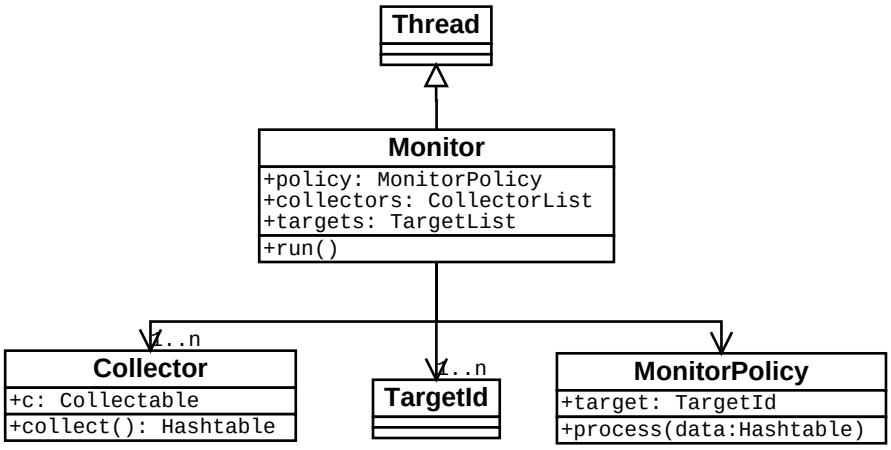



Figure 4.2: The design of DMonA.

rate largely exceeds the departure rate, a good solution would be to migrate processing resources (threads) to the concurrency unit associated with the overloaded component area. During peak loads, however, when free thread resources may no longer be available, an admission control approach would be more appropriate. Such approach may limit the input rate by dropping particular packets. Depending on the optimizations aimed for and the processing power available, the most appropriate processing algorithm can be installed as a monitor policy.

The management target identifier represents the target for management, but hides the actual implementation from the monitor. The policy suggests adaptations that may be employed by a third party having control over all resources. A particular policy implementation may call a thread manager, for instance, to add or remove a DiPS+ packet handlers to or from a particular concurrency unit. The monitor policy, therefore, calls the thread manager and provides it with the concurrency unit target identifier. In case of adding a thread, the thread manager associates an extra packet handler with the provided concurrency unit.

To control the system globally, local monitors should be coordinated by a global system monitor. This prevents that local monitor policies become counterproductive. A local policy, for instance, cannot distribute threads appropriately without a global view on the system. The global monitor, therefore, actively gathers information from the individual monitors and processes this information to announce a system-wide decision. This results in specific tasks that are delegated to the individual monitors. The DMonA framework allows both monitors that are associated with a single unit, and monitors that have a global system view, i.e. associated with multiple concurrency units.

## 4.2 A case study: load management

This section first proposes two often used thread allocation models (thread-per-request and thread pool) and identifies their disadvantages concerning adaptability and control. Subsequently, the section describes how DMonA has been customized to be used for load management. This allows to demonstrate its power as a generic framework and its advantages compared to other models.

Load management can be viewed from two perspectives. From a concurrency point of view, load management distributes the available processing power (threads) across the system such that the overall system performance is optimized. By consequence, the management plane should be able to detect performance bottlenecks, i.e. component areas where packets arrive faster than they can be processed. In addition, the management plane must solve these bottlenecks by migrating processing resources, which are associated with the concurrency component in front of a component area, from underloaded to overloaded component areas.

At the same time, load management can be viewed from an access control perspective. Access control deals with overload by limiting or shaping the arrival rate of new requests to a sustainable level. In order to do so, admission control classifies incoming packets in order of priority based on several parameters, such as their destination, data size, encapsulated protocol, packet type (connection establishment or data transfer), or application-specific preferences passed via meta-information. Low priority packets may be handled separately, postponed until all higher priority packets have been processed, or simply dropped to release system resources for higher priority packets.

Concurrency and admission control are compatible and synergetic though. Obviously, thread re-allocation is only relevant if enough thread resources are available to handle incoming packets. For extremely high loads, however, even optimal resource allocation will not prevent the system from being overwhelmed. Instead of balancing resources, such cases require controlling and reducing the input to prevent the system from collapsing under the high load pressure.

### 4.2.1 Thread allocation models

Multiprogramming is a well-known technique that allows multiple programs[6] to be executed on a mono-processor machine as if they were executed in parallel. However, multiprogramming on a mono-processor system only offers *virtual* parallelism, since only one thread at a time can use the processor. For that reason, mono-processor operating systems offer preemptive scheduling algorithms that transparently switch the processor between threads, by suspending the active thread and resuming one of the threads that are ready to execute.

[6] A program can consist of one or more processes, each of which can have one or more internal threads.

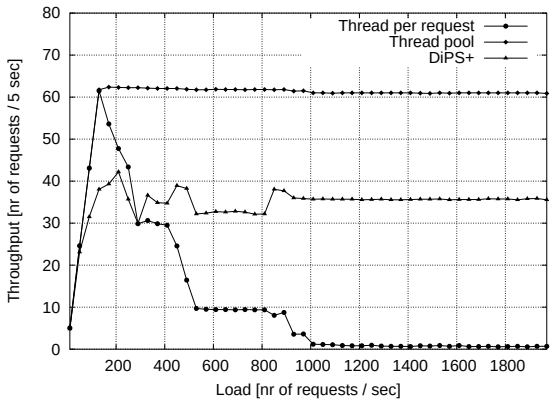



Figure 4.3: Illustration of three thread allocation models: the thread-per-request, the bounded thread pool, and the DiPS+ model. The component pipeline consists of three components, with increasing load. This creates a bottleneck at the end of the pipeline for the DiPS+ model, which inserts a concurrency unit in front of each component (resulting in three component areas). The thread-per-request model is performing well until 300 concurrent requests. As load increases, throughput degrades exponentionally. The throughput of the other two models remains constant for high loads. The overhead of concurrency components inside the component pipeline, and the imperfect balance of processing resources explain the throughput difference between the bounded thread pool model and DiPS+.

Multiprogramming and preemptive scheduling hide concurrency from applications on top of the operating system. This transparency is a great advantage, since it saves these applications from considering low-level scheduling details. The downside of this virtualization, however, is that it prevents applications from customizing thread scheduling behavior [WC01, KLM+97].

We present two well-known thread allocation models and show their weaknesses in the context of overload. After that, we describe how these weaknesses can be solved by a totally different approach and system architecture, which enables fine-grained adaptability.

- *Thread-per-request.* The thread-per-request model creates a new thread for every incoming request (e.g. an incoming network packet). Although this model works fine during normal load conditions, thread scheduling overhead leads to serious performance degradation when the number of threads is high (see Figure 4.3). Consequently, this model's scalability is limited by the number of threads the system can start without performance degradation due to thread-switching. In addition, thread scheduling policies are typically hidden inside the operating system, which prevents applications from customizing

them.

- *Bounded thread pool.* The bounded thread pool model limits the number of threads and reuses threads from a a so-called thread pool to prevent the cost of thread initialization. Whenever a thread has finished its job, it is released and put back in the thread pool. The thread pool size is a crucial parameter in this respect. If the size is chosen too large, thread switching overhead becomes the bottleneck, comparable to the thread-per-request model. If, by contrast, the number of threads in the pool is too small, the processor may not be fully exploited (e.g. because all threads are waiting for I/O). Figure 4.3 shows that the throughput of the bounded thread pool model remains constant at the maximum of the thread-per-request model while the load increases.

Although both the thread-per-request and the thread pool model make it easier to write applications in a multiprogramming context, they share some problematic characteristics. First of all, both models hide scheduling policies, which prevents applications from customizing them. In other words, the system's external interface is not open to application-specific customization. Secondly, their main multiprogramming objective is efficiency and safety of thread virtualization [WC01], rather than fine-grained thread management and control. As a result, the internal architecture and design of both models is not open to such internal flexibility. By consequence, even if the system's external interface were open, the design of the system would still limit its management and control capabilities for two reasons.

The first reason is that these models do not allow to inspect the request stream or interpret particular request information during its traversal through the protocol stack. By consequence, scheduling policies cannot take into account request-specific information. Some requests may be inexpensive to process (e.g. a cached ARP [Plu82] address translation request), while others will be processing-intensive (e.g. to encrypt or fragment a data packet). Both models associate to an incoming request a thread, which guides it through the system. This means that all request characteristics that may be used to influence the scheduling policy must be known when the request arrives. Such characteristics may be, for instance, a packet's source or destination address, its data size, or the type of data it is carrying. Yet, it may well be the case that not all request-specific information is available at packet arrival-time. The network source address of a network packet arriving at the bottom of a protocol stack, for instance, is encapsulated in one of its protocol headers, which are only parsed as the packet traverses the stack. Having such information available can improve the system's resource allocation to those requests with the highest (application-specific) priority.

The second reason why both models are not open to management and control is their lack of control over the execution of a thread, once it starts processing a request. This prevents them from detecting the source of a performance bottleneck

and, by consequence, handling it appropriately. Both models arbitrarily queue or drop requests that cannot be processed immediately. Yet, the system should be able to decide which requests must be dropped first in order to preserve the highest possible throughput.

The following two classes of thread allocation improve the allocation of processing resources in the system and allow their scheduling policies to be customized to application-specific requirements based on request-specific characteristics.

- *Event-based parallelism.* Event-based approaches model a task flow as a finite state machine, with non-blocking functional modules representing the states, and events representing state transitions. Event-based systems are driven by event scheduling instead of thread scheduling. Events that cannot be handled immediately are queued in an event queue, with its own scheduling policy. Having multiple event queues that are intelligently distributed over the system allows differentiating which queues are most important to be de-queued. Event-driven systems tend to be more robust to extreme load conditions [Wel02, WHS01].

- *DiPS+ concurrency units.* DiPS+ offers a means to control threads in a fine-grained manner, by sub-dividing the component pipeline into component areas. Each component area is preceded by a concurrency component, which has a thread pool and a packet queue associated with it. The more threads are allocated to a concurrency unit, the more packets can be processed in parallel. In addition, DiPS+ allows to control and manipulate how sub-tasks (i.e. component areas) are interleaved by choosing the queue from which the next packet is selected.

  In addition, each concurrency unit may have a specific packet scheduling policy, which can use packet-specific (meta-)information, possibly produced during the packet's traversal through the stack. DiPS+ allows to customize packet scheduling inside the component pipeline and, as such, to focus on the packets with the highest priority.

  Figure 4.3 shows that DiPS+ throughput remains constant while the load increases. However, the bounded thread pool model outperforms DiPS+, which can be explained by the imperfect balance of packet handlers across the component areas, and the overhead introduced by the concurrency components in the component pipeline. A monitoring and management system like DMonA is needed to reach the best-of-both-worlds solution: being able to schedule packets inside the component pipeline based on application-specific requirements, while still providing reasonable throughput by solving internal bottlenecks.

Both event-based models and DiPS+ enable to apply to each component area a specific thread scheduling policy. Since incoming requests have been partially processed at that point in the component pipeline, these scheduling policies may be

based on request-specific information. Such fine-grained control is much more difficult to achieve with the thread-per-request and the thread pool model, since these associate a request with a thread that guides it throughout the system without further control.

In addition, both the event-based and the DiPS+ model identify sub-tasks in the system, by means of stages or component areas. On the one hand, this considerably facilitates finding and solving internal throughput bottlenecks, i.e. component areas where much more packets arrive than can be processed. One solution is to concentrate processing resources to high priority component areas. Another way to solve a bottleneck is to manipulate how sub-tasks are interleaved, i.e. to process extra packets from queues that are associated with bottleneck areas.

On the other hand, component areas may help prioritize particular component areas based on application-specific requirements. Prioritizing requests is highly relevant in case of parallel component areas, which is very likely in protocol stacks. One component area may handle new requests (e.g. for connection establishment in TCP, or user authentication in RADIUS [RRSW97]), while a parallel area deals with follow-up requests (e.g. data transfer over an established TCP connection, or RADIUS (Remote Authentication Dial in User Service) accounting messages that start or stop a session after a user is authenticated [Rig00]). In such cases, a relevant requirement would be to maximize the number of *completed* requests (e.g. TCP connections or RADIUS sessions). In DiPS+/DMonA this requirement can be translated into a policy that processes first the packets in the queue of the follow-up component area. This prioritization is only possible because the component pipeline can be sub-divided into independent component areas, and because protocol-specific information can be extracted from packets while they traverse the protocol stack. Neither the traditional thread-per-request nor the thread pool model can easily apply such prioritized packet processing based on application-specific requirements, since thread control is impossible once a thread is associated with a request.

### 4.2.2 Locally and dynamically assigning threads

Despite the fine-grained thread allocation control in DiPS+, dynamic self-adaptation support of DMonA is necessary. Indeed, load circumstances may vary arbitrarily and are often very short-term. In view of that, we demonstrate how DMonA may be used to find an optimal balance between system load and available processing power.

**Throughput sensors.** The packet stream is monitored by installing throughput sensors, i.e. management modules that count the number of passed packets. Figure 4.4 shows how sensors are plugged in at both the packet receiver and forwarder of a concurrency unit. The DMonA monitor collects on a regular basis the information stored in both sensors and resets them to start the next collecting phase.

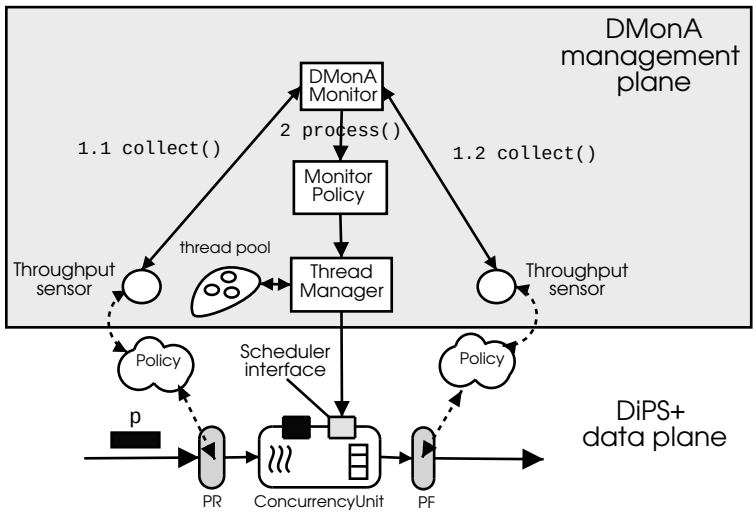



Figure 4.4: Illustration of DMonA attached to DiPS+ via two policies, one associated with the packet receiver and one with the packet forwarder. Processing resources are retrieved from a pool of free resources and allocated to a concurrency unit via its scheduler interface.

**The monitor policy: an example.** Concurrency cannot be managed from a local perspective only. If every concurrency monitor decided autonomously to add or remove packet handlers (see also Figure 4.4), a global optimum would be very difficult to reach. In the worst case, one local concurrency monitor would allocate all available packet handlers, leaving no room for optimizing other concurrency units.

For that reason, concurrency monitoring is provided at two levels: the local monitor policy controls one concurrency unit, while the overall monitor policy in turn controls all local monitors. Local monitor policies can fine-tune global decisions since they have a detailed view on the local status. For example, a local monitor can temporarily hide local status fluctuations from the global level to prevent packet handlers from being migrated constantly from one concurrency unit to another.

The concurrency adaptability policy presented here concentrates on the global view [Des02]. The algorithm adjusts thread scheduling based on the concurrency unit's progress, comparable to the feedback-driven approach proposed by Steere [SGG+99]. Collecting status information, however, is delegated to local monitors, which collect the information from their local management modules. Based on this status information the global monitor policy decides *when and how* to adapt local concurrency behavior to improve performance.

The global monitor policy serves two purposes. First of all, it is responsible for detecting performance bottlenecks. More than one bottleneck can arise, but the algorithm searches for the bottleneck with the smallest throughput. Similarly, the algorithm searches for areas with an arrival rate that is smaller than the departure rate (i.e. areas that can process more requests than they actually receive). Those areas are candidates to free one or more packet handlers in favor of the overloaded area. Secondly, the policy tries to optimize global throughput by distributing all available packet handlers such that processing power is balanced between *underloaded* and *overloaded* component pipes.

The policy is based on the following iterative algorithm, which determines in step $i$ the number of threads for each component area, based on the current throughput for each area, and the throughput and thread distribution in step $i-1$. The current throughput for an area $a$ is represented as $f_a^i$, the number of allocated packet handler threads for a particular area $a$ is $t_a^i$ and the collection of all component areas is $A$. The objective of the algorithm is to reach an equal throughput distribution $f^i$ over all component areas:

$$\forall a \in A : f^i = f_a^i \tag{4.1}$$

In addition, the algorithm presupposes that the throughput for each individual thread remains constant, i.e.:

$$\forall a \in A : \frac{f_a^{i-1}}{t_a^{i-1}} = \frac{f_a^i}{t_a^i} \tag{4.2}$$

Finally, the total number of available threads (N) in the system is known, which results in the following invariant for every step in the algorithm:

$$\sum_{a=1}^{\#A} t_a = N \tag{4.3}$$

Based on formulas 4.3 and 4.2, we can conclude that:

$$\sum_{a=1}^{\#A} \frac{f_a^i * t_a^{i-1}}{f_a^{i-1}} = N \tag{4.4}$$

When we combine formula 4.4 with the objective stated in 4.1 we can extract $f_a^i$, which allows to calculate the global throughput $f^i$ as follows:

$$f^i = f_a^i = (\sum_{a=1}^{\#A} \frac{t_a^{i-1}}{f_a^{i-1}})^{-1} * N \tag{4.5}$$

For each component area, the measured throughput can be calculated by DMonA (both $f_a^{i-1}$ and $f_a^i$). Also the number of allocated threads ($t_a^{i-1}$) is known for

each area. The optimized number of threads for each area based on the measured throughput ($t_a^i$) can then be found by combining equations 4.2 and 4.5:

$$\forall a \in A : t_a^i = \frac{f^i * t_a^{i-1}}{f_a^{i-1}} \tag{4.6}$$

Figure 4.5 represents three sub-tasks as a pipeline composed of component areas, each having a different diameter, which represents the throughput. In this case, processing resources are balanced over the three areas (i.e. three threads for each area). To explain the example, we suppose nine packets could be processed per time unit in case the pipeline were *not* sub-divided in component areas (i.e. each thread guides a packet throughout the system). By consequence, the best possible throughput when component areas are introduced is again nine packets per time unit. However, this result can only be accomplished if processing resources are optimally balanced over the component areas. In other words, if some areas are more processing-intensive than others, they should be allocated the most processing resources too. Figure 4.5 shows an example where processing resources are equally distributed over the three areas, but area C is more processing-intensive than area B and A. Suppose area A requires 20% of the processing-time, area B requires 30%, and area C 50%. Consequently, processing resources should be migrated from areas A and B towards area C to prevent the queue in front of area C from overflowing.

Figure 4.5 illustrates the balancing algorithm intuitively in three steps. On top of the figure, the most critical bottleneck has been detected at the end of the pipeline, thereby restricting the overall throughput considerably. In the middle, this bottleneck has been solved by migrating packet handlers from area B to area C. At this point, area B becomes the bottleneck, since the packet arrival rate in this area largely exceeds its departure rate. At the bottom, Figure 4.5 shows the balanced pipeline with packet handlers optimally distributed across the system according to the load of each individual area.

The algorithm above can be optimized in two ways: by local optimum disturbance and by prioritizing component areas. The former addresses the problem that optimization algorithms tend to result in a stable but sub-optimal distribution. This is the case when the thread distribution is optimal according to the thread distribution algorithm, although local bottlenecks can still be found. In order to prevent this, the optimal distribution is intentionally disturbed from time to time by adding an extra packet handler in an area and re-initiating the distribution algorithm. This technique of using excitations to migrate from one stable state to another, more optimal stable state is well-known in biology and chemistry [Wal93]. The second optimization differentiates between more and less important areas in the system. This is especially relevant for a system architecture with a dispatcher that splits the component pipeline into parallel component areas.

**Self-healing.** Decisions proposed by a monitor policy can be deployed in

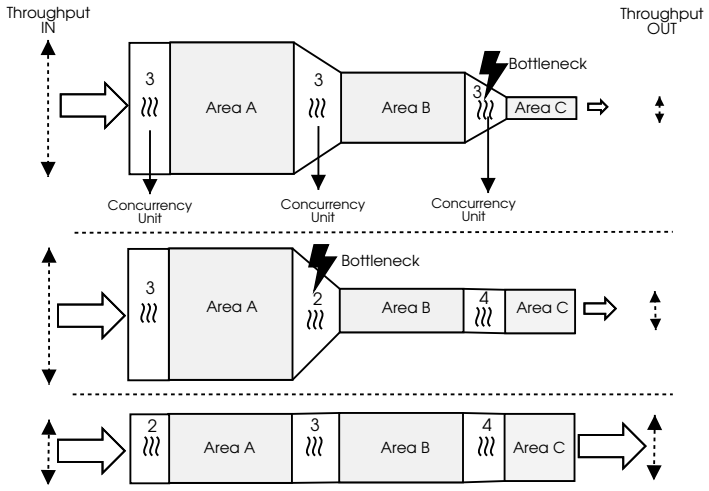



Figure 4.5: Intuitive illustration of balancing processing resources over multiple component areas to resolve bottlenecks. Area A requires 20% of the processing-time, area B 30%, and area C 50%. In the top figure, processing resources are equally distributed over the three component areas (i.e. 33% of the available resources for each area). Since area C is more processing-intensive than area A and B, it becomes the bottleneck (i.e. the packet queue before the area fills). The middle figure shows how DMonA migrates one packet handler to area C. Since this creates a new bottleneck in area B, DMonA moves a packet handler from area A to area B. This results in the optimal balance of processing resources according to the load of each area.

the concurrency unit in two ways.[7] On the one hand, a concurrency unit can be linked or unlinked with a packet handler. This is done via the concurrency unit's scheduler interface. The monitor policy triggers the thread manager when a re-allocation of packet handlers is required. On the other hand, the buffer overflow and underflow policies of a concurrency unit can be replaced by calling its management interface.

### 4.2.3 Admission control

Another approach to handle load management, which is complementary with thread (re)distribution, is controlling the packet arrival rate (e.g. in front of a component area). Such admission control may, for instance, selectively drop low-priority requests to preserve the throughput of the most important requests. This is crucial in case too much requests arrive to be handled by the available processing resources.

Admission control has been effectively employed in networks, for example, to provide quality-of-service guarantees to applications by individually controlling network traffic flows (also known as traffic shaping) [BJS00]. Typically, a leaky bucket algorithm [Tan96] is used to adjust the rate at which incoming packets are forwarded. A leaky bucket forwards packets at a constant rate and, in case the packet arrival rate exceeds a prescribed threshold, stores packets temporarily. When the bucket overflows, packets are dropped or migrated to a lower priority bucket.

A variety of performance metrics have been studied in the context of overload management, including throughput and response-time targets [Wel02, CM02, CMC01], CPU utilization [AL00, CP98, DGH$^+$02] and differentiated service metrics based on a given performance target [KK00, LASS01]. Welsh [Wel02] proposes the 90th percentile response-time as a realistic and intuitive measure of client-perceived system performance. It is defined as follows: if the 90th percentile response-time is t, then 90% of the requests experience a response-time equal to or less than t.

If we applied the DMonA framework in the context of admission control, we should provide the information collectors, the monitor policy and the management target (see Figure 4.6). As a concrete case, we use the 90th percentile approach [Wel02] DMonA.

First of all, a response-time sensor measures the response-times for packets passing through a component area. Such sensors are installed at each concurrency unit's packet forwarder and determine how long it takes between a request leaving the concurrency unit and the release of the associated thread. A DMonA collector may collect the response-times of all packets that have passed through a component area.

[7] See also Chapter 3 (page 50).

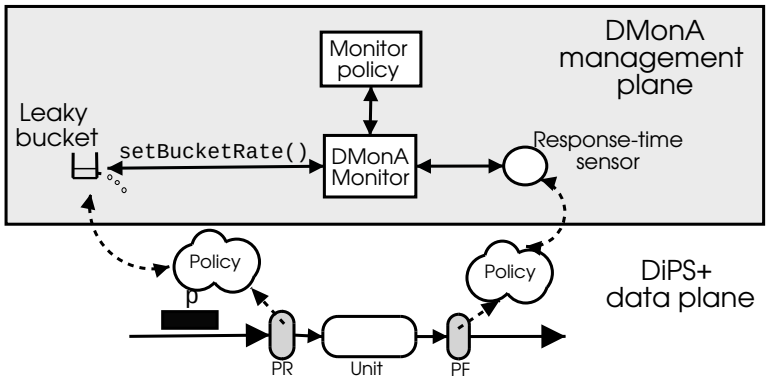



Figure 4.6: Using DMonA to control the admission of new packets in a component area. The management module associated with the packet receiver acts as a leaky bucket and drops packets when the buffer is full. The management module at the packet forwarder measures response-time, which is used to steer the packet delivery rate of the leaky bucket.

Secondly, the 90th percentile algorithm itself can be implemented as a monitor policy, which processes the collected information at regular times. In this case the algorithm will check whether 90% of the packets experience a response-time equal to or less than some pre-defined threshold t.

Thirdly, the management target is no longer the concurrency unit itself (as was the case in Section 4.2.2), but the leaky bucket that controls the admission rate of entering packets. We install the leaky bucket as a management module, associated with the packet receiver of the concurrency unit. This packet receiver is the perfect place for such control, since it represents the entry of a component area.

## 4.3 The DMonA prototype

This section points out the advantages of using the DMonA prototype to handle overload conditions, and dynamically changing bottlenecks inside the component pipeline. In addition, we demonstrate the performance benefits of automatically balancing threads in DMonA compared to both DiPS+ and the thread-per-request model.

Note that all performance results reported on in this section should be considered in relation to each other, instead of being interpreted as stand-alone performance numbers. Our objective – in this chapter and more generally in this dissertation – is to propose a software architecture, design and prototype of a

system that enables developing and managing protocol stacks. In view of that, we have developed a prototype in Java. The choice for Java as the programming language is driven by its software engineering and robustness benefits (such as support for dynamic class loading, threading and garbage collection). Although the performance of Java code is still considerably lower than C code, the Java prototype does allow us to show the advantages of our approach.

In order to demonstrate its benefits, DMonA is compared with two other thread allocation models, the first of which is the thread-per-request model. This model assigns a new thread to every incoming request, without identifying internal subtasks. Each thread guides a request until it is processed. This model is designed as a DiPS+ concurrency unit at the front end of a component pipeline, without any further concurrency units inside the pipe. In this case, the concurrency unit spawns an extra thread whenever a request arrives.

On the other hand, DMonA is compared with the standard DiPS+ concurrency model. When the number of arriving requests exceeds the number of threads available in the concurrency unit, additional requests are queued or rejected. DiPS+ allows dynamically assigning packet handlers to different concurrency units, but only under human control. The DiPS+ model is comparable to a bounded thread pool model with thread pools distributed over the component pipeline.

For both the DiPS+ and DMonA model, the component pipe has been split into independent component areas, by inserting a concurrency unit in front of each area. In the case illustrated in Figure 4.5, the test setup consists of three DiPS+ component areas, each with different and varying processing latency. The DiPS+ setup assigns a static number of threads to each concurrency unit. The DMonA setup for its part manages each concurrency unit by the monitor policy described in Section 4.2.2.

### 4.3.1 Handling peak loads

In order to demonstrate the importance of a well-defined concurrency control mechanism, let us first have a look at how the three models under test behave in case of overload. To do so, we incrementally increase the number of incoming requests and measure the system's throughput, i.e. the number of finished requests per time unit.

Figure 4.7 combines the throughput graph in Figure 4.3 with the results DMonA. From 300 parallel requests on, both the DiPS+ and the DMonA model reach a much higher level of throughput, which remains constant as the number of parallel requests further increases. Irrespectively of the load, the DMonA model outperforms the DiPS+ model.

Multiple conclusions can be drawn from these results. First of all, assigning a thread to each incoming request is clearly beneficial when dealing with a limited number of parallel requests. In this case, each thread can be allocated relatively much processing-time compared to the thread-switch-time, while the waiting-time

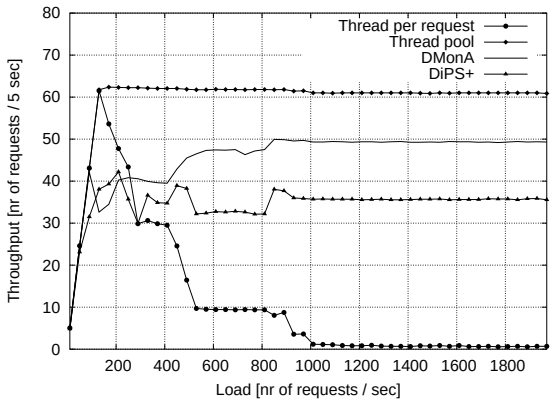



Figure 4.7: Comparison of four thread assignment methods handling increasing processing load. The component pipeline consists of three components, with increasing load. This creates a bottleneck at the end of the pipeline for the DiPS+ and the DMonA model, which insert a concurrency unit in front of each component (resulting in three component areas). The thread-per-request model is performing better until 300 concurrent requests. As load increases, throughput quickly drops below that of the bounded thread pool, the DiPS+ and the DMonA model. By balancing the processing resources more optimally, DMonA clearly outperforms the DiPS+ model.

for each request is limited. The more threads are active, however, the more significant thread-switching becomes and the longer it will take for each request to be processed. This explains the sudden drop of throughput for a load higher than 300 requests. Since the DiPS+ and DMonA models set an upper bound to the number of parallel threads, they do not suffer from this effect.

The second conclusion is that a well-chosen thread distribution across component areas heavily influences overall system throughput. Indeed, due to the initial non-optimal thread allocation to the three component areas, the DMonA model performs approximately 30% better than DiPS+. Although DiPS+ enables a human administrator to change the thread allocation when such non-optimal situation occurs, detecting bottlenecks is far from trivial since peak loads are often very short-term and unpredictable. Moreover, finding a better global thread distribution becomes more difficult as the number of concurrency units increases.

### 4.3.2 Handling bottlenecks

In order to fully exploit the power of DMonA, it is applied in a dynamic context with respect to processing bottlenecks. Again, the three thread assigning models are compared to show DMonA's performance benefit. More precisely, the goal is to

| Intervals (sec) | Load area A | Load area B | Load area C |
|---|---|---|---|
| **Phase 1 (0 - 399)** | 100 | 700 | 100 |
| **Phase 2 (400 - 799)** | 200 | 200 | 600 |
| **Phase 3 (800 - 1199)** | 100 | 700 | 100 |

Table 4.1: Dynamically changing the load in three component areas. The interval between two changes of load is 400 seconds.

demonstrate how DMonA reacts to changing load conditions. Varying bottlenecks may occur, for instance, when a burst of packets that need decryption alternates with a burst of non-encrypted packets (not needing decryption). This may result in the decryption component area being alternately overloaded and underloaded. Figure 3.2 (page 42) shows a component pipeline with two parallel paths, for example, for encrypted and non-encrypted packets.

Table 4.1 describes the variable area load test scenario. The scenario identifies three phases of 400 seconds, with the load of each area changing at the start of each phase. During the first phase, area B is loaded more heavily than areas A and C. Phase two shifts the bottleneck towards area C, which thus becomes loaded much more heavily than A and B. Finally, during phase three, area B is again overloaded compared to the other two areas.

Figure 4.8 shows the throughput results of the three tested models. First of all, DMonA throughput outperforms the other models throughout all phases. Moreover, no relevant difference in throughput has been measured between the three phases. This contrasts with the DiPS+ throughput, which is affected by the load variation across the three phases. Throughput slightly increases during phase two, but drops again after the major burst at the start of phase three. This can be explained as follows. During phase two, requests are queued in the concurrency unit in front of area C, since this area is heavily loaded and creates a bottleneck. At the start of phase three, processing load in area C suddenly drops, and since area B becomes overloaded again, new packets arrive relatively slowly at the concurrency unit in front of area C. As a result, all buffered requests can be released quickly, which causes a major burst. However, once the queued requests have been processed, throughput drops again below DMonA performance. Finally, the throughput result of the thread-per-request model is again very low because of the high number of parallel requests.

Another DMonA effect shown in Figure 4.8 is the throughput deviation from the average of ten requests per second. This “jojo” effect is mainly caused by the narrow decision margins for adding or removing a packet handler, which results in continuously migrating threads from one concurrency unit to another when this is not absolutely necessary. Although the DMonA monitor policy used may not be optimal, it allows demonstrating DMonA performance benefits in the presence of

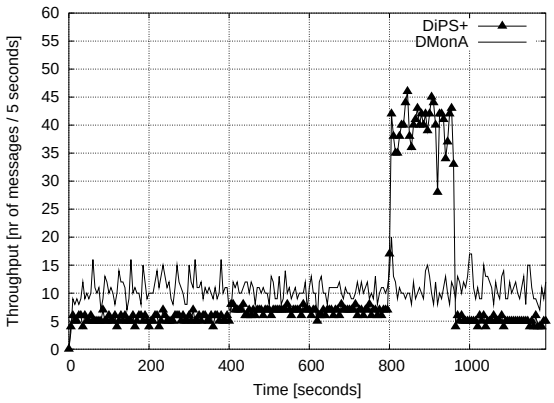



Figure 4.8: Measured throughput in three areas with changing load. The DiPS+ model suffers from unbalanced thread distribution. The DMonA model clearly outperforms the other two, although throughput varies due to a non-stable thread distribution.

highly variable load conditions.[8]

Figure 4.9 zooms in on the DMonA model and shows how it migrates nine available packet handlers when load conditions change along the three phases of the test scenario. Thread assignment for the concurrency unit in front of area A remains constant at one packet handler, since the load for this area is nearly constant as well. The peaks in the graph reflect the competition for an extra packet handler, and the deliberate disturbances of the policy in use. The graphs for area B and C demonstrate more clearly how the monitor policy follows the load conditions. During the first phase, five threads are assigned to the more heavily loaded area B, while the three remaining threads are assigned to area C. When the bottleneck shifts to area C in the second phase, two threads are moved from area B to C. The original thread distribution is restored in phase three, since load conditions return to normal.

The overall conclusion that can be drawn from these self-healing test results is that a well-chosen thread distribution policy effectively keeps throughput at a constant level during changing load conditions. Moreover, DMonA throughput is higher than that of the thread-per-request and the DiPS+ model.

[8]Note that the deliberate disturbances of balanced thread distributions carried out by the adaptability policy also contribute to the "jojo" effect, yet this effect occurs at regular time intervals instead of continuously.

## 4.4 Conclusion

This chapter has presented the DMonA management framework and demonstrated how it extends DiPS+ to enable automatic self-management and monitoring. Thanks to DiPS+ abstractions such as concurrency units and explicit component entry and exit points, DMonA could be developed as an orthogonal DiPS+ extension that can be added and removed when needed without affecting DiPS+ units at the data plane. Comparison of load management by the DMonA prototype with the DiPS+ and thread-per-request thread assignment models has shown that DMonA can gracefully handle peak loads and variable internal load conditions.

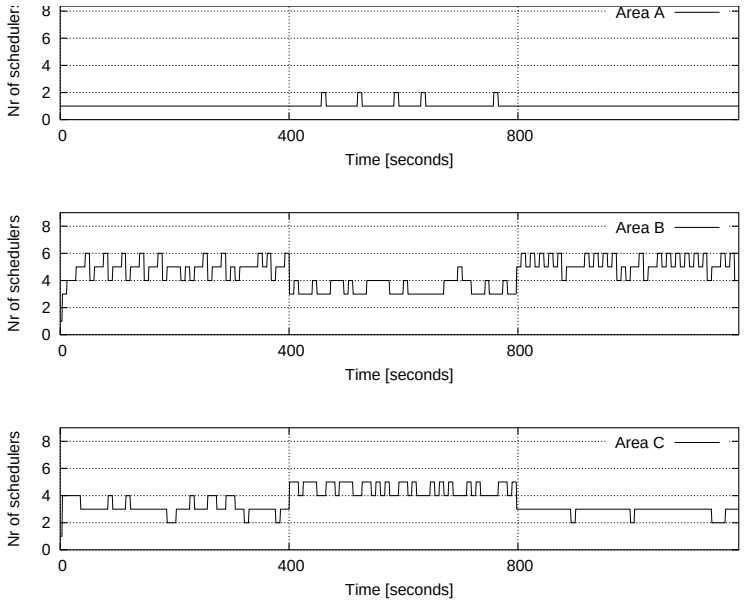



Figure 4.9: Dynamic thread re-allocation over three component areas.

*"The question of whether computers can think is like the question of whether submarines can swim."*

*– From Edsger Wybe Dijkstra.*

# Chapter 5

# DiPS+ testing

During the past two decades, many methods and tools have been developed to solve testing problems of object-oriented software [Bin99]. However, component-oriented software development for its part poses some new testing challenges. The quality of component-based software products depends not only on the quality of individual components, but also on the effectiveness of test infrastructure. If we are to deploy software components in diverse projects, products and environments, new methods for testing are needed in order to make them highly reliable and reusable [Wey98].

Moreover, the complexity of protocol stacks emphasizes and confirms testing to be a difficult, tedious and time-consuming task. The basic problem is that, for performance reasons, the protocol stack is often designed as a monolithic and multi-threaded block of code without well-separated functional module boundaries. This prevents such software from being tested properly for two reasons. First, it is very difficult to isolate the basic building blocks as stand-alone components. Even if these components were designed as separate entities, they are not necessarily independent (stand-alone): any component can create references to other components making them dependent on each other. Secondly, protocol stack software typically involves concurrency (or parallelism). Since test scenarios are impossible to reproduce in a concurrent context, testing is complicated considerably. To make matters worse, concurrency code is often spread out over the functional source code [KLM+97], which prevents multi-threading from being controlled in an effective way. Ideally, testing focuses first on the functional code only, apart from concurrency. Later, when source code has become robust, concurrency code can be added incrementally. Existing tests should be re-executed and supplemented with new cases that test the software when executed in parallel.

This chapter serves two purposes: demonstrating that the DiPS+ component design leads to "testable" component software and proposing a test infrastructure that supports DiPS+ developers in writing and executing test cases. Testable component software is designed to support four crucial testing aspects: isolation testing, uniformity, incremental testing and test-environment control. In addition, it limits testing overhead by restricting and enforcing a component design. This means that the DiPS+ component framework, which enforces the component design, helps to develop software that is open to testing. As a result, component developers can write and execute tests more easily, which avoids unnecessary waste of development-time.

In view of that, we propose DiPSUnit, a lightweight test infrastructure for DiPS+ components that generalizes test setup, execution and tear down [MWJV, MWJV02a, MWJV02b]. Fine-grained components as well as coarse-grained protocol layers and stacks can be tested in a uniform way. DiPSUnit exploits the restrictions enforced by the DiPS+ component framework design. The goal is to encourage software developers to effectively test their software by enforcing design rules that are crucial to component and system testability, and by offering them an infrastructure that facilitates testing.

## 5.1 Testing issues from a DiPS+ perspective

"Testability" is an important concept in software design and testing. Programs with good testability simplify test operations and reduce the cost of testing. As discussed in [Fre91], software testability can be viewed from two angles: controllability and observability of a component in isolation. In addition, the development process itself must be open to testing by offering a consistent uniform test approach, allowing incremental and iterative test execution, and finally enabling full control of the environment of the system under test.

The DiPS+ architecture and framework design offer several benefits as far as testing is concerned. When presenting DiPS+ from a testing perspective, relevant characteristics include DiPS+ packet objects, a separate configuration description, connection control, anonymous communication, separate unit interfaces, fine-grained functional units, concurrency units and external control signals.

This section puts generic software testability issues into a DiPS+ perspective. More specifically, it presents how the DiPS+ component framework supports each of the following testability aspects: component controllability, component observability, a uniform test approach, incremental testing, and environment control.

### 5.1.1 Component controllability

The controllability of a software component (or program) indicates how easily its input and output, operations and behaviors can be controlled. By consequence, the

more controllable a component, the easier it is to prepare and execute a particular component test scenario.

We identify three types of control: input/output control, operations control, and behavior control. While input control allows creating and sending specific packets to a component, output control intercepts and interprets component output for comparison with a pre-defined expected result. Operations control for its part checks what a component under test is expected to do, for instance, by setting up a certain scenario of multiple input requests, creating particular packets or building specific component configurations that induce certain functionalities. Finally, component behavior is controlled by installing particular component policies that impose different behavior, or by marking an input request with meta-information that is interpreted inside the component.

Component controllability is supported in DiPS+ by its packet abstraction, meta-information, and its architecture description.

**Packet.** The packet abstraction in DiPS+ is crucial to controllability. All components are designed to accept, process and forward packets in a pre-defined manner, and packets are received via a fixed and uniform component interface. This consistency makes it easier to control component input and output.

**Meta-information.** By attaching meta-information to a packet, a component's behavior can be influenced easily. Meta-information is, for instance, used by DiPS+ dispatching points to decide on the path a packet follows throughout the system.

**Architecture description.** Component configurations are described separately from the code, which allows to control or fine-tune the tested functionality by instantiating different configurations. This can be done independently from test packets.

### 5.1.2 Component observability

Isolation testing observes a piece of software apart from other cooperating entities and its physical execution environment. It concentrates on a single entity (fine-grained or coarse-grained) by manipulating or simulating all external influences.

Isolating a DiPS+ component requires control of three entities: the connection to other components in the system, control events, and meta-information.

**Connection control.** DiPS+ controls and enforces how components interact. All component interaction passes via packet receivers and forwarders, which are DiPS+ component framework entities. As a result, units have no direct knowledge of each other and only interact via the narrow and fixed unit interface (`incomingPacket(p)`). Since component interaction is interceptable, a test infrastructure can replace the DiPS+ interaction mechanism by feeding packets at the entry point of a component and catching them at the exit.

**Control events.** DiPS+ components may emit control signals. Next to meta-information, these signals are a way for components to communicate anonymously

with their environment or other components. When a component is tested in isolation, it is obvious that such control events must be intercepted. Again, a test framework may virtualize the real target component and simulate its expected behavior or an error, in case of testing an error condition. Since components are very fine-grained, such virtualized target components (often called stubs [Bin99]) can be kept very simple.

**Meta-information.** Components can pass meta-information to each other by attaching it to a packet object. This anonymous way of communicating keeps them independent of each other. Moreover, test packets can easily be prepared – by attaching specific meta-information to them – and verified afterwards – by checking whether specific information has been added.

### 5.1.3 A uniform test approach

There are two dimensions to uniformity. First of all, uniformity across different types of components means that a single test infrastructure can be used for all software produced (from fine-grained components to layers and complete protocol stacks). Secondly, uniformity throughout the development process means that the same approach is used for unit, integration and system testing.

Three design decisions in DiPS+ are crucial in this context: the uniform design of all kinds of units (functional, dispatching and concurrency units); the fixed and narrow packet interface offered by these units; and the separation between the management and the packet interface.

**Uniform unit design.** A DiPS+ component can have multiple entry and exit points. For example, a basic DiPS+ component has one entry and one exit point only. A DiPS+ layer holds two entry points (one for downstream packets and one for upstream) as well as two exit points. A dispatching component has one entry point and two or more exits.

Having a uniform DiPS+ design for all unit types considerably facilitates test generalization in a test framework. In particular, it allows offering a generic and automated approach to link a component under test with the test framework. Experience has learned that plugging in by hand a component with two entries and two exits is already a great burden to the test developer.

**Fixed packet interface.** DiPS+ units offer a fixed and narrow packet interface (`incomingPacket(Packet p)`). Since this interface is the same for all components, executing a test scenario can easily be generalized into a test framework. As a result, writing a test case[1] becomes as easy as providing the tested unit with specific test packets and checking the resulting packet(s) leaving it.

**Separate management interface.** DiPS+ units expose one or more interfaces. Each of these concentrates on one specific concern. For instance, the

[1] The term test case is introduced in Chapter 2 (page 30).

packet interface serves packet forwarding through the component pipe, whereas the management interface of a concurrency unit allows controlling its behavior.

Concentrating on a single concern makes testing more focused since the scope is well-defined. Moreover, by separating the generic packet interface from the specific management interface, testing the packet flow only involves passing DiPS+ packets to a very simple unit interface.

### 5.1.4 Incremental testing

Ideally, testing should be a step-by-step (incremental) process [Amb02, Bin99]. In an extremely incremental software development process, test code development starts together with functional code development [Bec99]. Moreover, all tests should be executed repeatedly to make sure no code is broken when adding new code. Yet, the incremental testing approach should also be applied to the source code itself (cfr. separation of concerns [KLM+97]). In other words, testing should concentrate first on the functional code, and only when this has become robust, concurrency code should be added incrementally. At this point, all existing test scenarios are re-executed, and extended with specific concurrency tests.

Testing concurrent systems is often very complicated for two reasons. The first one is the lack of concurrency control that is available to the (test) developer. In a test case it is interesting, for example, to limit concurrency to one thread at a time instead of testing the complete concurrent system at once. Secondly, multi-threading code is often scattered across the code (Kiczales e.a. [KLM+97] say concurrency code *cross-cuts* the functional code). Languages like Java make threads so easy to use that programmers do not always think thoroughly about the reasons for and consequences of introducing them. It is obvious that inattentive or abundant use of threads complicates testing, maintaining, debugging and even understanding the code.

DiPS+ supports such incremental test approach, both concerning the development process and separation of concerns in the source code, by offering fine-grained units and separating active behavior in concurrency units.

**Fine-grained units.** Each DiPS+ unit is very fine-grained and provides a well-defined functional behavior. Fine-grained units make unit specifications simple and clear. The more coarse-grained the component, the more complex its specification will be and the more difficult it becomes to test it. Yet, since a composed component (e.g. a protocol layer) is built from fine-grained units, it can be tested incrementally, i.e. testing individual units first, followed by integration and system testing.

**Concurrency units.** Since concurrency in DiPS+ is well-separated from functionality, testing can focus on functionality first. Subsequently, concurrency units can be added and the system can be re-tested. Again, this incremental approach facilitates testing considerably since problems are tackled step-by-step and with a solid basis to start from.

DiPS+ allows changing the concurrency model, say from a single-threaded to a multi-threaded model. This in turn allows to test components in a single-threaded context first (just to test component cooperation). Later, when these sequential (single-threaded) tests succeed, attention can shift to the asynchronous (multi-threaded) behavior of components.

### 5.1.5 Environment control

Software, and especially protocol stacks, cannot be tested effectively without taking into account the environment. In case of protocol stacks, the environment is formed by the network. Consequently, it should be possible to simulate network conditions, both normal and erroneous. Moreover, as in every component-based system, all relevant component configurations must be easily testable.

Testing a protocol stack in a real-life network is difficult, because the network behavior cannot be sufficiently controlled. It would be very hard, for instance, to test the behavior of a protocol stack in the presence of typical network conditions such as packet delays, packet loss, or packet duplication. Such error conditions should rather be tested in a simulated environment that enables producing network errors in a controlled fashion.

Unfortunately, simulating the network at wire-level is far from sufficient to test protocol layer functionality in detail. Such tests should be able to interpret protocol-specific information. To test the correctness of a connection-oriented transport protocol like TCP, for instance, we must be able to influence specific control packets for establishing (and closing) a connection. A three-way handshake protocol is often used to guarantee unambiguous connection setup and tear down in the presence of data loss, duplication or delay [Ste00]. To setup a TCP connection, for instance, a "SYN" packet is sent to the receiver, which replies with a "SYN/ACK" packet. This is again acknowledged with an "ACK" packet. Consequently, if we want to test the correctness of TCP connection establishment in the context of packet loss, we should delete one (or more) of the control packets involved in the handshake (SYN, SYN/ACK or ACK).

Yet, accessing protocol-specific information may be impossible at the bottom of the protocol stack, since this information is encapsulated in protocol headers, which should not be accessible outside the scope of their protocol layer. One option to found out specific information about a packet is to parse the protocol headers again. Obviously, this is not an ideal solution. Another option is to inspect the packet's meta-information at the time it leaves the protocol stack. Yet, it is not sure that all meta-information is still available at that time. Moreover, it would be more efficient to apply specific operations immediately, instead of postponing them until the packet leaves the protocol stack.

Therefore, if particular packets could be filtered and manipulated when all relevant meta-information is still present, i.e. inside the protocol layer itself, it

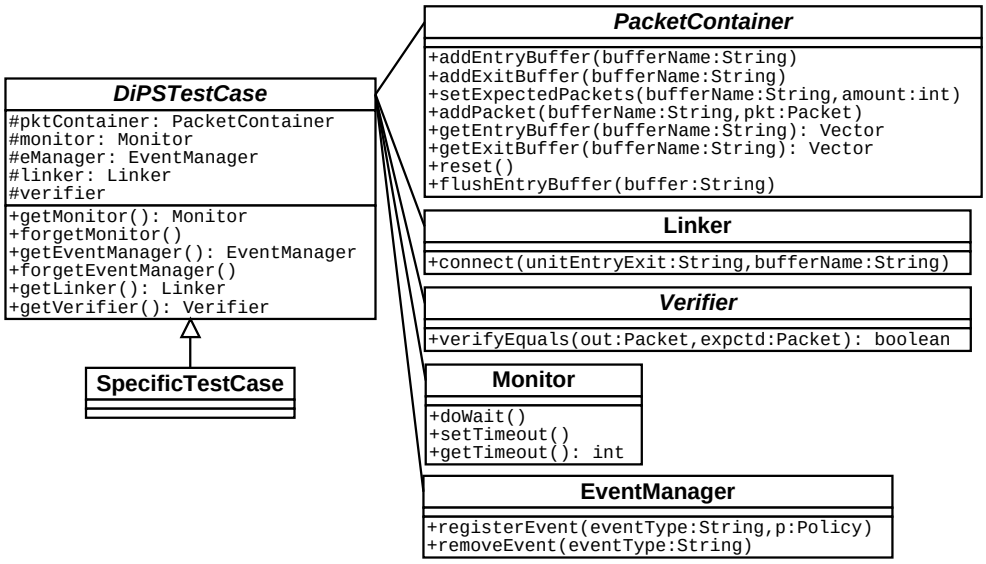



Figure 5.1: The design of the DiPSUnit test framework.

would allow identifying the packets to be manipulated in a very straightforward manner.

DiPS+ supports this by its architecture description and component design, which both allow to extend a protocol stack configuration with test components. **Separate architecture description.** Since the system's architecture description is separated from the source code, different configurations can be created without having to change the code. In this way, specific test configurations can simulate the environment by adding test components. Such components may, for instance, introduce errors, impose exceptional environment conditions the tested system must survive, or simply virtualize some irrelevant part of the environment.

**Independent components.** DiPS+ components are freely reconfigurable thanks to two characteristics. First, connection control in the DiPS+ framework prevents components from having direct references to each other, a condition sine qua non for (dynamic) component reconfiguration. Secondly, meta-information allows components to communicate with each other in an anonymous way and thus prevents again that components have to "know" each other. To some extent, however, component reliance on available meta-information restricts possible configurations, since meta-information producers must precede the tested component.

## 5.2 The DiPSUnit test framework

Having sketched the DiPS+ characteristics relevant to testing, this section demonstrates how the DiPS+ design can be exploited to build a DiPS+ test framework (DiPSUnit) that is powerful enough to test protocol stacks without complicating the development of test cases. DiPSUnit provides a generic infrastructure for unit, integration and system testing of DiPS+ software. The core framework is presented first, after which we zoom in on its support for testing asynchronous systems, event control and environment control.

### 5.2.1 Uniform test infrastructure

This section proposes the core design of the DiPSUnit test framework (see also Figure 5.1), and describes how it supports unit, integration and system testing. DiPSUnit is applied first to components with exactly one packet receiver (entry) and one packet forwarder (exit). In a second step, the DiPSUnit approach is generalized to support components with multiple entries and exits, such as layers. The section closes by showing the generic test execution template that is driven by DiPSUnit and summarizing the advantages of using DiPSUnit.

#### Testing components

From a unit testing perspective, a component under test can be a single DiPS+ functional component with one packet receiver and one packet forwarder. Yet, from an integration testing point of view, the component under test may as well be an entire pipeline of DiPS+ components, as long as it has exactly one entry and one exit point.

Setting up a DiPSUnit test actually comes down to linking the tested component into DiPSUnit. When executing a test, one or more packets are sent through the tested component and output packets are collected. Central to test execution is obviously verification whether the output packets comply with the expected result(s). The most important abstractions for setting up and executing a test are modeled in DiPSUnit by three classes, `PacketContainer`, `Linker` and `Verifier`.

First of all, DiPSUnit provides a so-called `PacketContainer` class that manages packet buffers. A packet container is initialized by creating a packet buffer for each entry and exit point of the tested component. As such, it is designed to support components with more than one entry and/or exit point as well. A packet container's entry buffer is used to store specific test packets that are created at test execution-time. An exit buffer for its part collects result packets that are received from the tested component during a test run. Figure 5.2 shows how the DiPSUnit framework is linked to a DiPS+ component. Note that the packet container is shown as two queues instead of one logical entity.

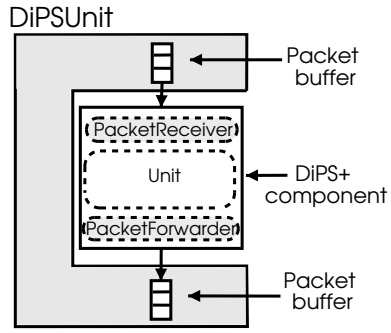



Figure 5.2: The DiPSUnit test framework connected to a component with one packet receiver and one packet forwarder.

The packet container in DiPSUnit allows executing test cases with one or more input and output packets. The simplest test case sends one input packet to the entry of the tested component and expects one packet at the exit point. However, all other combinations – one-to-many, many-to-one or many-to-many – are possible too. Such test cases are relevant when testing packet fragmentation and reassembly, demultiplexing one packet stream into different sub-streams or multiplexing different streams into one.

The packet container allows specifying the number of packets to be expected at each exit buffer. This is relevant since DiPS+ components may introduce new packets (such as a fragmenter component) or remove packets (such as a reassembly component or a header constructor that drops packets with erroneous headers). For that reason, one cannot assume the number of packets at the exit(s) to be equal to the number of packets supplied at the input(s).

The second major DiPSUnit abstraction is the `Linker` class, which allows linking any component to the DiPSUnit framework. This comes down to connecting the entry buffer in the packet container to the component's entry point and the exit buffer to the exit point. Connecting the exit buffer involves introducing a specific packet receiver that puts incoming packets from the component's exit point into the exit buffer. This is taken care of by the test framework and remains completely transparent to the test developer.

Finally, DiPSUnit provides a generic `Verifier` class for test assertion, i.e. for checking whether the actual results match the expectations. Examples include testing that two DiPS+ packets are equal, comparing raw data arrays, checking the number of result packets, or verifying whether some condition is true.

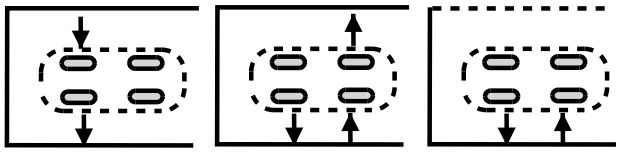


Figure 5.3: Some intuitive examples of layer test case scenarios. On the left, a normal packet is sent through the layer. In the middle, a packet is received from the network, forwarded through the bottom exit point, and (virtually) delivered to the local stack through the upper exit point. The right most scenario is the same, but limited to packet forwarding. The upper entry and exit point are not used in this case.

### Testing layers and protocol stacks

In essence, the difference between testing layers and components is twofold. First, layers have more than one entry and exit point. Most protocol layers have two entries and two exits. Secondly, input test packets and output result packets can be spread out over the entry and exit points of the layer under test. Both differences between testing a single component and testing a layer are supported in DiPSUnit by the `PacketContainer` and the `Linker` class.

Figure 5.3 shows three layer test scenarios using specific entry and exit points. The right most scenario illustrates packet forwarding in a network layer. Packets arrive at the bottom entry point and leave the layer again at the bottom exit. The upper entry and exit point are not used in this case.

The packet container's generic design allows to deal with more than one entry or exit buffer. Specific packet buffers can be flushed by a test case. Moreover, a test case may influence the order in which test packets are delivered to the tested component. One example of such non-trivial packet forwarding policy alternates between entry buffers to simulate send-receive scenarios such as a three-way handshake protocol [Ste00]. In this case, the test case and DiPSUnit play the role of sender, while the component under test acts as receiver.

The linker connects all entries and exits to the appropriate packet buffer, similar to linking a DiPS+ component. Following the strategy design pattern [GHJV95], the DiPSUnit framework differentiates between linkers based on the type of entity under test. In this way, the difference between linking specific entities is hidden from the test programmer.

Figure 5.4 shows the DiPSUnit framework and a set of DiPS+ components grouped into a layer. For clarity reasons, the packet container is shown as four buffers instead of one logical entity, while component details such as entry and exit points have been removed.

Basically, there is no significant difference between testing a single layer and testing a (protocol) stack of layers. Like for a layer, packets can enter and exit the

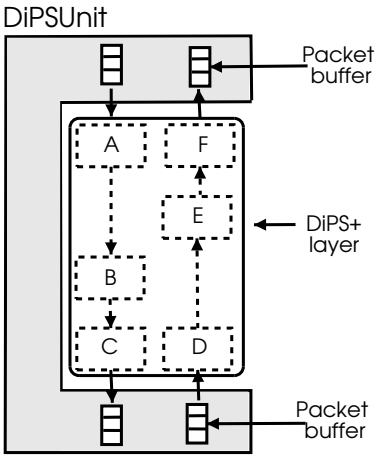



Figure 5.4: The DiPSUnit test framework connected to a DiPS+ layer (two packet receivers, two packet forwarders).

protocol stack at different places. To simulate send/receive scenarios, typically a loopback layer is used that forwards down going packets immediately to the up going path of the stack. The DiPSUnit packet buffers at the bottom are not used in this case.

When testing a protocol stack system, the top branch of DiPSUnit virtualizes the socket on top of a protocol stack, while the bottom branch virtualizes the network device driver below the stack. It accepts down going packets from the protocol stack, and delivers up going packets to the bottom of the stack. Having control over the top and bottom of the protocol stack allows testing any application-specific scenario or any packets that arrive from the network.

#### Uniform test scenario

DiPSUnit treats different granularities of testing consistently and hides component differences from the test developer. To illustrate how a test case is developed in DiPSUnit, the generic test template is summarized here.

Unit testing is typically a three step process involving a setup phase, a test and verification phase, and a tear down phase [BG98]. A DiPS+ component test case translates these steps into the following scenario. The setup phase creates and initializes a packet container by creating a packet buffer for each entry and exit of the tested component. The number of expected packets is then set for each exit packet buffer. The component is linked with the DiPSUnit framework by connecting each component entry and exit point to an entry and exit buffer respectively. This part of the setup process is test case-specific. Secondly, to run

a test case, specific test packets are created and added to specific entry buffers in the packet container. Subsequently, the test case is started, i.e. the packets in the entry buffer(s) are delivered to the component. Upon completion, the test is followed by comparing the packet(s) in the packet container's exit buffer(s) to the expected result packet(s). Finally, in the tear down phase, test packets are disposed of by resetting the packet container.

The three step test process is summarized below. We will refer to this template later when describing how DiPSUnit supports asynchronous behavior and event control.

1. *setup phase* :
   (a) create and initialize a packet container
   (b) specify the number of expected packets at each exit
   (c) link the component under test with the framework
2. *test and verification phase* :
   (a) fill specific buffers in the packet container with prepared test packets
   (b) deliver the packets to the component for processing
   (c) check assertions on the output packet(s)
3. *tear down phase* : clear the packet container

**Advantages**

A first advantage of using DiPSUnit lies in its uniform test approach for both basic and composed components. It makes no difference for the framework whether a single component or an entire sequence of components is tested, since the whole is treated as one black-box component anyway.

A second advantage is that production components and their property descriptions can be used for testing without any modifications. Reusing production source code prevents the introduction of errors when migrating the code under test to a production release. Such errors may have to do with extending the interface to enable code inspection, or changing an object invocation into a stub call. Property descriptions can be reused to test the normal configuration of a component, layer or protocol stack. Yet, they can also be extended to create specific test situations, for instance by inserting manipulator components in between layers.

Finally, components and layers can be tested in isolation on two conditions: there should be neither active behavior inside nor emission of control events. The former condition relates to one or more concurrency units being present, as supporting asynchronous behavior requires extra control. The latter is concerned with events emitted by a component or layer to an external entity. In this case, a stub

is needed to capture and react to this control event. The remainder of this section describes how to deal with these two conditions in order to test any component or protocol layer in isolation.

### 5.2.2 Dealing with asynchronous behavior

The basic issue for testing is that a call to a composed component with a concurrency unit inside returns before all processing is finished. By consequence, we need a mechanism that allows for the test to be *suspended* until all packets have arrived at the component's exit(s) (i.e. until they are all available in the packet container). Moreover, this mechanism should avoid a test to be suspended infinitely in case packets get lost or merged together. A header parser, for instance, may reject an incoming packet because its header is corrupted, or a reassembler component may convert N fragments into a single packet. For that reason, it must be possible to specify a time-out value after which the test resumes.

#### Design

The DiPSUnit test framework offers a `Monitor` class. The monitor blocks a test case for an asynchronous component after all input packets have been delivered to the component under test. Only when all result packets have arrived in the exit buffers of DiPSUnit, or when a pre-set time-out value has expired, the monitor sends a wake-up signal to unblock the test case. Therefore, the packet container and the monitor are coupled to allow the latter to inspect the former in order to know when all result packets have been generated.

Internally, the DiPSUnit monitor functions as follows. When the monitor object is called by a test case (see test case scenario, step 2c), it creates a specific packet receiver for each exit point of the component. This packet receiver does not only put incoming packets into an exit buffer, it also signals the monitor when a packet is received. When the total number of expected result packets (that can be retrieved from the container) equals the total number of signals received by the monitor, the suspended test is resumed (unless in the meantime a time-out has occurred).

Figure 5.5 illustrates the DiPSUnit design extended with the monitor. It shows a down going path with an active (concurrency) unit. The monitor controls the down going path by blocking the test until all expected packets have arrived at the lower exit point. The test case scenario for concurrent data flows is very similar to the basic test scenario. The only difference is that the monitor must be initialized during test case setup (step 1b), called during execution (step 2c), and disposed of during tear down (step 3b).

1. *setup phase* :

   (a) create and initialize a packet container

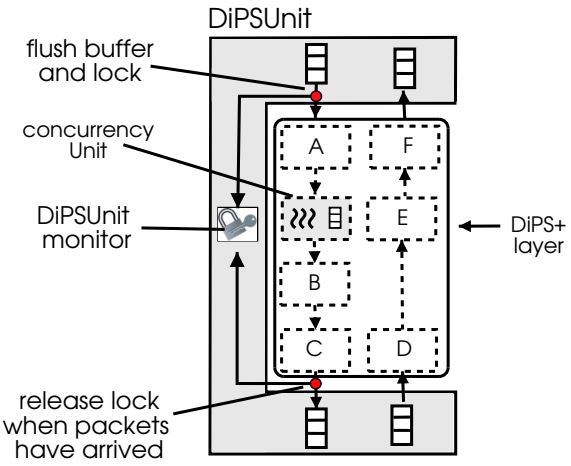



Figure 5.5: The DiPSUnit test framework with monitor support connected to a DiPS+ layer with a concurrency component.

(b) *initialize the monitor*

(c) specify the number of expected packets at each exit

(d) link the component under test with the framework

2. *test and verification phase* :

(a) fill specific buffers in the packet container with prepared test packets

(b) deliver the packets to the component for processing

(c) *ask the monitor to suspend the test until time-out expires or the number of expected packets has been reached*

(d) check assertions on the output packet(s)

3. *tear down phase* :

(a) reset the packet container

(b) *dispose of the monitor*

### 5.2.3 Event control

The DiPS+ design controls how components can interact and prevents them from having direct references to each other. In addition to the packet flow through the `incomingPacket(p)` interface, components can send control events to each other. By consequence, extra support is needed to catch control events and simulate

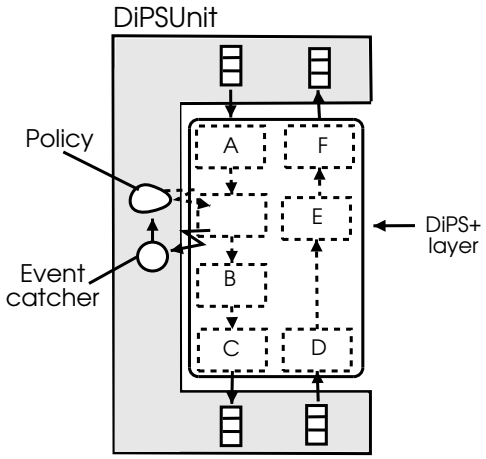



Figure 5.6: The DiPSUnit test framework connected to a DiPS+ layer with an event throwing component.

the reaction to them. To this end, every control event is caught by a dummy component acting as a stub.

DiPSUnit offers infrastructure support to deal uniformly with external control events. The test developer only specifies what must happen in case a certain control event is received. Installing the stubs and connecting them to the tested component is taken care of by DiPSUnit itself.

To facilitate describing how to respond to a given control event, a test developer can use the manipulator `Policy` class. Similar to a stub, such policy describes how a control event must be handled by simulating the actual functionality or an error. Examples are not sending a reply event although it is expected, delaying the reply or deliberately producing a wrong answer. Separating policies from actual test cases makes a generic policy easily reusable. A policy that silently removes incoming triggers, for example, is generic enough to test packet loss in different cases. The same holds for a *delay* and a *duplicate* manipulator policy.

All policies have to be registered in the DiPSUnit `EventManager` class, together with the event type that triggers them. This registration is taken care of at test initialization-time by a particular test case. The event manager actually serves as the link between a test case and the DiPSUnit framework: a test case registers its policies in the event manager, while DiPSUnit looks up the associated policy for each control event that is received (as will be explained further).

In fact, control events are caught by an `EventCatcher` object. The test developer creates an event catcher for each event type that is registered in the event manager. The default event catcher offered by DiPSUnit simply delegates all control events to the event manager to look up the associated policy object. Yet,

non-trivial event catchers may be introduced as well (for example, one that collects a number of control events before calling the event manager). In DiPSUnit an event catcher is registered as a DiPS+ event listener.

Figure 5.6 illustrates how a component emits a control event that is caught by the event catcher and delegated to a specific policy. As shown, a `Policy` object can send control events back to a component if necessary.

The following scenario summarizes how DiPSUnit is used to support control events. The concurrency steps introduced in the previous section have been omitted in order to focus on control flow stubs. Notice that the extra steps for their support are again optional. When no control events are triggered within the tested component, the scenario comes down to the original version described in Section 5.2.1, thus minimizing the complexity of the test environment.

1. *setup phase* :
   (a) create and initialize a packet container
   (b) *register event catchers and associated policies*
   (c) specify the number of expected packets at each exit
   (d) link the component under test with the framework
2. *test and verification phase* :
   (a) fill specific buffers in the packet container with prepared test packets
   (b) deliver the packets to the component for processing
   (c) check assertions on the output packet(s)
3. *tear down phase* :
   (a) clear the packet container
   (b) *reset the event manager*

The combination of DiPS+ and DiPSUnit offers several advantages. The introduction of stubs is transparent to the code being tested. This reduces the risk of replacing stubs by the actual functionality during acceptance testing (when all components are put together) since no code changes have occurred. DiPS+ components (and therefore also stubs) are typically very fine-grained functional elements. This decreases the development cost of a component (or a stub) and encourages a highly iterative test process ( *"test a little, code a little"* [Bec99]). DiPSUnit can build on the event catching and throwing support provided in DiPS+, which keeps it simple, even though it provides non-trivial support to a test developer.

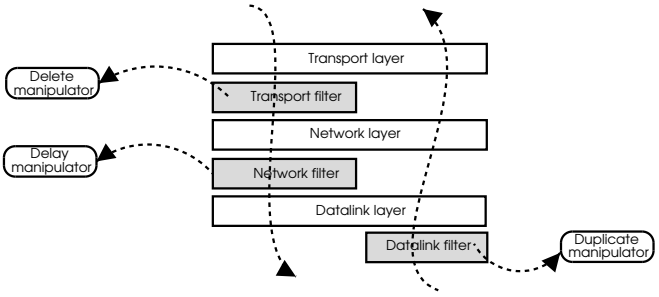



Figure 5.7: Packet filtering inside the protocol stack. The two filters in the down going flow delete and delay packets at respectively the transport and the network layer. The filter in the up going flow duplicates incoming packets at the datalink level.

### 5.2.4 Environment control

It is easier to perform protocol stack tests in a simulated environment since this provides full control over network behavior. We can introduce erroneous network behavior such as deleting or duplicating packets. In this way, we can test the basic functionality of a protocol (stack) and its behavior in special situations like error conditions, fragmentation of large packets, etc.

For that reason, we introduce filters that delegate packets to manipulator components for inspecting and influencing the packet flow. Figure 5.7 shows a protocol stack with a filter layer between the TCP and the IP layer in the down going path, and a datalink filter in the up going path.

A filter creates the link between the protocol stack and the manipulator components. It intercepts packets and delegates them through one ore more manipulators. After processing in the manipulators, the filter forwards the packet transparently to the component in the original path. Generic manipulators can be reused at different levels of the protocol stack. For example, manipulators to delete, duplicate, delay or reorder packets are protocol-independent. Protocol-specific manipulators, however, must obviously be provided for each protocol individually. An example of such a manipulator is one that removes SYN packets during TCP connection establishment.[2]

[2]Cfr. the example given in Section 5.1.5 (page 96).

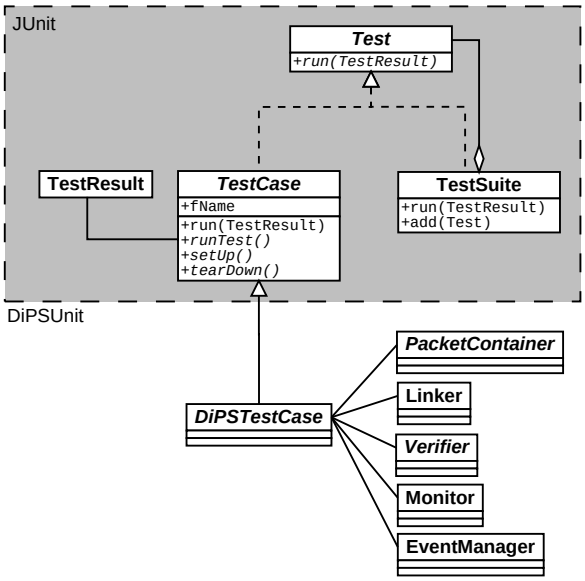



Figure 5.8: The DiPSUnit framework coupled with JUnit. The detailed DiPSUnit design can be found in Figure 5.1 on page 97.

## 5.3 Examples

To validate the DiPSUnit design, we have implemented a prototype and coupled it with an existing test tool and framework for unit testing in Java (*JUnit test framework* [BG98]) (see Figure 5.8). JUnit is a well known Java-based unit test framework that originated from the XP (Extreme Programming)-approach [Bec99]. The JUnit framework consists of a few base classes that allow developers to quickly write specific tests, group them into larger test suites, and run them either in batch or from within a simple GUI (see also the dark shaded box in Figure 5.8). For a more detailed explanation of the JUnit-framework, we refer to [BG98].

Within JUnit, everything centers around the `TestCase` class. It offers methods to initialize a test case (the setup phase), to run several test methods and to clean-up temporary objects afterwards (the tear down phase). DiPSUnit provides the `DiPSTestCase` class, which inherits from `TestCase`. The `DiPSTestCase` class in turn provides test developers with a packet container, a linker, an event manager and a monitor. Figure 5.8 shows the link between DiPSUnit and JUnit. [3]

This section explains two examples: a simple unit test case for a DiPS+ Ethernet header constructor, and an Ethernet protocol layer test case. Both examples show how DiPSUnit is used in a uniform manner, and how it supports the three

[3] A detailed view on DiPSUnit can be found in Figure 5.1 (page 97).

test phases (setup, execute and tear down) in combination with asynchronous behavior and/or event control.

### 5.3.1 A component test case

The entire `setUp()` phase of our test is defined in three steps (see also Listing 5.1). First thing to do is to initialize the packet container with the necessary buffers. As one might notice in the code below, the packet container allows to dynamically add any number of entry/exit buffers (`addEntryBuffer(...)`, `addExitBuffer(...)`). Each of these buffers can be assigned a name to enable the developer to distinguish between them. Secondly, we create the unit to be tested, in this case the `EthernetHeaderConstructor` unit. Finally, this component is linked to the framework. To this end, a request for an appropriate linker (`getLinker(testedUnit)`) is sent to the framework. This linker is used to connect specific entry/exit points of a unit with specific buffers in the packet container.

```
1  protected void setUp() throws Exception{
2
3    // create the PacketContainer and add all buffers
4    pktContainer = new PacketContainer();
5    pktContainer.addEntryBuffer("packets_to_below");
6    pktContainer.addExitBuffer("packets_from_above");
7
8    // create the unit to be tested
9    testedUnit = new EthernetHeaderConstructor();
10
11   // link the unit under test with DiPSUnit
12   linker = getLinker(testedUnit);
13   linker.connect("entry", "packets_to_below");
14   linker.connect("exit", "packets_from_above");
15 }
16
17 protected void tearDown() throws Exception {
18    // reset the packet container
19    pktContainer.reset();
20 }
```

Listing 5.1: The setup and tear down code of a DiPS+ Ethernet header constructor test case in DiPSUnit.

The `test()` phase describes the actual scenario of the executed test (see Listing 5.2). First of all, the test packets that will be delivered to the tested unit are created and initialized by setting the necessary meta-information. Subsequently, we add these test packets to an entry buffer of the `pktContainer`. Since we will not be dealing with a multi-threaded unit, all that is left to do is send the packets

```
1  public void testNormalEthernetHeaderConstruction{
2      // create a test packet
3      Packet testPkt = new Packet(.....);
4      testPkt.setMeta("src_address'', new EthernetAddress("00:06:5B:AB:DE:A9''));
5      testPkt.setMeta("dst_address'', new EthernetAddress("00:C0:4F:A0:84:B8''));
6
7      // add it to the an input buffer of the packet container
8      pktContainer.addPacket("packets_to_below", pkt);
9
10     // instantiate the expected packet
11     Packet expectedPacket = new Packet(...);
12
13     // flush the entry buffer = send packet through the unit
14     pktContainer.flushEntryBuffer("packets_to_below");
15
16     // verify the output
17     verifyEquals(getExitBuffer("packets_from_above").firstElement(),
18                  expectedPacket);
19 }
```

Listing 5.2: The code of a test case for the DiPS+ Ethernet header constructor in DiPSUnit.

through the unit by flushing the entry buffer (`flushEntryBuffer(...)`) and check some assertions on the output (`assertEquals(...)`).

The `tearDown()` phase removes all test packets from the packet container (`pktContainer.reset()` in Listing 5.1).

### 5.3.2 A protocol layer test case

This example shows that a test case for complex units does not need to become increasingly complex. *DiPSUnit* allows to deal with issues such as concurrency and control events, while still providing test developers with a uniform way of testing that keeps writing tests simple and intuitive.

We complicate our test example by embedding the header constructor into a larger unit: the Ethernet protocol layer. Like in the simple test case, all entries and exits are added to the packet container and the unit is linked to DiPSUnit (see also Figure 5.9). The extra complexity comes on the one hand from ARP request events that are emitted by the Ethernet layer. An ARP request is used to map an IP-address to an Ethernet-address. On the other hand, the Ethernet layer contains a concurrency unit to avoid being blocked while ARP is resolving an IP address.

If we compare the `setup()` code in Listing 5.3 with the simple test case of the `EthernetHeaderConstructor` unit, we see two differences: the registration

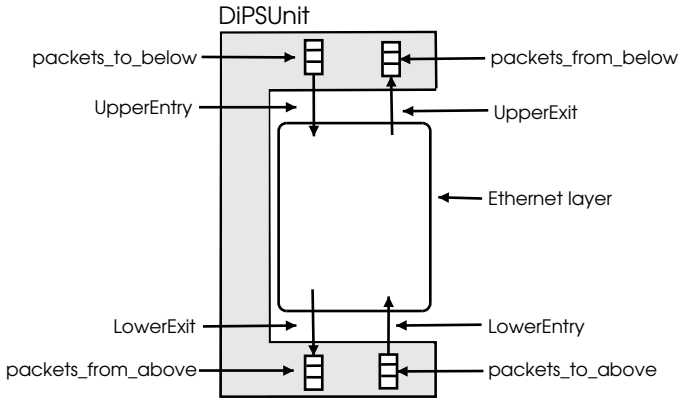



Figure 5.9: Graphical representation of the Ethernet layer test case.

of a control event and a policy at the event manager, and the request to use a monitor. Since our test case focuses on testing in isolation the functionality of the Ethernet protocol, we need to replace ARP with a stub. In this example, the `NormalARPRequestReplyPolicy` object simulates normal behavior of the ARP layer. In order to handle the asynchronous behavior of Ethernet (implied by the concurrency unit) the test case calls the DiPSUnit monitor to block test execution until all packets have been processed.

The actual `test(...)` phase, as described in Listing 5.4, is practically identical to the test method for the `EthernetHeaderConstructor` unit[4]. Data in the test packet might be different, but this is irrelevant for the example. Since the monitor needs to know the number of packets expected in each of the exit buffers, we have to call `setExpectedPackets(...)` to specify this number. By calling the `monitor.doWait()` method, the current thread (the one in which the test is running) is suspended until all expected packets have arrived or a time-out has occurred (whichever happens first).

Also the `tearDown()` phase (Listing 5.3) is similar to the one in the simple test case. The only additions are the calls to `forgetMonitor()` and `forgetEventManager()`. The former method call informs the framework that the monitor is no longer needed, the latter that there is no longer interest in the previously registered control events.

[4]Of course more complex tests can be defined.

```
1   public void setUp() throws Exception{
2
3     // create the PacketContainer and add all the buffers
4     pktContainer = new PacketContainer();
5     pktContainer.addEntryBuffer("packets_to_below");
6     pktContainer.addEntryBuffer("packets_to_above");
7     pktContainer.addExitBuffer("packets_from_above");
8     pktContainer.addExitBuffer("packets_from_below");
9
10    // create the unit to be tested
11    testedUnit = new EthernetLayer();
12
13    // register an event type and policy with the event manager
14    getEventManager().registerEvent( "ARPRequestEvent",
15                                     new NormalARPRequestReplyPolicy() );
16
17    // ask for a monitor to deal with concurrency
18    Monitor monitor = getMonitor();
19
20    // link the unit under test with DiPSUnit
21    linker = getLinker(testedUnit);
22    linker.connect("UpperEntry", "packets_to_below");
23    linker.connect("LowerExit", "packets_from_above");
24    linker.connect("LowerEntry", "packets_to_above");
25    linker.connect("UpperExit", "packets_from_below");
26  }
27
28  protected void tearDown() throws Exception {
29      // reset the packet container
30      pktContainer.reset();
31      forgetMonitor();
32      forgetEventManager();
33  }
```

Listing 5.3: The setup and tear down code of a DiPS+ Ethernet layer test case in DiPSUnit.

```
1  public void test....{
2      // create a test packet
3      Packet testPkt = new Packet(.....);
4      ...
5      // add it to the an input buffer of the packet container
6      pktContainer.addPacket("packets_to_below", pkt);
7      pktContainer.setExpectedPackets("packets_from_above", 1);
8
9      // flush the entry buffer = send packet through the unit
10     pktContainer.flushEntryBuffer("packets_to_below");
11
12     // go to sleep
13     monitor.doWait();
14
15     // verify assertions
16     verifyEquals(getExitBuffer("packets_from_above").size(), 1);
17     ...
18 }
```

Listing 5.4: The code of a test case for the DiPS+ Ethernet layer in DiPSUnit.

## 5.4 Lessons learned

### 5.4.1 Software fault detection

Obviously, using a test framework such as DiPSUnit helps discover software faults. However, there are other non-trivial conclusions that can be drawn from using DiPSUnit.

First, software errors are discovered early in the software development process. This is thanks to repeated execution of all unit, integration and system tests each time a component change is published, or a new component is added to the system. In this way, all relevant test configurations are executed over and over again during code development.

Secondly, repeatedly executing all tests allows to discover not only the error but also its cause. Executing tests after every code change keeps the difference between two source code releases small. Since all tests were executed successfully before the last software change, the last increment is likely to have caused the error – directly or indirectly.

Thirdly, the system can easily be kept consistent since DiPSUnit enables executing all tests every time something has changed. This emphasizes the incremental development (rapid-prototyping) process, inherent to component-based software development such as DiPS+.

### 5.4.2 Testing experiences

Experience shows that DiPSUnit is very easy to learn and use. Researchers as well as students are using DiPSUnit as a test tool. They all started from the generic test case template for setup, execution and tear down along with some concrete examples of test cases. Very soon they were able to develop their own test cases.

Software developers have gained more confidence in changing existing software. This is because the global test suite provides immediate feedback on the impact of a change on the rest of the system. Confidence grows during the development process, since more and more tests become available as new components (and their corresponding tests) are published.

Fortunately, programmers start to motivate each other for writing more tests. At the beginning, people were rather conservative and definitely skeptical towards the idea of writing tests. Yet surprisingly, these negative reactions soon turned into a positive motivation for writing more tests. Two reasons may explain this switch: the immediate feedback on system correctness by executing tests repeatedly, and the limited overhead of actually writing test code thanks to the uniform approach offered by the test framework and DiPS+.

DiPSUnit has proven to stand the test of reality. Our research team has used DiPSUnit to test DiPS+ protocol implementations such as TCP, UDP, IPv4, Ethernet and SIP. Tests have been executed at different levels of granularity, ranging from fine-grained components to complete layers and protocol stacks. Moreover, layers have been tested in isolation from other protocols and network conditions have been simulated by using manipulator components. Up till now, DiPSUnit has shown to be powerful enough to support these tests.

Judging from our experience, DiPSUnit stimulates an agile software development process [Aoy98, Amb02]. Agile development methodologies are highly incremental and iterative, encourage software change and place the programmer in the center of attention. One of the corner stones of an agile development approach is collective code ownership, i.e. every team member has access to all components. Having available a simple and lightweight test infrastructure that keeps testing intuitive and simple is a prerequisite. It allows errors to be detected and solved very rapidly, which consequently limits the amount of hidden software faults to a minimum. DiPSUnit fits in this picture by supporting a uniform test approach along the development process, from fine-grained component level to composed component level and even acceptance level.

## 5.5 Conclusion and future work

This section has proposed the DiPSUnit test framework for DiPS+ unit, integration and system testing. DiPSUnit offers test developers support in four ways. First, it provides a uniform test infrastructure for unit, integration and system

testing. Secondly, it provides support for testing asynchronous systems. Thirdly, it enables control event interception and simulation of the environment's reaction to these events. And finally, it allows simulating the environment by introducing manipulators at network or even at protocol layer level.

The ability to introduce manipulator components in the system is an important advantage for testing network protocols in the context of network errors. The integration of manipulator components into the system structure is much more intuitive and logical than manipulating raw packets in the network.

DiPSUnit has confirmed the potential benefits of the DiPS+ design with respect to software testing. Component connection control by the DiPS+ framework, control events and anonymous communication via meta-information enable isolation testing. The uniform design of DiPS+ units allows for a uniform test approach for unit, integration and system testing. Fine-grained DiPS+ units encourage an incremental and iterative test approach. Moreover, since concurrency units are separated, systems can be tested first without concurrency until robust enough to add and test internal concurrency. Finally, the separate DiPS+ configuration description is a considerable advantage for environment simulation.

We conclude that testing complex software systems such as protocol stacks can be controlled by enforcing a component design and architecture. A lightweight, intuitive and iterative test approach is crucial to agile software development methodologies, which impose frequent software changes by many programmers working on the same code.

### Future work

Test automation could be applied not only to test execution but also to test creation. For the moment programmers have to program test cases and test suites themselves. Ideally, they should specify test cases in a high-level property description, similar to the way component, layer and structure properties are described. Such a description could also indicate how a component must be linked to the DiPSUnit framework. The current linker then becomes a service for test code equivalent to what the DiPS+ builder is for production code.

A next step could be to describe how test packets are initialized. In this way, a test developer would no longer need to write any test code, but could simply describe what has to be tested.

*"C makes it easy to shoot yourself in the foot. C++ makes it harder, but when you do, it blows away your whole leg."*
*– From Bjarne Stroustrup.*

# Chapter 6

# DiPS+ framework optimization

This chapter puts into perspective two important, though often contradicting concerns within protocol stack design and implementation: performance and run-time adaptability. As demonstrated in Chapter 4, DMonA has proven to be very effective in preserving a constant throughput in overload conditions by dynamically adapting internal concurrency. Although run-time adaptability can improve performance effectively, it also causes overhead that cannot always be compensated by adaptive behavior.

Overhead may originate from various sources. First of all, overhead is introduced by the adaptive behavior itself. In order to adapt internal parallelism, for instance, monitoring is needed to detect throughput bottlenecks, after which processing resources must be re-distributed to solve them. There clearly is a performance price to pay for monitoring. Secondly, overhead may be introduced by the software architecture and design, which must take into account run-time adaptability. For example, one approach to allow adding monitoring to the system in specific places is to provide "hooks" where passing requests can be intercepted by a higher (meta-)level of control. This solution requires the code to be modularized into relatively fine-grained entities. It also requires concerns – such as concurrency – to be represented as well-manageable and therefore separate modules.

This overhead, however, is not compensated in "normal" load conditions, when data interception and monitoring are neither effective nor required. In addition, monitoring is typically needed only in very specific places in the system, for instance around a single component or concurrency unit. In such case, framework support for monitoring is redundant in all other places.

The challenge addressed in this chapter is to provide a best-of-both-worlds solution: having support at run-time to adapt the system where needed, while avoiding all run-time overhead wherever this support is unnecessary. We describe how DiPS+ addresses this challenge and how it manages this trade-off between performance and run-time adaptability. To do so, DiPS+ enables administrators to specify these places where run-time adaptability is not needed and, by consequence, better removed to optimize performance. Note that this research focuses on optimization of the DiPS+ framework and not on the functionality inside the units.

The chapter is structured as follows. Section 6.1 presents a DiPS+ performance analysis to measure the overhead the framework causes, after which Section 6.2 explains the limitations of optimizing compilers to solve this overhead. Section 6.3 describes our solution for optimizing DiPS+, which we validate in Section 6.4 by comparing the performance of an optimized DiPS+ stack with other implementations. Contributions presented in this chapter are summarized in Section 6.5.

## 6.1 DiPS+ performance analysis

In order to identify the main sources of DiPS+ framework overhead and determine potential performance benefits, this section analyzes the DiPS+ performance. The analysis concentrates first on the overall protocol stack level, and then zooms in to the fine-grained unit level and investigates the relevance of framework overhead with respect to unit processing-time.

### 6.1.1 System level analysis

We compare a DiPS+ and a non-DiPS+ UDP/IP protocol stack over a loopback interface, since we are only interested in packet processing measurements without network delays. The loopback interface simply sends all received packets back to the IP layer. Both protocol stacks offer exactly the same functionality. The DiPS+ version contains three layers, eight components (three header constructors and parsers for UDP, IP and loopback, a routing and an ARP component), two active concurrency units and one dispatcher. The protocol stack provides all support for run-time adaptable behavior, without the DMonA layer actually being plugged in. The non-DiPS+ version merges unit functionality of an original DiPS+ protocol stack, which results in three coarse-grained objects, one for each component area (i.e. one for down going and two for up going packets). Because packet receivers and forwarders have been removed in this version, it lacks any support for run-time adaptability. Moreover, by removing all possible DiPS+ overhead (e.g. module boundary crossing, explicit entry and exit points, indirect

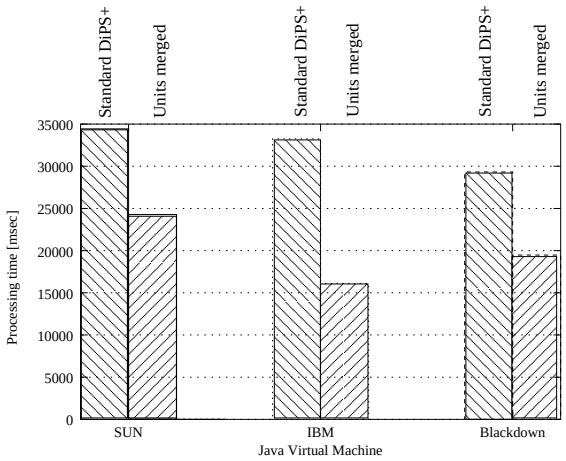



Figure 6.1: UDP/IP/loopback protocol stack performance tests for a standard DiPS+ version and for one with units merged. 100.000 packets sent, ran on an Intel Pentium 4 CPU 1.70GHz with 512 MB RAM running Linux 2.4.20-pre11. All software written in Java, compiled and executed with JVMs from SUN (1.4.0) [Sunc], IBM (1.4) [IBM03] and Blackdown (1.4.1) [Bla03]. Tests on SUN and Blackdown executed with the -server option for full hotspot optimization [Sun03b].

parameter passing through meta-information, and layer dispatching), this version can serve as the lower boundary for framework overhead.

Figure 6.1 shows performance results of these two protocol stacks executed on three Java Virtual Machines (JVMs) of Sun, IBM and Blackdown. Because each JVM uses different optimization techniques, performance results vary between them. Yet, all three show a clear performance improvement for the non-DiPS+ version, ranging from 30 % for the Sun and Blackdown versions to even 50 % for the IBM version.

These results clearly show there is a considerable potential to reduce the overhead of the DiPS+ component framework. Notice, however, that these tests take into account a specific protocol stack configuration, namely UDP/IP/loopback. This may color the results because the relevance of this overhead depends on the processing-time ratio of all units involved on the one hand, and on the DiPS+ framework code on the other hand. In other words, if relatively much time is spent inside each unit, the extra time lost in the framework becomes less relevant than in case of lightweight units.

| Configuration | Processing-time [msec] |
|---|---|
| Dummy - Simple pipe - Dummy | 14 |
| Dummy - Dispatcher - Dummy | 58 |

Table 6.1: Processing-time of the pipe connector and the dispatching component for sending 10.000 packets.

### 6.1.2 Unit level analysis

The objective of fine-grained unit level performance tests presented in this section is twofold. First of all, they enable measuring the overhead caused by the DiPS+ framework, and in particular by the module boundary crossings. A packet traveling through a DiPS+ component passes three boundaries: from the packet receiver to the unit, from the unit to the packet forwarder, and finally, from the packet forwarder to the packet receiver of the next component in the flow. In addition, these tests enable to point out the impact on throughput of adding architectural connectors (e.g. a direct simple pipe connection and a dispatching unit).

The second objective of this section is to relate framework overhead to the load of a real unit implementation in order to gain insight in the relevance of DiPS+ overhead in a realistic protocol stack configuration. Only with such insight, overhead conclusions can be drawn objectively.

Three test cases and their performance results are presented, which taken together allow for an objective conclusion about DiPS+ framework overhead and its relevance. Test configurations consist of two functional components, connected via a dispatcher. In order to obtain measurable quantities given the system clock granularity (i.e. milli seconds), we calculate the processing-time for a batch of 10.000 packets instead of for a single packet only. In addition, we repeat each measurement 100 times and keep the average as final result.

- **Connector overhead.** The first test measures connector overhead in combination with "dummy" functional units to focus purely on the framework code. A dummy unit forwards each incoming packet it receives without additional processing. A simple pipe is a conceptual connector formed by a packet receiver and forwarder that are directly connected.[1] Since such a connector does not add processing overhead besides the module boundary crossings (between PR, unit, and PF), the overhead measurements represent the lower boundary for the processing-time of two connected DiPS+ components. Similarly, the dispatching strategy used is kept as simple as possible: it allows to choose only between two straightforward conditions, and the first is always true.

[1] Cfr. Section 3.3.2 (page 48).

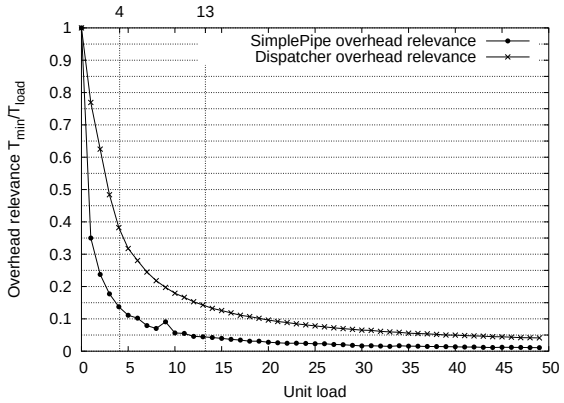



Figure 6.2: Connector overhead relevance, given "load simulator" components with increasing load. The measured load of existing unit implementations varies between 4 and 13.

The processing-time results in Table 6.1 show a clear difference between the dispatcher overhead and the simple pipe result. The dispatcher is about four times slower than the simple pipe connector.

- **Overhead relevance.** The second test case provides insight in the relevance of framework overhead in comparison with unit processing-time. We calculate overhead relevance as R = $T_{minimal}$ / $T_{measured}$(l). $T_{minimal}$ is the component execution-time for a dummy unit. $T_{measured}$(l) represents the actual measured processing-time for a given component load l. In order to obtain a continuous load scale, we introduce a special "load simulator" component that allows fine-tuning its load by setting the number of array copy operations that is executed for each incoming packet. In this way, higher load is simply simulated by more array copies.

  The graph in Figure 6.2 clearly shows that the difference in overhead between the different connectors is only relevant for very lightweight components (the left side of the figure). From a component load of 50 on, the overhead difference is no longer relevant.

- **Existing component load.** Finally, we repeat the simple pipe test in combination with existing DiPS+ components (e.g. header parsers and constructors of the UDP, IP and Ethernet protocol layers). The crossing point between the processing-time graph for a real DiPS+ component on the one hand, and the "load simulator" component load graph on the other hand gives us a good idea about the position of a DiPS+ component on our load scale. The load of real unit implementations is then placed on the relev-

| Component under test | Processing-time [msec] | Load indicator |
|---|---|---|
| EthernetHeaderConstructor | 188 | 8 |
| EthernetHeaderParser | 143 | 6 |
| LoopbackHeaderParser | 94 | 4 |
| LoopbackHeaderConstructor | 141 | 5,5 |
| IPv4HeaderConstructor | 214 | 10 |
| IPv4HeaderParser | 285 | 13 |

Table 6.2: Processing-time and load indicator of standard DiPS+ components. Time measurements are for processing 10.000 packets. The load indicator positions each component on a virtual load scale, which is also used in Table 6.1

ance graph to calculate the relevance of framework overhead with respect to component load (see the vertical bars in Figure 6.2).

Table 6.2 shows the processing-time results for a group of DiPS+ components, and associates each unit with a load indicator. The IP header parser, for instance, is the most processing-intensive component with a load of 13. In general, we can conclude that the load of most relevant DiPS+ units varies between 4 and 13. Combining the real component load results in Table 6.2 with the overhead relevance graph in Figure 6.2 shows that overhead relevance for these units lies between 5 and 15 % – for component loads between 13 and 4 – in the simple pipe case. Dispatching overhead is much higher: between 15 and 40% for the same unit loads.

Two main conclusions can be drawn from this unit level performance analysis. First of all, since DiPS+ units are very lightweight, the overhead induced by the DiPS+ framework is considerable. Yet, having such lightweight entities has major advantages for reconfigurability, reusability of code, and system maintenance. By consequence, a best-of-both-worlds solution should reduce the overhead without loosing all DiPS+ flexibility.

Secondly, dispatchers clearly form the main performance bottleneck in the framework, because a dispatcher strategy involves more processing than a simple pipe connector. Since dispatching is crucial – not only for defining different subflows inside a single layer, but also for demultiplexing between layers – it deserves extra attention from an optimization perspective.

## 6.2 Limitations of optimizing compilers

Current compilers are very powerful with respect to performance optimizations. They do not only take into account typical code optimizations such as loop unrolling, dead code elimination and function in-lining, but also focus on the run-time

status of the code. Just-in-time (JIT) compilers, for instance, translate intermediate virtual machine code into native machine code to increase performance.

Different JIT approaches exist. Continuously optimizing JIT compilers (e.g. the Java HotSpot engine [Suna, Sun03b]), for instance, focus on code executed most frequently and are able to de-optimize code when circumstances change. The latter may be the case when loading at run-time a new class that conflicts with previous optimizations. Lag*oo*na [GFF02] tries to solve some problems induced by component-oriented software and describes a system for dynamic re-compilation. This system allows for dynamic interface call optimizations, which are typically not supported by traditional JIT compilers. Such compilers assume that message dispatch through an interface is less likely than early binding [GFF02]. In component-oriented software, however, interface calls are the common case as components heavily rely on interfaces to abstract inter-component communication and increase reusability. Therefore, multiple compilations of the same code co-exist in Lag*oo*na and, depending on the actual interface call receiver, the corresponding trace is executed. Another example of continuous program optimization is presented by Kistler [KF01]. Kistler continuously optimizes the system by dynamic profiling, code optimization and component hot swapping.

However, neither of these examples helps to deal with the framework optimizations aimed at. First of all, to perform optimizations that focus on (removing) framework code, the compiler should know about framework abstractions. Yet, generic (Java) compilers cannot differentiate between functional and framework code. Secondly, even if compilers were able to do so, it would be hard to handle localized and instance-specific optimizations.

Aspect-Oriented Programming (AOP) approaches such as AspectJ [Asp03] and Hyper/J [OT01] allow programmers to write an extra program (called an aspect or a concern), which is merged into the functional code in specific places during a pre-compilation step, referred to as "weaving". In this way, very localized changes and optimizations are possible. Yet, even weavers are not sufficiently powerful to handle instance level optimizations [SGC02], i.e. optimizations that focus only on a sub-set of the running instances of a given object type. By contrast, DiPS+ optimizations may focus, for instance, on a sub-set of the concurrency unit instances in the system instead of on all of them.

Figure 6.3 illustrates two relevant shortcomings of AOP today [SGC02]. First of all, AOP cannot relate a single aspect instance to more than one object, for instance, to express a component – or object – connection relationship. Aspect instances are object-specific, i.e. an aspect instance is associated with one object at most. Unfortunately, our focus is exactly on optimizing component connections. A second shortcoming of AOP is that a unit – or object – cannot be associated with more than one aspect instance of a single type. This is problematic when, for instance, a dispatcher connection in DiPS+ needs optimization, since dispatchers have connections to be optimized with at least three components.

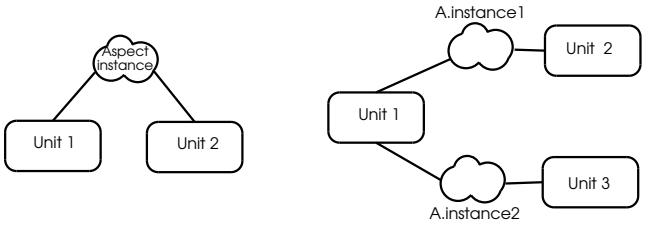



Figure 6.3: Two relevant AOP shortcomings: aspect relation between two or more objects (left), and two or more aspect instances related to the same object (right).

The main cause for these shortcomings is that AOP approaches are type-oriented. AOP treats all object instances of a given type the same, which explains why associating to an object multiple instances of a single aspect makes no sense. Moreover, multiple object instances cannot be differentiated, unless the design is changed so as to provide a sub-class for each role. This is, however, not always a viable solution. In the DiPS+ case, for instance, providing different unit sub-types to express different connection types (e.g. a "directly connectable" unit, a "dispatcher connectable" unit, etc.) would make no sense, as this would violate the idea of strict separation between functional and interaction logic.

What we need is, on the one hand, a description that enables software engineers to express component-specific connection relationships (similar to an aspect language) and, on the other hand, a "smart" weaver that interprets this description to find connections needing optimization and identify the components involved. With this information it is able to customize the description to specific component instances involved.

## 6.3 The DiPS+ optimization approach

An optimized DiPS+ protocol stack offers the best-of-two-worlds. It maximally limits DiPS+ overhead while still allowing to evolve towards a system that is adaptable at run-time without recompilation. The approach to create such optimizations automatically is based on the DiPS+ architecture description and builder, which have been proposed in respectively Section 3.5.2 and 3.5.1.

### 6.3.1 Optimization granularity

**Inter-layer optimizations.** The most coarse-grained optimization approach focuses on inter-layer relationships and layer (de)multiplexing. The main point of attention for this kind of optimizations are dispatchers, which serve as demultiplexing points inside a layer. An example that illustrates such inter-layer optimizations, is a stack with a well-known and fixed configuration. In this case,

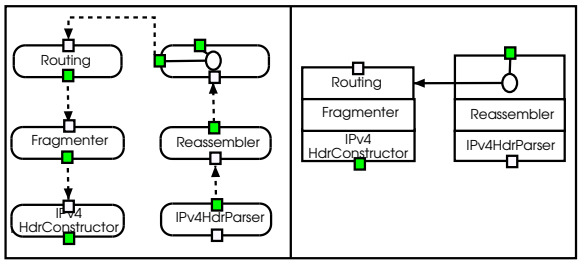



Figure 6.4: Intuitive illustration of the standard DiPS+ IPv4 layer (left), and the fully optimized version (right).

dispatchers can be hard-wired into the code, leaving no possibilities for adding new layers to the stack and, by consequence, removing overhead of dynamically looking up the correct packet forwarder.

**Layer optimization.** The "normal" IPv4 layer structure (see also Listing 3.5 in Chapter 3) provides all DiPS+ flexibility to monitor and manage (concurrency) units. If one knows in advance that this flexibility is not needed, however, this support becomes pure processing and memory overhead. Figure 6.4 zooms in on the IPv4 layer. The dotted arrows in the left figure identify DiPS+ component connections that are unwanted at run-time (e.g. because it is sure that no run-time monitoring is needed). The figure on the right shows the same IPv4 layer, but without all unnecessary DiPS+ support. Only the packet receivers and forwarders at the boundaries are preserved. The dispatcher now has a direct reference to the routing component and this reference can no longer be changed at run-time.

**Partial layer optimization.** A second example is presented in Figure 6.5. Because reassembling fragments is a processing-intensive operation, a concurrency unit is inserted in front of the `Reassembler` component to increase processing power. To allow throughput monitoring, the PR and PF of the concurrency unit must not be removed. If no further DiPS+ support is desired, the other PRs and PFs are pure overhead and thus better removed. The right side of Figure 6.5 shows an example of such partial layer optimization. PRs and PFs at the border components are preserved to enable managing the IPv4 layer as a whole.[2]

**Instance level optimizations.** Various instances of the unit type may be treated differently. As Figure 6.5 shows, the concurrency unit instance in the up going path is not optimized in order to allow monitoring. By contrast, the instance in the down going path is "melted" with its predecessor and successor component. Instance level optimizations are critical since adaptability requirements may vary between particular component instances.

[2]If this is not desired, these PRs and PFs can be removed as well.

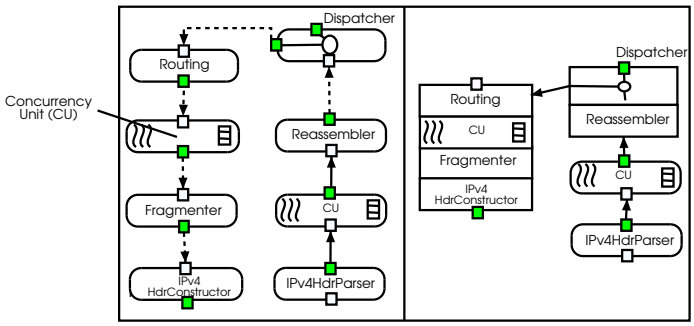



Figure 6.5: Intuitive illustration of the standard DiPS+ IPv4 layer (left), and the partially optimized version (right).

### 6.3.2 Language support for framework optimization

Optimizing protocol stacks is not a common task for today's system administrators. Indeed, traditional protocol stack software is often so complex that the risk of introducing errors during optimization is much too high. Moreover, protocol stacks are rarely developed to be open to customization or optimization. Changing a protocol stack requires digging into the code and altering it, with all risks for errors associated.

Similar to the DiPS+ architecture description, we propose a protocol stack optimization description. This allows system administrators to express their optimization preferences at a descriptive level. In addition, an optimization description may be interpreted and transformed automatically into an optimized protocol stack. This relieves administrators from the difficult and error-prone task of applying optimizations.

The main optimization target is to remove framework code, such as packet receivers, forwarders and dispatchers, inside both component pipes and layers. Listing 6.1 specifies how the optimizations of the IPv4 layer in Figure 6.5 are described. It describes five unit connections that can be "melted" as illustrated in the example. Not only units but also layers may be melted. In this case, the UDP protocol layer must be melted with IPv4.

### 6.3.3 An optimizing builder

Optimizations are effectively deployed by means of an optimizing DiPS+ builder. Thanks to its modularized design, properties as well as handlers that describe how to create a parsed property can easily be introduced. An optimization DiPS+ builder creates a running protocol stack based on an architecture and optimization description. The optimization description specifies where code should be optim-

```
1 <melt_connection from=''Routing'' to=''Active''/>
2 <melt_connection from=''Active'' to=''Fragmenter''/>
3 <melt_connection from=''Fragmenter'' to=''IPv4HdrConstr''/>
4 <melt_connection from=''Reassembler'' to=''Dispatcher''/>
5 <melt_connection from=''Dispatcher'' to=''Routing''/>
6
7 <melt_layer  lower=''IPv4'' upper=''UDP''/>
```

Listing 6.1: The optimization description for a DiPS+ UDP/IP protocol stack.

ized, while the optimization builder deploys layer and connection optimizations.

Layer optimization (`melt_layer` as shown in Listing 6.1) involves removing packet dispatchers, which delegate packets from one layer to another based on the attached meta-information. If it is known that only one layer will be on top of another and if the upper layer will never be replaced at run-time, this support is redundant. In this case, the upper unit from the lower layer is directly connected to the lower unit of the upper layer (and vice versa). If there are multiple alternatives, the dispatcher strategies are replaced by a static condition test.

Connection optimization (`melt_connection`) involves replacement or removal of PRs and PFs. One way of optimization is to replace the PF and PR between two adjacent units by a single optimized packet forwarder that connects the units directly. This preserves the possibility of upgrading to the standard situation, by replacing the optimized forwarder by a standard receiver and forwarder.

The ideal situation removes all DiPS+ code for connecting units. This requires manipulating the source code inside a functional unit, as it drastically changes how units interact. The approach is similar to the AspectJ weaver, which allows changing the code in specific places. Representing connection optimization as an aspect, for instance, results in changing unit code that forwards a DiPS+ packet via a PF and PR to the next unit in the pipe. To this end, packet forwarding code (`forwarder.deliverPacket(Packet p)`) is replaced in every unit by a direct invocation (`nextUnit.incomingPacket(p)`) to omit the extra indirections. Unlike the previous, this optimization approach is irreversible, in other words upgrading would be impossible even if extra framework support were available.

## 6.4 Validation

To validate our optimization approach, we compare the standard and merged protocol stack presented in Section 6.1 with two optimized protocol stacks. (i) On the one hand, an optimized stack ("Optimized DiPS+" version) that minimizes DiPS+ overhead, but can be upgraded to a run-time adaptable version. Moreover, different unit instances of the same class can be treated differently, since the optim-

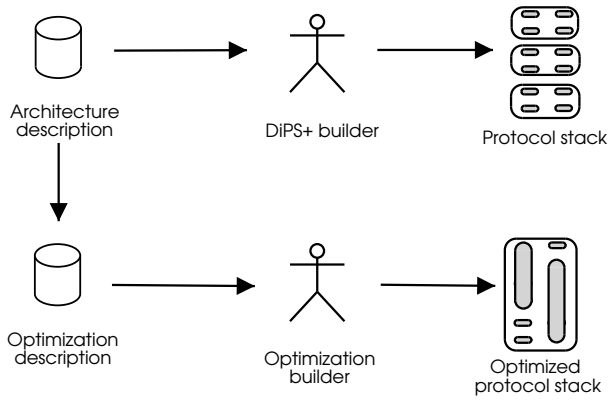



Figure 6.6: Scenario for building a standard and an optimized DiPS+ protocol stack.

ization is transparent to the unit. (ii) An optimized stack ("AspectJ simulation" version) built out of "manipulated" DiPS+ units that cannot be upgraded to a run-time adaptable version, since all DiPS+ support has been removed manually. An ideal AOP weaver would be able to do so automatically, but a weaver that supports this behavior does not yet exist in practice, and simulating this behavior would imply much overhead, coloring performance results considerably. Therefore, we have simulated the ideal situation by hand in order to compare it with the DiPS+ solution.

Figure 6.7 displays performance results of these four protocol stacks. Three conclusions can be drawn. First of all, DiPS+ overhead in the optimized DiPS+ stack is significantly reduced with approximately 40 % compared to the standard DiPS+ version (depending on the JVM used). Secondly, the performance difference between the two optimized versions (Optimized and AspectJ) is very small (2 %), showing our approach to be competitive with AspectJ. Finally, for the SUN and Blackdown JVMs, the overhead of the optimized DiPS+ version is small (5 %) compared to the merged version.[3] Most probably this overhead is due to modularization in version (ii).

## 6.5 Conclusion

This chapter has addressed two challenges for flexible protocol stack software: run-time adaptability and framework optimization. The main contribution of this

[3]In case of the IBM JVM, the difference in throughput between the optimized and the DiPS+ version is considerable. Yet, we have not been able to determine vendor-specific compiler optimizations that may cause this effect.

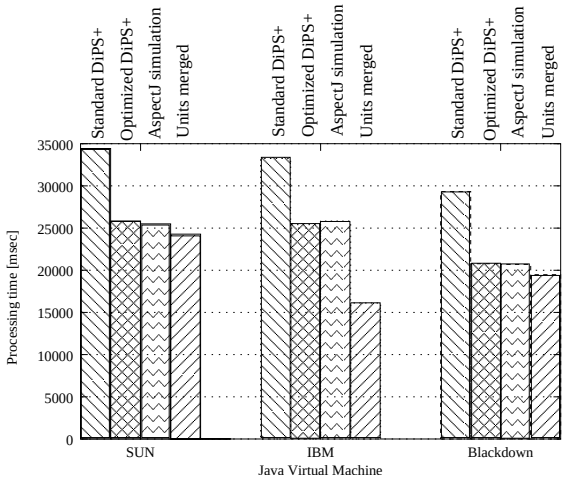



Figure 6.7: UDP/IP/loopback protocol stack performance tests for a standard DiPS+ version, an optimized DiPS+ version, a simulated AspectJ version and one with units merged. 100.000 packets sent, ran on an Intel Pentium 4 CPU 1.70GHz with 512 MB RAM running Linux 2.4.20-pre11. All software written in Java, compiled and executed with JVMs from SUN (1.4.0) [Sunc], IBM (1.4) [IBM03] and Blackdown (1.4.1) [Bla03]. Tests on SUN and Blackdown executed with the -server option for full hotspot optimization [Sun03b].

chapter is threefold: a performance analysis of the DiPS+ framework at both protocol stack and fine-grained unit level, a description language, and an associated DiPS+ builder. The latter two are tailored to optimizing framework overhead in specific places of the architecture. The builder allows differentiating between multiple instances of the same unit type.

*"There is no reason anyone would want a computer in their home."*
*– From Ken Olson.*

# Chapter 7

# Related work

Although multiple frameworks for protocol stack development have been described in the literature [BKC00, FML97, HJE95, KK95, MR01], this chapter compares the DiPS+ approach to the four most relevant frameworks for protocol stack and system software development: Scout [MMOP95], Click [Koh01], Cactus [Bha96], and SEDA [Wel02]. After presenting the four frameworks and comparing their key ideas to the DiPS+ approach, the chapter positions all four along the three software development support challenges studied in this dissertation: adaptability, testability and framework optimization.

## 7.1 Protocol stack frameworks

### 7.1.1 Scout

The Scout operating system [MMOP95] uses a layered software architecture that is similar to layers in the DiPS+ data plane, yet does not offer fine-grained entities such as components for functionality, dispatching, or concurrency. Its primary target are multimedia network protocol stacks. Unlike conventional systems, which are typically structured around computation-centric abstractions (e.g., processes, jobs, tasks), Scout proposes a communication-oriented operating system design.

Scout is designed around a communication-oriented abstraction called the path [MP96]. A path represents an I/O channel throughout a multi-layered system and essentially extends a network connection into the operating system. A path encapsulates the specific attributes of an I/O channel and provides access to these attributes across (protocol) layers.

A path in Scout is comparable to a DiPS+ packet in combination with its meta-information. DiPS+ components can share specific attributes with components

down the packet flow, possibly across layers. As such, a DiPS+ packet leaves a trace through the protocol stack, similar to a path in Scout.

### 7.1.2 Click modular router

The Click modular router [Koh01, KMC+00] is based on a design analogous to DiPS+. Although one can recognize a pipe-and-filter architectural style, Click pays much less attention to software architecture than DiPS+. A Click router is built from fine-grained packet processing modules with private state, called elements. Elements are very similar to DiPS+ units. They can be grouped together into compound elements and each element has input and output ports, serving as end-points for packet transfer. These end-points are comparable to the DiPS+ packet receiver and forwarder. Elements can communicate either directly using function calls or indirectly using a packet queue that uncouples adjacent elements.

Click supports two packet transfer mechanisms: push and pull. The push mechanism is similar to DiPS+ packet forwarding. When using packet pull, the destination element picks one of its input ports and asks the connected source element to return a packet. A network interface, for instance, may pull packets from an upstream queue that collects out going packets.

DiPS+ offers a uniform push packet transfer mechanism and allows active behavior inside the component graph by means of concurrency units. An active concurrency unit encapsulates a packet queue and one or more packet handlers, each of which selects a packet to be pushed throughout the component pipeline. A concurrency unit does not only uncouple units, it also offers active behavior as a separate first-class entity. This allows internal concurrency to be managed and adapted to dynamic circumstances in the system or its environment. Click does not offer information distribution mechanisms like paths in Scout or meta-information in DiPS+.

### 7.1.3 Cactus

The Cactus project offers an integrated design and implementation framework for supporting customizable and fine-grained service quality attributes related to dependability, real time, and security in distributed systems. The model proposed in the Cactus project decomposes complex communication protocols into fine-grained entities, called micro-protocols [Bha96]. Each micro-protocol implements a specific semantic property. This idea of fine-grained decomposition is again similar to the DiPS+ unit design. Micro-protocols have well-defined interfaces and communicate by sending events to each other. Events are intercepted by the Cactus framework and delegated to the appropriate receiver module. Protocols can be reconfigured by adapting the destination of incoming events. The Cactus approach has been applied to middleware services [Hil96, HS00] and network protocols [Bha96, WHS01].

Unlike the DiPS+ architecture, which supports various communication architectural styles, and its design, which offers natural abstractions such as protocol layers, the Cactus architecture is purely event-driven, which makes it difficult to understand how a composed protocol works. Moreover, its design completely lacks protocol stack building support.

### 7.1.4 Staged Event-Driven Architecture

The Staged Event-Driven Architecture (SEDA) [WCB01, Wel02] offers an architecture for supporting massively concurrent systems, primarily web servers. SEDA decomposes an application into a flow of independent modules, called stages. Its modularized design allows for fine-grained development, maintenance and control of the system. Stages can communicate with each other by sending events. Every stage consists of a queue of incoming requests (i.e. events), an event handler that processes incoming requests, and a thread pool to support parallelism.

Stages in SEDA can be compared with a DiPS+ component area along with its preceding concurrency unit. SEDA does not provide a clean separation between the functional and the management level, as the DiPS+ architecture does. This strong integration of management into SEDA does not allow changing management strategies easily, let alone removing management altogether by simply unplugging a management level as in DiPS+.

## 7.2 Provided support during software development

If we are to compare our work with these related research projects, we should not only take into account their software architecture and framework support, but also their position along the software development support tracks of adaptability, testability and software optimization.

### 7.2.1 Adaptability

Conceptually, SEDA stages [WCB01, Wel02] are similar to DiPS+ active concurrency units. Stages as well as concurrency units allow specific areas in the system to be individually conditioned to load. Functional logic (presented in the SEDA event handlers) has been separated from non-functional aspects such as thread management and scheduling. A SEDA stage can be managed by a controller that affects scheduling and thread allocation. A thread pool controller, for example, periodically samples the input queue and adds one or more threads when a certain threshold is exceeded. However, the SEDA controller and associated stage are tightly coupled, whereas DiPS+ clearly separates a concurrency unit from the functional code.

Scout's resource management model [SP99] resembles that proposed by resource containers [BDM99] in that both allow managing the resources for an entire data flow through the system as a single unit. Resource containers enable applications to treat the resources consumed by several I/O channels as being part of the same activity. Composition of a path in Scout is static, while resource containers encompass arbitrary sets of resources at run-time. The DiPS+ resource management model concentrates on distributing processing threads. To this end, it offers the active concurrency unit, which allows applying multiple thread distribution models. On top of that, the DMonA management layer offers a means to balance threads dynamically with an eye to optimizing global packet throughput.

### 7.2.2 Testability

Although none of the related frameworks explicitly takes into account testing, we still can judge their ability to provide testing support for system developers.

Just as DiPS+, systems like Click, Cactus and SEDA offer well-defined module interfaces. Moreover, modules are kept independent of each other, a major advantage for isolation testing. To this end, both Cactus and SEDA incorporate an event-based architectural style. Click for its part keeps elements independent by explicit entry-points, similar to DiPS+ packet receivers. Scout is rather coarse-grained and, as a result, less suited for testing elementary functionalities.

### 7.2.3 Framework optimization

The Scout operating system focuses on structural optimizations, such as connection splicing [MMOP95, SHHP00]. Click [KMC+00] optimizes code by restructuring components and by exploiting application, domain or environment knowledge. To this end, it provides a router configuration description of how Click elements are interconnected. Click also offers a configuration language to express such configurations, which can then be processed off-line by (one or more) optimization tools. Each optimization tool reads a configuration, analyzes and/or manipulates it and finally writes out the (possibly modified) configuration. Optimization tools can be composed into a flow in any user-specified order. They are categorized based on three properties: first, on whether they work entirely at the configuration language level or also manipulate element source code; secondly, on whether or not they need to understand element semantics; and finally, on the level of complexity of graph analyses and manipulations.

DiPS+ provides a component and optimization description as well. Yet, whereas Click optimization improves functional performance (by manipulating component code), DiPS+ rather tries to achieve a good balance between performance and flexibility.

*"The more I learn, the more I realize I don't know."*
*– From Albert Einstein.*

# Chapter 8

# Conclusion

## 8.1 Major achievements

This dissertation has presented DiPS+, a sophisticated software architecture and component framework to support developing flexible protocol stacks that are easily customizable to application- and environment-specific requirements. This flexibility has been achieved by enforcing a strict separation of concerns in the DiPS+ component framework. The framework distinguishes not only management from data processing, but also functionality from concurrency, and functionality from component interaction.

The DiPS+ component framework plays a central role in the protocol stack development process. It does not only offer programmers support in developing protocol stacks, it also lays the foundation for a robust and lightweight development process and a balanced software life-cycle. This dissertation has identified and addressed three core challenges in protocol stack development:

- **Self-management** has been addressed by providing a monitoring and management system (DMonA), which is customizable by application-specific adaptability strategies. In addition, the self-management plane has been conceived as an orthogonal extension to the DiPS+ data plane as it is attached to a component's entry and exit points. In this way, DiPS+ units can be reused, whether or not DMonA is present.

- **Testing** involves testing functional correctness from day one of the software development process. Providing a powerful test framework (DiPSUnit) confirms the trend towards agile software engineering methodologies, which are highly iterative and presuppose change of both the design and the required

functionality. This in turn requires the underlying design of production software to be open to testing.

- **Framework optimization** compensates for the overhead that is inherent to the flexibility the DiPS+ component framework offers, without necessarily throwing all flexibility overboard. This sense for real-life concerns is crucial to protocol stack development.

## 8.2 Validation of the DiPS+ framework

DiPS+ has been validated by using it in the context of research projects and Master's theses. The involvement of our research team in both fundamental and applied research projects provides for DiPS+ the ideal balance between innovation and validation in a realistic (industrial) context.

### 8.2.1 Admission control in the RADIUS protocol

RADIUS (Remote Authentication Dial-in User Service) is a client-server internetworking security system running on top of the UDP transport protocol [RRSW97, Rig00]. It controls authentication (verifying user name and password), accounting (logging relevant user activities), and authorization (access-control) in a multi-user environment. A RADIUS client (typically referred to as the Network Access Server or NAS) is usually a dial-up server residing at an Internet Service Provider (ISP), a Virtual Private Network (VPN) server, or a wireless access point. RADIUS clients send user credentials and connection information to a RADIUS server, which is responsible for authenticating the user, and subsequently returning all configuration information necessary for the RADIUS client to deliver service to the user. For example, ADSL[1] users contact the RADIUS client of their ISP to authenticate themselves, and to obtain network configuration information, such as the local IP address to use on the Internet. RADIUS is designed to simplify the security process by separating security technology from communication technology. All user authentication and network service access information is located on the RADIUS server, away from the NAS. As part of accounting services, RADIUS logs user information in a database or a file according to customer's requirements. A typical RADIUS session consists of a sequence of authentication and accounting requests: (1) the user is authenticated by sending an access request; (2) the session is initiated using an accounting-start request; (3) during an active session accounting-interim requests can be sent optionally to log status information; (4) finally, the session is terminated using an accounting-stop request.

[1] ADSL stands for Asymmetric Digital Subscriber Line and provides broadband Internet access using a telephone line.

A RADIUS server provides the context that confirms the need for adaptability, as provided by DiPS+, since it can easily become a bottleneck when, for instance, thousands of ADSL users contact a RADIUS server simultaneously to get Internet access. By consequence, the server should be flexible enough to handle such overload situations gracefully. Indeed, simply dropping packets randomly does not ensure that the overload can be controlled in a pre-defined manner.

To validate the opportunities concerning adaptability, we have used DiPS+ to implement RADIUS authentication and accounting, and customize it to allow admission control based on various application-specific requirements.[2] In this way, low-priority packets can be queued or dropped in favor of high-priority packets. Thanks to its modularized design and the flexibility offered by DiPS+ concurrency components, various priority scheduling strategies can be inserted in multiple places in the RADIUS protocol.

We briefly discuss the RADIUS design and describe two examples of application-specific packet scheduling strategies: one is based on user information, the other on the type of RADIUS request (authentication or accounting).

### The RADIUS design in a DiPS+ nutshell

Figure 8.1 shows the high-level DiPS+ design of RADIUS authentication (left side), and accounting (on the right).[3] Following the RADIUS specification, the accounting layer accepts packets via UDP port 1812, the accounting layer is connected via UDP port 1813. The parsing components (`HeaderParser` and `AttributeParser`) interpret the header and the list of attributes of an incoming RADIUS packet. The header and attribute constructors (`HeaderConstructor` and `AttributeConstructor`) create the RADIUS header and the list of attributes, and attach these to an out going RADIUS response packet. The `NASChecker` verifies the originator of the request and drops the packet in case the NAS is unknown. The parsing and constructing components, as well as the NAS checker, are generic and are reused in both the authentication and the accounting layer.

The authentication layer consists of four extra components specifically for authentication, and a dispatching component to distinguish between processing accepted and rejected packets. The `Authenticator` component looks up the user's password in an external DiPS+ layer resource (e.g. a database or a file). The `AccessRequestAuthenticatorChecker` compares this password with the encrypted password that is encapsulated in the authentication request. Finally, a positive or negative authentication response is prepared by the `AcceptReplyPreparator` or the `RejectReplyPreparator`. The former contacts the authentication database

[2]This research has been carried out in order of Alcatel Bell, and has been part of the SCAN (Service Centric Access Networks) research project, supported by the Flemish institute for the advancement of scientific-technological research in the industry (IWT SCAN #010319).

[3]The top and bottom design in Figure 8.1 only differ in that they insert the two scheduling strategies in different places in the design. The actual strategies are explained later.

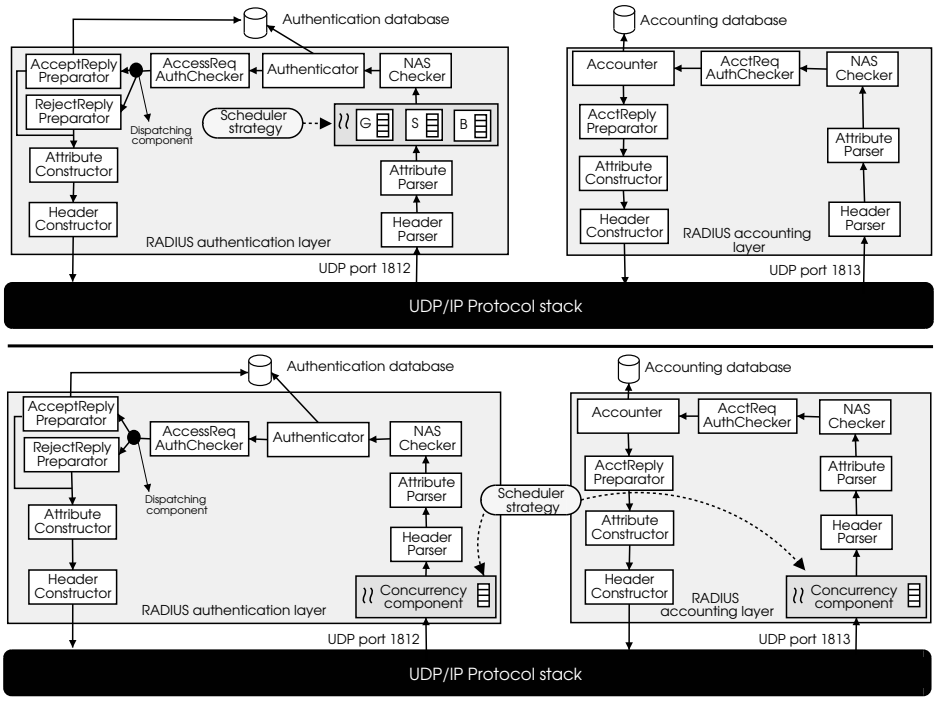



Figure 8.1: Design of RADIUS authentication and accounting in DiPS+. The top figure illustrates a scheduler strategy that distinguishes gold, silver, and bronze users. The bottom figure, demonstrates a strategy that prioritizes accounting requests over authentication requests.

to lookup user-specific configuration information (e.g. the IP address) to respond to the client.

The accounting layer consists of three accounting-specific components. The `AcctRequestAuthenticatorChecker` verifies the integrity of an incoming accounting request. The `Accounter` logs the information in the accounting request to an external DiPS+ layer resource (e.g. a database or a file). The `AcctReply-Preparator` creates a response packet and attaches the necessary meta-information.

### Admission control by user differentiation

A first strategy is based on the idea that RADIUS authentication requests can be prioritized according to the associated user. An ISP, for instance, may identify gold, silver and bronze types of users. One of the advantages of being a gold user is that RADIUS authentication requests have priority over silver and bronze users.

This guarantees fast gold authentication responses, even during peak loads, while silver and bronze users are queued or dropped.

As the top of Figure 8.1 illustrates, user differentiation is accomplished by inserting a concurrency component in the pipeline, after the RADIUS header and attributes have been parsed (i.e. after the `HeaderParser` and the `AttributeParser`). At this point, all required user information is available to classify gold, silver, and bronze users. Based on this classification, the concurrency component stores incoming packets in three separate packet queues. The scheduling strategy associated with the packet handler of the concurrency unit then simply selects packets from the gold queue, until it is empty. As long as no gold packets are available and the silver queue is not empty, silver packets are processed. Finally, bronze packets are only processed when no pending gold or silver packets are available.

#### Admission control by request type differentiation

A second strategy for intelligent admission control is based on the idea that the RADIUS server should maximize the number of active RADIUS sessions. This means that the server should not process any authentication requests as long as accounting requests are pending. Indeed, accepting authentication requests would imply that extra accounting (start/interim/stop) requests are initiated, which only increases the number of pending accounting requests. By blocking authentication requests temporarily, the server controls the load it to be processed, and thereby protects itself from going down under the high load of authentication requests. Moreover, established sessions are not affected by such an overload situation.

As the bottom of Figure 8.1 illustrates, request type differentiation can easily be added in the DiPS+ design by introducing two concurrency components as the entry points for the authentication and the accounting layer. The associated scheduler strategy selects authentication packets only when no accounting packets are available. Note that the strategies for user and request type differentiation can easily be combined. In this case, gold authentications are handled before silver and bronze, but only when no accounting requests are pending.

#### Results

We have compared the DiPS+ RADIUS prototype with an existing Java implementation of RADIUS. During overload, the advantage of using application-specific scheduling strategies becomes clear. On the one hand, gold users are always processed at the maximum throughput while, during overload, the non-DiPS+ version randomly spreads the throughput degradation over all user types. On the other hand, by dropping authentication requests when the throughput limit is reached, the DiPS+ version is able to maximize the number of active sessions at the highest possible throughput. Moreover, the throughput in this case

is substantially higher than for the non-DiPS+ version, since existing sessions are not affected by the overload of incoming authentications.

### 8.2.2 On demand protocol stacks

The DiPS+ framework has also been validated in the context of on-demand composition of a protocol stack, based on application-specific requirements.[4] This is highly relevant given the wide variety of consumer devices that connect to Internet services. An application for Internet telephony, for instance, may be used on a personal computer as well as on a hand-held device, or even on a public Internet terminal.[5] Each of these devices has particular connection characteristics (e.g. wireless or not), and a set of available resources (e.g. processing power or memory), which may have an impact on the protocol stack underneath the application. The protocol stack required for wireless Internet telephony is obviously different from one that is usable in a wired context. Dependent on the level of service quality required, particular protocols should be used (e.g. for reliability of data transfer). In such a dynamic context, where applications may be installed on various types of devices with various capabilities and various required quality levels, creating a protocol stack on-demand is highly relevant.

We have built a prototype in DiPS+ that allows an application to express high-level service requirements (e.g. reliability of data transfer, encryption, local or networked transfer, etc.) that must be supported by the underlying protocol stack. Based on these requirements, a combination of protocol layers is suggested by a stack composition tool [ŞMM01, ŞMBV02, MMMV01], and a protocol stack is built by the DiPS+ builder. Although the research concerning protocol stack composition goes beyond the scope of this dissertation, the stack composition tool can easily be extended to interpret flexibility requirements that describe where in the stack run-time flexibility is needed. In this way, it provides an ideal means to exploit DiPS+ framework optimization, which is crucial for devices with limited processing resources. Moreover, such devices can benefit maximally from intelligently balancing processing resources, as provided in DMonA.

### 8.2.3 DiPS+ Master's theses

Many Master's theses have explored and validated the DiPS+ component framework from various perspectives. First of all, DiPS+ has been used to design and implement particular protocols (e.g. the Session Initiation Protocol (SIP

[4]This research has been carried out in the context of the ITEA/PEPiTA project and supported by the Flemish Institute for the advancement of scientific-technological research in the industry (IWT PEPiTA #990219).

[5]In the future, an Internet terminal may be used as a computer that is activated and customized by inserting a smart card (i.e. a plastic card with a computer chip). Such customizations may involve the protocol stack needed to offer the required level of service quality.

[HSSR99]) in [Van03], the Internet Protocol version 6 (IPv6 [DH95]) in [Jan02], or a Quality-of-Service routing protocol in [Bug01]). Secondly, DiPS+ has been extended without modifying the framework itself in order to explore its applicability. The work in [Bje02], for instance, describes how network management techniques (such as the Simple Network Management Protocol (SNMP) [CFSD] and Java Management eXtensions (JMX) [Sunb]) can be used in combination with DiPS+ protocol stacks. Finally, more research related theses have explored fundamental extensions to the DiPS+ component framework and architecture. The work in [Des02], for instance, explores self-adaptability, while [Mic03a] investigates how parallel access to a component can be controlled independently from the unit.

Two main conclusions may be drawn from our experiences in guiding Master students during their thesis. First of all, the DiPS+ framework and architecture can quickly be assimilated, even by students with limited experience in software architectures and a mainly object-oriented design background. Nevertheless, creating a high-level modularized DiPS+ design of a protocol is not always trivial, and sometimes required assistance of a DiPS+ team member to put the student on the right track. In our view, the students' lack of design experience and the often poor documentation of protocol specifications lie at the basis of the complicated modularization process. The main advantage, compared to an object-oriented design, is that packet flows are clearly defined and well-identifiable, which makes a DiPS+ design much more understandable. Once the high-level design becomes clear, development of individual components is straightforward. Secondly, the theses show that DiPS+ allows for a highly incremental software development process. Stated differently, the first running prototype can usually be delivered quickly after implementation has started (i.e. after a few weeks). From then on, the prototype can easily be customized and extended towards the thesis requirements.

## 8.3 Applicability of the DiPS+ approach

Although the DiPS+ component framework has been proposed in the context of protocol stacks, we are convinced of its applicability in other operating system domains. The first research results in the context of device drivers and file systems are very promising. These systems reflect a layered architecture, which perfectly matches the DiPS+ architecture.

### 8.3.1 USB device drivers

Many device connection standards are competing in the market of both personal computers and audio/video consumer devices. Two major players in device connectivity today are USB (Universal Serial Bus) [CHPI$^+$00] and FireWire (IEEE1394) [App03]. The introduction of a new standard often comes with incomplete and erroneous implementations of the first device driver releases, a situation that may

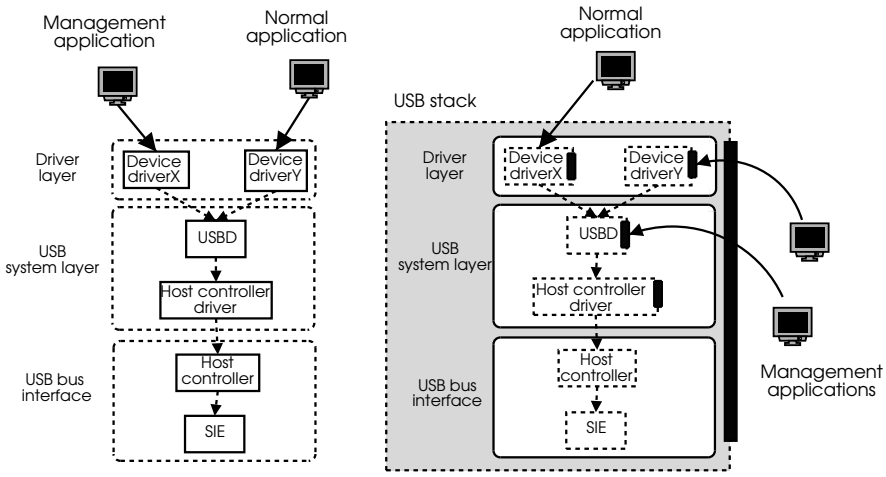



Figure 8.2: Two high-level views on the USB stack: traditional layered architecture (left) and DiPS+ manageable architecture (right).

take several years. Not surprisingly, testing such drivers has become an industry in itself [PMTCP03, USB03].

Stable and robust device driver software is fundamental to gain consumer confidence, which in turn is crucial for a new standard to breakthrough. At the same time, a new standard should anticipate changing consumer preferences and requirements. Since re-implementing complex device drivers would be too expensive and time-consuming, a modular approach that allows software reuse and adaptation is called for.

As illustrated in Figure 8.2, the USB architecture [CHPI+00] is typically layered into three major levels: the USB bus interface – the hardware access layer; the USB system, encapsulating the core of USB; and the USB device driver layer, hosting the actual device-specific drivers. Testing such stack for USB compliance is complicated since only the top (USB driver) layer is directly accessible for applications, which is obviously insufficient from a testing or management perspective. Layering is sometimes bypassed by running at USB driver level a special test driver that acts as the test program. Thoroughly testing implies, however, that particular (erroneous) requests can be injected at specific layers and not only at the top of the stack. Although this problem can be bypassed, such solutions are unnatural, complex, and therefore likely to be error-prone.

We have built a DiPS+ prototype[6] of a USB driver stack that enables managing

[6]This research has been carried out in the context of the CoFraDeD (Component Framework for Device Drivers) research project, supported by the Flemish institute for the advancement of scientific-technological research in the industry (IWT CoFraDeD #990280).

and testing its functionality at various levels [CK01, MKM+00]. The layered architecture of the DiPS+ USB stack closely resembles that of a protocol stack. It allows management applications to interact with the management interface of specific components residing at various layers.

### 8.3.2 File systems

Another key operating system domain is represented by the file system. Again, file systems are typically presented as layered architectures [Mad92, Sou03]. An I/O request sent through these layers is conceptually similar to a network packet traversing a protocol stack.

Like in protocol stacks, a major challenge in file systems is I/O throughput. Apart from applying thread balancing – the solution proposed in Chapter 4 – detecting I/O access patterns and dynamically applying the most optimal caching strategy has been proposed as a highly relevant adaptability approach [MR97a]. For a pattern with largely sequential read requests, for instance, the best choice is clearly a read-ahead caching strategy, which tries to pre-fetch and cache data before it is actually requested. However, as the read pattern becomes more scattered, this strategy suddenly becomes counter-productive. An application domain that is extremely sensitive to such pattern switches is that of web servers [KGF94]. Incoming requests for web pages may differ considerably, from static and largely text-based html pages to dynamically generated pages dominated by embedded picture files. The latter are more processing-intensive and much larger, making them good candidates for data pre-fetch. Or, web pages can be generated dynamically, which migrates the problem from the file system to a back-end data base. One of the major issues here is to develop powerful pattern recognition algorithms (e.g. neural networks).

The DMonA design offers the core platform for monitoring and is open to plugging in specific monitor strategies. As proof-of-concept, we have used a very basic pattern recognition algorithm as DMonA monitoring strategy and allowed data pre-fetching and caching to be switched on and off dynamically. Performance results executed on this prototype have confirmed our expectations and show considerable throughput improvement [MDJ+02, Des02].

## 8.4 Future Work

This dissertation gives rise to multiple research tracks, which will be explored in the future. On the one hand, these tracks will try to validate the benefits of the DiPS+ approach. On the other hand, they may extend DiPS+ with other meta-levels than DMonA, to support the requirements of specific application domains.

The need for distributed system adaptability is highly relevant in the context of so-called active and programmable networks. Such networks – i.e. the routers on

it – are adaptable in order to improve their support of application-specific quality requirements. Unlike current research projects, which typically provide a software execution platform on top of the routers, DiPS+ focuses on the core of a network router, its protocol stack. Manipulating the protocol stack offers more powerful improvements, but implies some major challenges.

### 8.4.1 Component hot-swapping

A first challenge research track is to allow for protocol stack components to be replaced, added or removed. Component hot-swapping is a complex task, especially in a multi-threaded context. Removing a component, for instance, requires blocking the packet flow during removal and re-connecting components to repair the pipeline afterwards. A related challenge is to migrate the internal component state, for instance from a component to its substitute. The ultimate challenge lies in consistently replacing components at multiple network nodes atomically, which is highly relevant to inherently distributed protocol stacks.

The DiPS+ component design may reduce this complexity by allowing to control and temporarily block the incoming packet flow right before the entrance of a component [JMMV02, JMMV03]. Blocking should be transparent to the units themselves, since component hot-swapping is clearly a non-functional concern unit programmers should not be distracted by.

### 8.4.2 Distributed system management

Adapting a protocol stack at the client side may require to change the protocol stack at the server side, and even at all intermediate network nodes. For example, replacing an encryption component at the client probably implies that the decryption component at the server must be replaced too. If data is encrypted in the network layer, all routers in between client and server are affected.

While this dissertation has explored system management of a single network node, the next challenge lies in extending these ideas towards distributed system management. In such context, distributed coordination will be required to deploy adaptations consistently across multiple network nodes.

### 8.4.3 Safe adaptations

The motto “safety first” most certainly applies to core network servers and routers. System adaptability or customization will never be accepted without strong guarantees about (1) the system’s behavior after the adaptation has been deployed, (2) the particular manipulations allowed, and (3) the protection against untrustful adaptors. Together with DiPS+, component hot-swapping, and distributed system management, ensuring safe adaptations is needed to complete the flexible protocol stack puzzle.

# Bibliography


[ABB+86] M. Acetta, R. Baron, W. Bolosky, Il. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. In *Proceedings of the USENIX 1986 Summer Technical Conference*, pages 93–112, Atlanta, GA, USA, June 1986. The USENIX Association, El Cerrito, CA, USA.

[AC96] M. Abadi and L. Cardelli. *Theory of Objects*. Springer-Verlag, New York, NY, USA, 1996.

[AIS77] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[AL00] T. F. Abdelzaher and C. Lu. Modeling and performance control of internet servers. Invited paper at 39th IEEE Conference on Decision and Control, December 2000.

[Amb02] S. W. Ambler. *Agile modeling: Effective Practices for eXtreme Programming and the Unified Process*. J. Wiley & Sons, New York, NY, USA, 2002.

[Aoy98] M. Aoyama. Agile software process and its experience. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 3–12, Kyoto, Japan, 1998. IEEE Computer Society, Washington, DC, USA.

[App03] Apple Computer, Inc. FireWire home page. http://www.apple.com/firewire/, 2003.

[Asp03] AspectJ project. AspectJ – crosscutting objects for better modularity. http://www.eclipse.org/aspectj/, 2003.

[BA01] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, October 2001.

[BBB+00] F. Bachman, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Volume II: Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, May 2000.

[BCK98] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[BDM99] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 45–58, New Orleans, LA, USA, February 1999. USENIX Association, Berkeley, CA, USA.

[Bec99] K. Beck. *Extreme Programming explained: embrace change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Ber94] L. Bergmans. *Composing Concurrent Objects - Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs*. PhD thesis, University of Twente, Enschede, The Netherlands, June 1994.

[BG98] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *The Java Report*, 3(7):37–50, 1998.

[BGJ+91] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. R. Malan, and D. Bohman. Microkernel operating system architecture and Mach. *Journal of Information Processing*, 14(4):442–453, 1991.

[BH02] J. Baker and W. C. Hsieh. Runtime aspect weaving through metaprogramming. In *Proceedings of the First International Conference on Aspect-Oriented Software Development (AOSD'02)*, pages 86–95, April 2002.

[Bha96] N. T. Bhatti. *A System for Constructing Configurable High-level Protocols*. PhD thesis, Department of Computer Science, University of Arizona, Tucson, AZ, USA, December 1996.

[Bin94a] R. V. Binder. Design for testability in object-oriented systems. *Communications of the ACM*, 37(9):87–101, 1994.

[Bin94b] R. V. Binder. Testing Object-Oriented Systems: A Status Report. In *Proceedings of the Third International Conference on Software Testing, Analysis & Review*, May 1994.

[Bin99] R. V. Binder. *Testing object-oriented systems: models, patterns and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Bje02] S. E. Bjerke. Support for Network Management in the DiPS Component Framework. Master's thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, May 2002.

[BJS00] L. Breslau, S. Jamin, and S. Schenker. Comments on the performance of measurement-based admission control algorithms. In *Proceedings of IEEE INFOCOM 2000*, pages 1233–1242, March 2000.

[BKC00] F. J. Ballesteros, F. Kon, and R. Campbell. Off++: The Network in a Box. In *Proceedings of ECOOP Workshop on Object Orientation in Operating Systems (ECOOP-WOOOS 2000)*, Sophia Antipolis and Cannes, France, June 2000.

[Bla03] Blackdown project. Java Linux. http://www.blackdown.org/, 2003.

[BMM+99] J. Bosch, P. Molin, M. Mattsson, P. Bengtsson, and M. E. Fayad. Object-oriented frameworks: Problems & experiences. In *Building Application Frameworks: Object Oriented Foundations of Framework Design*, pages 55–82. J. Wiley & Sons, New York, NY, USA, 1999.

[BMR+96] F. Bushmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. J. Wiley & Sons, New York, NY, USA, 1996.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin-Cummings, Redwood City, CA, USA, second edition, 1994.

[BSW98] J. Bosch, C. Szyperski, and W. Weck. Summary of the third international workshop on component-oriented programming. In W. Weck, J. Bosch, and C. Szyperski, editors, *Proceedings of the Third International Workshop on Component-Oriented Programming (WCOP'98)*, number 5 in TUCS General Publication, pages 1–5. Turku Centre for Computing Science, 1998.

[Bug01] B. Van Buggenhout. Study and Implementation of a QoS router. Master's thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, May 2001.

[Cah96] V. Cahill. Flexibility in object-oriented operating systems: A review. Technical Report TCD-CS-96-05, Department of Computer Science, Trinity College Dublin, Ireland, 1996.

[CFSD] J. D. Case, M. Fedor, M. L. Schoffstall, and J. R. Davin. A simple network management protocol specification (snmp). http://www.ietf.org/rfc/rfc1157.txt.

[CHPI+00] Compaq, Hewlett Packard, Intel, Lucent, Microsoft, NEC, and Philips. Universal Serial Bus (USB) 2.0 specification. http://www.usb.org/, April 2000.

[CK01] W. De Coster and M. De Krock. CoFraDeD: a Component Framework for Device Drivers. Technical report, internal use only, PIMC/K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, October 2001.

[CKF+01] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong. Structuring operating system aspects: using aop to improve os structure modularity. *Communications of the ACM*, 44(10):79–82, 2001.

[CM02] H. Chen and P. Mohapatra. Session-based overload control in QoS-aware web servers. In *Proceedings of IEEE INFOCOM 2002*, New York, NY, USA, June 2002.

[CMC01] X. Chen, P. Mohapatra, and H. Chen. An admission control scheme for predictable server response time for web accesses. In *Proceedings of the tenth international conference on World Wide Web*, pages 545–554. ACM Press, New York, NY, USA, May 2001.

[Cop96] J.O. Coplien. *Software Patterns*. SIGS Publications, New York, NY, USA, 1996.

[CP98] L. Cherkasova and P. Phaal. Session based admission control: a mechanism for improving the performance of an overloaded web server. Technical Report HPL-98-119, HP labs, June 1998.

[Des02] L. Desmet. Adaptive System Software with the DiPS Component Framework. Master's thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, May 2002.

[Deu89] L. P. Deutsch. Design reuse and frameworks in the smalltalk-80 system. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, pages 57–71. ACM Press, New York, NY, USA, New York, NY, USA, 1989.

[DGH+02] Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. Tilbury. Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server. In *Proceedings of Network Operations and Management Symposium*, Florence, Italy, April 2002.

[DH95] S. Deering and R. Hinden. Internet Protocol, version 6 specification (IPv6). http://www.ietf.org/rfc/rfc1883.txt, 1995.

[DIX80] Digital Equipment Corporation, Intel Corporation, and Xerox Corporation. The ethernet - a local area network, version 1.0, September 1980.

[EAK+01] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing aspects of AOP. *Communications of the ACM*, 44(10):33–38, October 2001.

[ERFL01] E. Eide, A. Reid, M. Flatt, and J. Lepreau. Aspect weaving as component knitting: Separating concerns with Knit. In *Proceedings of the Workshop on Advanced Separation of Concerns in Software Engineering, ICSE 2001*, May 2001.

[FML97] B. Ford, K. Van Maren, and J. Lepreau. The flux os toolkit: Reusable components for OS implementation. *In Proceedings of the Sixth IEEE Workshop on Hot Topics in Operating Systems (HoTOS-VI)*, May 1997.

[Fow00] M. Fowler. Put your process on a diet. *Software Development. Special issue on Lightweight Methodologies*, 8(12):32–36, 2000.

[Fre91] R. S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, June 1991.

[FS97] M. E. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, October 1997.

[Gab96] R. Gabriel. *Patterns of Software.* Oxford University Press, 1996.

[Gao00] J. Gao. Component testability and component testing challenges. In *Proceedings of International Workshop on Component-based Software Engineering (CBSE2000, held in conjunction with the 22nd International Conference on Software Engineering (ICSE2000)*, Limerick, Ireland, June 2000.

[GFF02] A. Gal, P. H. Fröhlich, and M. Franz. An efficient execution model for dynamically reconfigurable component software. In *Proceedings of Workshop on Component-Oriented Programming (WCOP'02)*, June 2002.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GK96] K. M. Goudarzi and J. Kramer. Maintaining node consistency in the face of dynamic change. In *Proceedings of the Third International Conference on Configurable Distributed Systems (ICCDS'96)*, pages 62–69, Annapolis, MD, USA, May 1996. IEEE Computer Society Press.

[GKW02] D. Garlan, J. Kramer, and A. Wolf, editors. *Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)*, Charleston, SC, USA, 2002. ACM Press, New York, NY, USA.

[GP95] D. Garlan and D. E. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4):269–274, 1995.

[GS93] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, Singapore, 1993. World Scientific Publishing Company.

[GSPS01] A. Gal, W. Schröder-Preikschat, and O. Spinczyk. On minimal overhead operating systems and aspect-oriented programming. In *Proceedings of the 4th ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS'2001)*, Budapest, Hungary, June 2001. SERVITEC.

[Har00] M. J. Harrold. Testing: a roadmap. In *Proceedings of the conference on The future of Software Engineering, held in conjunction with the International Conference on Software Engineering (ICSE)*, pages 61–72, Limerick, Ireland, 2000. ACM Press, New York, NY, USA.

[HAV03] Home Audio Video Interoperability HAVi. Havi v1.0 specification. http://www.havi.org/, 2003.

[Het88] B. Hetzel. *The complete guide to software testing.* QED Information Sciences, Inc., Wellesley, MA, USA, second edition, 1988.

[Hil96] M. A. Hiltunen. *Configurable Fault-Tolerant Distributed Services.* PhD thesis, Department of Computer Science, University of Arizona, Tucson, AZ, USA, July 1996.

[HJE95] H. Hüni, R. E. Johnson, and R. Engel. A framework for network protocol software. In *Proceedings of the tenth annual conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, pages 358–369, Austin, TX, USA, October 1995. ACM Press, New York, NY, USA.

[HM02] E. R. Harold and W. S. Means. *XML in a Nutshell*. O'Reilly & Associates, Inc., second edition, 2002.

[HMF92] M. J. Harrold, J. D. McGregor, and K. J. FitzPatrick. Incremental testing of object-oriented class structures. In *Proceedings of 14th International conference on Software Engineering (ICSE'92)*, pages 68–80, 1992.

[HP91] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[HS00] M. A. Hiltunen and R. D. Schlichting. The cactus approach to building configurable middleware services. In *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000), in conjunction with the 19th IEEE Symposium on Reliable Distributed Systems*, Nurnberg, Germany, October 2000.

[HSSR99] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. Sip: Session initiation protocol. http://www.ietf.org/rfc/rfc2543.txt, 1999.

[IBM03] IBM Centre for Java Technology Development. IBM developer kits. http://www-106.ibm.com/developerworks/java/jdk/index.html, 2003.

[Int03] Intel Corporation. Intel pentium 4 processor with hyper-threading technology. http://www.intel.com/home/desktop/pentium4/hyperthreading.htm, 2003.

[ISI81a] University of Southern California Information Sciences Institute. Internet protocol specification (ip). http://www.ietf.org/rfc/rfc0791.txt, 1981.

[ISI81b] University of Southern California Information Sciences Institute. Transmission control protocol specification (tcp). http://www.ietf.org/rfc/rfc0793.txt, 1981.

[ITE99] ITEA. Information Technology for European Advancement (ITEA) project home page: PEPiTA (Platform for Enhanced Provisioning of Terminal-independent Applications). http://pepita.objectweb.org/, 1999.

[IYT95] J. Itoh, Y. Yokote, and M. Tokoro. Scone: using concurrent objects for low-level operating system programming. In *Proceedings of the tenth annual conference on Object-Oriented Programming Systems,*

*Languages, and Applications (OOPSLA'95)*, pages 385–398, Austin, TX, USA, October 1995. ACM Press, New York, NY, USA.

[Jan02] G. Janssen. Implementation of IPv6 in DiPS. Master's thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, May 2002.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, first edition, 1999.

[JF88] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

[JMMV02] N. Janssens, S. Michiels, T. Mahieu, and P. Verbaeten. Towards Hot-Swappable System Software: The DiPS/CuPS Component Framework. In *Proceedings - The Seventh International Workshop on Component Oriented Programming*, 2002.

[JMMV03] N. Janssens, S. Michiels, T. Mahieu, and P. Verbaeten. Towards Transparent Hot-Swapping Support for Producer-Consumer Components. In *Proceedings of Second International Workshop on Unanticipated Software Evolution (USE 2003)*, Warsaw, Poland, April 2003.

[Joh97] R. E. Johnson. Frameworks = (components + patterns): How frameworks compare to other object-oriented reuse techniques. *Communications of the ACM*, 40(10):39–42, October 1997.

[JR91] R. E. Johnson and V. F. Russo. Reusing object-oriented design. Technical Report UIUCDCS-R-91-1696, University of Illinois at Urbana-Champaign, IL, USA, May 1991.

[KCK+00] E. Kohler, B. Chen, M. F. Kaashoek, R. Morris, and M. Poletto. Programming language techniques for modular router configurations. Technical Report MIT-LCS-TR-812, MIT Laboratory for Computer Science, August 2000.

[KdRB91] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[KF01] T. Kistler and M. Franz. Continuous program optimization: Design and evaluation. *IEEE Transactions on Computers*, 50(6):549–566, June 2001.

[KGF94] J. F. Karpovich, A. S. Grimshaw, and J. C. French. Extensible File Systems ELFS: An object-oriented approach to high performance file I/O. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94)*, pages 191–204, Portland, OR, USA, October 1994.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–355, Budapest, Hungary, June 2001. Springer-Verlag.

[KK95] K. Calvert R. Kravets and B. Krupczak. An Extensible End-to-End Protocol and Framework. In *In Proceedings of the 1995 IFIP International Working Conference on Upper Layer Protocols, Architectures and Applications (ULPAA'95)*, Sydney, Australia, December 1995.

[KK00] V. Kanodia and E. Knightly. Multi-class latency-bounded web services. In *Proceedings of 8th IEEE/IFIP International Workshop on Quality of Service (IWQoS 2000)*, Pittsburgh, PA, USA, June 2000.

[KLL+97] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, and G. C. Murphy. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, pages 481–490, Boston, MA, USA, 1997. ACM Press, New York, NY, USA.

[KLM+93] G. Kiczales, J. Lamping, C. Maeda, D. Keppel, and D. McNamee. The need for customizable operating systems. In *Proceedings of the Workshop on Workstation Operating Systems*, pages 165–169, 1993.

[KLM+97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, Jyväskylä, Finland, June 1997.

[KM90] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[KMC+00] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[KMD+99] P. Kenens, S. Michiels, O. Debels, S. Van Baelen, W. Joosen, F. Matthijs, and P. Verbaeten. The SmartMove communication architecture. Technical report, SmartMove internal use only, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, August 1999.

[Koh01] E. Kohler. *The Click Modular Router*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, February 2001.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.

[KTW92] G. Kiczales, M. Theimer, and B. Welch. A new model of abstraction for operating system design. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems (IWOOOS'92)*, pages 346–349, Paris, France, 1992. IEEE Computer Society Press.

[LASS01] C. Lu, T. Abdelzaher, J. Stankovic, and S. Son. A feedback control approach for guaranteeing relative delays in web servers. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium (RTAS)*, Taipei, Taiwan, June 2001.

[Lem01a] R. Lemos. Antivirus firms team to thwart DoS attacks. Available at CNET News.com, http://news.com.com/2100-1001-271880.html, August 2001.

[Lem01b] R. Lemos. Web worm targets white house. Available at CNET News.com, http://news.com.com/2100-1001-270272.html, July 2001.

[LH89] K. J. Lieberherr and I. M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, 6(5):38–48, September 1989.

[Lop97] C. V. Lopes. *D - A language famework for distributed programming*. PhD thesis, College of Computer Science, Northeastern University, Boston, MA, USA, September 1997.

[LVC89] M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing user interfaces with interviews. *IEEE Computer*, 22(2):8–22, 1989.

[MA90] D. McNamee and K. Armstrong. Extending the mach external pager interface to accommodate user-level page replacement policies. In *Proceedings of USENIX MACH Symposium*, pages 17–29, Burlington, VT, USA, October 1990.

[Mad92] P. W. Madany. An object-oriented framework for file systems. Technical Report UIUCDCS-R-92-1751, University of Illinois, Urbana-Champaign, IL, USA, 1992.

[Mad96] P. W. Madany. JavaOS: A standalone java environment. Technical Report Available at http://java.sun.com/docs/white/, Sun Microsystems, Inc, May 1996.

[Mat99] F. Matthijs. *Component Framework Technology for Protocol Stacks*. PhD thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, December 1999.

[McG97a] J. D. McGregor. Component testing. *Journal of Object-Oriented Programming (JOOP)*, 10(1):6–9, March 1997.

[McG97b] J. D. McGregor. Parallel Architecture for Component Testing. *Journal of Object-Oriented Programming*, May 1997.

[MDJ+02] S. Michiels, L. Desmet, N. Janssens, T. Mahieu, and P. 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, SC, USA, 2002. ACM SIGSOFT, ACM press.

[MER96] T. M. Madhyastha, C. L. Elford, and D. A. Reed. Optimizing input/output using adaptive file system policies. In *Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems (MSS'96)*, pages 493–514, College Park, MD, USA, September 1996. IEEE Computer Society Press, CA, USA.

[Met99] R. Mettala. Bluetooth protocol architecture version 1.0. White paper, Nokia Mobile Phones, August 1999.

[Mic02] Micosoft Corporation. COM: Delivering on the Promises of Component Technology. http://www.microsoft.com/com/, 2002.

[Mic03a] D. Michiels. Concurrency Control in the DiPS framework. Master's thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, May 2003.

[Mic03b] Microsoft Corporation. Automatic update services. http://www.microsoft.com/downloads/updateservices.aspx, 2003.

[MJV+98] F. Matthijs, W. Joosen, B. Vanhaute, B. Robben, and P. Verbaeten. Aspects should not die. Technical Report CW-266, Dept. of Computer Science, K.U.Leuven, Leuven, Belgium, April 1998.

[MK96] J. D. McGregor and A. Kare. Parallel architecture for component testing. In *Proceedings of the Ninth International Quality Week*, 1996.

[MKM+00] S. Michiels, P. Kenens, F. Matthijs, D. Walravens, Y. Berbers, and P. Verbaeten. Component Framework Support for developing Device Drivers. In N. Rozic, D. Begusic, and M. Vrdoljak, editors, *International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, volume 1, pages 117–126, Split, Croatia, 2000. FESB.

[MMMV01] S. Michiels, T. Mahieu, F. Matthijs, and P. Verbaeten. Dynamic Protocol Stack Composition: Protocol Independent Addressing. In *Proceedings of the 4th ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS'2001)*, Budapest, Hungary, June 2001. SERVITEC.

[MMOP95] A. B. Montz, D. Mosberger, S. W. O'Malley, and L. L. Peterson. Scout: A communications-oriented operating system. In *Proceedings of Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 58–61, Orcas Island, WA, USA, May 1995. IEEE Computer Society Press.

[MMWV01] S. Michiels, F. Matthijs, D. Walravens, and P. Verbaeten. Position summary: DiPS: A Unifying Approach for Developing System Software. In A. D. Williams, editor, *Proceedings - The Eigth Workshop on Hot Topics in Operating Systems (HoTOS-VIII*, page 175. University of Karlsruhe, IEEE Computer Society Press, 2001.

[MN97] T. D. Meijler and O. Nierstrasz. Beyond objects: Components. In M. P. Papazoglou and G. Schlageter, editors, *Cooperative Information Systems: Current Trends and Directions*, pages 49–78. Academic Press, 1997.

[MP96] D. Mosberger and L. L. Peterson. Making paths explicit in the scout operating system. In *Proceedings of USENIX Second Symposium Operating Systems Design and Implementation (OSDI'96)*, pages 153–167, Seattle, WA, USA, October 1996. USENIX Association, Berkeley, CA, USA.

[MPBO96] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley. Analysis of techniques to improve protocol processing latency. In *Proceedings of the Conference on Applications, Technologies, Architectures and Protocols for Computer Communications*, pages 73–84, Stanford, CA, USA, August 1996.

[MR96] T. M. Madhyastha and D. A. Reed. Intelligent, adaptive file system policy selection. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'96)*, pages 172–179, Annapolis, Maryland, USA, 1996. IEEE Computer Society Press.

[MR97a] T. M. Madhyastha and D. A. Reed. Exploiting global access pattern classification. In *Proceedings of SC97: High Performance Networking and Computing*, San Jose, CA, USA, November 1997. ACM Press.

[MR97b] T. M. Madhyastha and D. A. Reed. Input/output access pattern classification using hidden markov models. In *Proceedings of Workshop on Input/Output in Parallel and Distributed Systems*, pages 57–67, San Jose, CA, USA, November 1997. ACM Press.

[MR01] H. Miranda and A. Pinto L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 707–710, Phoenix, Arizona, USA, April 2001. IEEE.

[MWJV] S. Michiels, D. Walravens, N. Janssens, and P. Verbaeten. DiPS: Filling the Gap between System Software and Testing.

[MWJV02a] S. Michiels, D. Walravens, N. Janssens, and P. Verbaeten. DiPSUnit: A JUnit Extension for the DiPS Framework. In *Proceedings of Third International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP2002)*, Alghero, Italy, May 2002.

[MWJV02b] S. Michiels, D. Walravens, N. Janssens, and P. Verbaeten. DiPSUnit: an Extension of the JUnit Test Framework for DiPS. Report CW 333, Dept. of Computer Science, K.U.Leuven, Leuven, Belgium, February 2002.

[ND95] O. Nierstrasz and L. Dami. Component-oriented software technology. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice-Hall, 1995.

[Nor99] S. Northcutt. *Network Intrusion Detection: An Analysts' Handbook*. Que, first edition, 1999.

[NSN+97] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Sixteenth ACM Symposium on Operating Systems Principles (SOSP-16)*, pages 276–287, Saint Malo, France, 1997. ACM Press, New York, NY, USA.

[OMT98] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In R. Kemmerer and K. Futatsugi, editors, *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 62–70, Kyoto, Japan, April 1998. IEEE Computer Society Press.

[Opd92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1992.

[OT01] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, October 2001.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Par76] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, 1976.

[PGA02] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the First International Conference on Aspect-Oriented Software Development (AOSD'02)*, pages 141–147, April 2002.

[Plu82] D. C. Plummer. An ethernet address resolution protocol (arp). http://www.ietf.org/rfc/rfc0826.txt, 1982.

[PMTCP03] Professional Multimedia Testing Centre (PMTC). Professional Interactive Media Centre (PIMC). http://www.pmtctest.com, 2003.

[Pos80] J. Postel. User datagram protocol specification (udp). http://www.ietf.org/rfc/rfc0768.txt, 1980.

[Pre95] W. Pree. *Design Patterns for Object-Oriented Software Development*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[PSJ+01] E. Pulvermüller, A. Speck, J.O.Coplien, M. D'Hondt, and W. DeMeuter. Proceedings of the ecoop'01 workshop on feature interaction in composed systems. Technical Report 2001-14, University of Karlsruhe, Karlsruhe, Germany, September 2001.

[PW92] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[RFS+00] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proceedings of 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 347–360, San Diego, CA, USA, October 2000. USENIX Association, Berkeley, CA, USA.

[Rig00] C. Rigney. Radius accounting. http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc2866.html, 2000.

[RJ97] D. Roberts and R. Johnson. *Pattern languages of program design 3*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[RMJ+98] B. Robben, F. Matthijs, W. Joosen, B. Vanhaute, and P. Verbaeten. Components for non-functional requirements. In W. Weck, J. Bosch, and C. Szyperski, editors, *Proceedings of Third International Workshop on Component-Oriented Programming (WCOP'98)*, TUCS General Publication, pages 75–81, Brussels, Belgium, January 1998.

[Rob99] B. Robben. *Language Technology and Metalevel Architectures for Distributed Objects*. PhD thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, May 1999.

[RRSW97] C. Rigney, A. Rubens, W. Simpson, and S. Willens. Remote authentication dial in user service specification (radius). http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc2138.html, 1997.

[RT78] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6):1905–1929, 1978.

[Rus90] V. F. Russo. *An Object-Oriented Operating System*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, October 1990.

[SG96] M. Shaw and D. Garlan. *Software Architecture - Perspectives on an emerging discipline*. Prentice-Hall, 1996.

[SGC02] K. Sullivan, L. Gu, and Y. Cai. Non-modularity in aspect-oriented languages: Integration as a crosscutting concern for aspectj. In Y. Coady, editor, *Proceedings of the First International Conference on Aspect-Oriented Software Development (AOSD'02)*, pages 19–27, Enschede, The Netherlands, April 2002. ACM Press, New York, NY, USA.

[SGG+99] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling.

In *Proceedings of the third USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 145–158, New Orleans, LA, USA, February 1999. USENIX Association, Berkeley, CA, USA.

[SHHP00] O. Spatscheck, J. S. Hansen, J. H. Hartman, and L. L. Peterson. Optimizing tcp forwarder performance. *IEEE/ACM Transactions on Networking (TON)*, 8(2):146–157, 2000.

[Sim94] W. Simpson. The point-to-point protocol (ppp). http://www.ietf.org/rfc/rfc1661.txt, 1994.

[ŞMBV02] I. Şora, F. Matthijs, Y. Berbers, and P. Verbaeten. Automatic Composition of Systems from Components with Anonymous Dependencies Specified by Semantic-Unaware Properties. In T. D'Hondt, editor, *Proceedings of Technology of Object-Oriented Languages and Systems Eastern Europe (TOOLSEE) 2001*. IEEE Computer Society Press, March 2002.

[ŞMM01] I. Şora, S. Michiels, and F. Matthijs. Policies for dynamic stack composition. Technical Report CW-313, Dept. of Computer Science, K.U.Leuven, Leuven, Belgium, February 2001.

[SN99] J-G. Schneider and O. Nierstrasz. Components, scripts and glue. In J. Hall L. Barroca and P. Hall, editors, *Software Architectures – Advances and Applications*, pages 13–25. Springer-Verlag, 1999.

[Sou03] SourceForge.net. Ext2fs home page. http://e2fsprogs.sourceforge.net/ext2.html, 2003.

[SP99] O. Spatscheck and L. L. Petersen. Defending against denial of service attacks in scout. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, New Orleans, LA, USA, February 1999. USENIX Association, Berkeley, CA, USA.

[SS97] M. Seltzer and C. Small. Self-Monitoring and Self-Adapting Operating Systems. In *The Sixth Workshop on Hot Topics in Operating Systems (HoTOS-VI)*, pages 124–129, Cape Cod, MA, USA, May 1997. IEEE Computer Society Press.

[SSC98] A. Sane, A. Singhai, and R. H. Campbell. Framework design for end-to-end optimization. In *Proceedings of 12th European Conference on Object Oriented Programming (ECOOP'98)*, volume 1445 of *Lecture Notes in Computer Science*, pages 135–157, Brussels, Belgium, July 1998.

[Ste00] W. R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley professional computing series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Sul01] G. T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44(10):95–97, October 2001.

[Suna] Sun Microsystems, Inc. The java hotspot performance engine architecture. Available at http://java.sun.com/products/hotspot/whitepaper.html.

[Sunb] Sun Microsystems, Inc. Java Management eXtensions. Available at http://java.sun.com/products/JavaManagement/.

[Sunc] Sun Microsystems, Inc. Java2 Platform, Standard Edition (J2SE). Available at http://java.sun.com/j2se/1.4.1/.

[Sund] Sun Microsystems, Inc. The javabeans specification 1.01. Available at http://java.sun.com/products/javabeans/docs/spec.html.

[Sun03a] Sun Microsystems, Inc. Enterprise javabeans technology. http://java.sun.com/products/ejb/, April 2003.

[Sun03b] Sun Microsystems, Inc. The java hotspot virtual machine. Available at http://java.sun.com/products/hotspot/docs/whitepaper/, 2003.

[Sun03c] Sun Microsystems, Inc. Ultrasparc processors: Throughput computing. http://www.sun.com/processors/throughput/, 2003.

[Szy98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.

[Tek96] B. Tekinerdogan. Adaptability in Object-Oriented Software Development, ECOOP'96 Workshop report. http://trese.cs.utwente.nl/ecoop96adws/wsreport.html, July 1996.

[TM00] P. Craig T. Mackinnon, S. Freeman. Endo-testing: Unit testing with mock objects. eXtreme Programming and Flexible Processes in Software Engineering XP2000, Connextra Ltd., London, UK, June 2000.

[USB03] USB implementers forum, Inc. Independent usb test labs. http://www.usb.org/developers/compliance/labs/, 2003.

[Van03] B. Vandewoestyne. Internet Telephony with the DiPS Framework. Master's thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, May 2003.

[VBC01] J. Viega, J.T. Bloch, and P. Chandra. Applying Aspect-Oriented Programming to Security. *Cutter IT Journal*, 14(2):31–39, February 2001.

[VK89] D. M. Volpano and R. B. Kieburtz. *Software reusability: vol. 1, concepts and models*. ACM Press, New York, NY, USA, 1989.

[Wal93] M. M. Waldrop. *Complexity. The emerging science at the edge of order and chaos*. Touchstone, 1993.

[WBJ90] R. J. Wirfs-Brock and R. E. Johnson. Surveying current research in object-oriented programming. *Communications of the ACM*, 33(9):104–124, 1990.

[WC01] M. Welsh and D. Culler. Virtualization Considered Harmful: OS Design Directions for Well-Conditioned Services. In A. D. Williams, editor, *Proceedings of Eigth Workshop on Hot Topics in Operating Systems (HoTOS-VIII)*, pages 139–146, Elmau, Germany, May 2001. IEEE Computer Society Press.

[WCB01] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP-18)*, pages 171–180, Banff, Canada, 2001. ACM Press, New York, NY, USA.

[Wel79] J. Welsh. *Introduction to Pascal*. Prentice-Hall, New Jersey, NJ, USA, 1979.

[Wel02] M. F. Welsh. *An Architecture for Highly Concurrent, Well-Conditioned Internet Services*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, August 2002.

[Wey98] E. J. Weyuker. Testing component-based software. *IEEE Software*, 15(5):54–59, 1998.

[WGM89] A. Weinand, E. Gamma, and R. Marty. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming*, 10(2):63–87, 1989.

[WHS01] G. T. Wong, M. A. Hiltunen, and R. D. Schlichting. A configurable and extensible transport protocol. In *Proceedings of INFOCOM 2001*, pages 319–328, 2001.

[WS00] L. A. Wald and S. Schwarz. The 1999 Southern California Seismic Network Bulletin. Seismological Research Letters, 71(4), July/August 2000.

[WVD02] B. De Win, B. Vanhaute, and B. De Decker. How aspect-oriented programming can help to build secure software. *Informatica*, 26(2):141–149, February 2002.

[Yok92] Y. Yokote. The apertos reflective operating system: the concept and its implementation. In *Conference Proceedings on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 414–434. ACM Press, New York, NY, USA, 1992.

[ZBC+81] H. Zimmerman, J-S. Banino, A. Cariston, M. Guillemont, and G. Morisset. Basic concepts for the support of distributed systems: The chorus approach. In *Proceedings of the second International Conference on Distributed Computing Systems*, pages 60–66, Paris, France, April 1981. Computer Society Press.

# List of Publications

## Contributions at international conferences, published in proceedings

1. "Self-Adapting Concurrency: The DMonA Architecture." S. Michiels, L. Desmet, N. Janssens, T. Mahieu, and P. Verbaeten. In *Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)*, November 18-19, 2002.

2. "Ant Algorithms in a Graph Environment: a Case Study." E. Steegmans, T. Holvoet, N. Janssens, S. Michiels, Y. Berbers, P. Verbaeten, P. Valckenaers, and H. Van Brussel. In *Proceedings of IASTED International Conference on Artificial Intelligence and Applications (AIA2002)*, September, 2002.

3. "Towards Hot-Swappable System Software: The DiPS/CuPS Component Framework." N. Janssens, S. Michiels, T. Mahieu, and P. Verbaeten. In *Proceedings of ECOOP Workshop on Component Oriented Programming (WCOP2002)*, June, 2002.

4. "DiPS: Filling the Gap between System Software and Testing." S. Michiels, D. Walravens, N. Janssens, and P. Verbaeten. In *Proceedings of Workshop on Testing in XP (WiTXP2002)*, May, 2002.

5. "DiPSUnit: A JUnit Extension for the DiPS Framework." S. Michiels, D. Walravens, N. Janssens, and P. Verbaeten. Dept. of Comp. Science, K.U.Leuven, Belgium, In *Proceedings of Third International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP2002)*, May, 2002.

6. "Dynamic Protocol Stack Composition: Protocol independent Addressing." S. Michiels, T. Mahieu, F. Matthijs, and P. Verbaeten. In *Proceedings of 4th ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS'2001)*, June, 2001.

7. "Position Summary: DiPS: A Unifying Approach for Developing System Software." S. Michiels, F. Matthijs, D. Walravens, and P. Verbaeten. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*. May, 2001.

8. "Component Framework support for developing Device Drivers." S. Michiels, P. Kenens, F. Matthijs, D. Walravens, Y. Berbers, and P. Verbaeten. In *Proceedings of International Conference on Software, Telecommunications and Computer Networks (SoftCOM2000)*, October, 2000.

9. "Open Implementation of a Mobile Communication System." E. Truyen, B. Robben, P. Kenens, F. Matthijs, S. Michiels, W. Joosen, and P. Verbaeten. In *Proceedings of European Conference on Object Oriented Programming (ECOOP'98), Object-Oriented Technology - ECOOP'98 Workshop Reader*, June, 1998.

10. "An AOP Case with Static and Dynamic Aspects." P. Kenens, S. Michiels, F. Matthijs, B. Robben, E. Truyen, W. Joosen, and P. Verbaeten. In *Object-Oriented Technology : ECOOP'98 Workshop Reader*, Lecture Notes in Computer Science, vol. 1543, June, 1998.

## Technical reports

1. "DiPSUnit: an Extension of the JUnit Test Framework for DiPS." S. Michiels, D. Walravens, N. Janssens, and P. Verbaeten. Technical Report CW-333, Dept. of Comp. Science, K.U.Leuven, Belgium, February, 2002.

2. "DiPS/CuPS: a Framework for Runtime Customizable Protocol Stacks." N. Janssens, S. Michiels, and P. Verbaeten. Technical Report CW-328, Dept. of Comp. Science, K.U.Leuven, Belgium, November, 2001.

3. "Policies for dynamic stack composition." I. Şora, S. Michiels, and F. Matthijs. Technical Report CW-313, Dept. of Computer Science, K.U.Leuven, Belgium, February, 2001.

4. "Integration of Device Switch Logic in the SmartMove Protocol Stack." Sam Michiels, O. Debels, P. Kenens, F. Matthijs, S. Van Baelen, W. Joosen, and P. Verbaeten. SmartMove Technical Report, Dept. of Computer Science, K.U.Leuven, Belgium, December 1998.

5. "The SmartMove Communication Architecture." P. Kenens, S. Michiels, O. Debels, S. Van Baelen, F. Matthijs, W. Joosen, and P. Verbaeten. SmartMove Technical Report, Dept. of Computer Science, K.U.Leuven, Belgium, August 1999.

6. "Communication System: System Design Document." P. Kenens, S. Michiels, F. Matthijs, S. Van Baelen, W. Joosen, and P. Verbaeten. SmartMove Technical Report, Dept. of Computer Science, K.U.Leuven, Belgium, February 1998.

# Curriculum vitae

Sam Michiels was born in Aarschot (Belgium) on October 28, 1974. After his secundary education (Latin-Sciences) at the Damiaaninstituut in Aarschot, he received a Bachelor degree in computer sciences (Kandidaat Informatica) at the university of Diepenbeek, and a Master's degree (Licentiaat Informatica) at the K.U.Leuven. In 1996 he graduated at the K.U.Leuven as Licentiaat Informatica, with the thesis 'Realisation of interactive video applications via broadband ATM networks' under supervision of Professor Pierre Verbaeten. The same year, he was admitted into the DistriNet (Distributed systems and computer Networks) research group as research assistant. Besides his doctoral research, his task mainly consists of guiding students during their Master's thesis and advising new members in the research group during their first year of research. In addition, he represented the (teaching and research) assistants in the board of the Faculty of Engineering at the K.U.Leuven, and currently represents the same group in the Academic Council of the K.U.Leuven.

# Componentenraamwerktechnologie voor het ontwikkelen van aanpasbare en beheersbare protocolstapels

Sam Michiels
DistriNet, Departement Computerwetenschappen, K.U.Leuven
Celestijnenlaan 200A
B-3001 Leuven (Heverlee)
Sam.Michiels@cs.kuleuven.ac.be

## Abstract

De context voor het onderzoek in deze thesis wordt gevormd door netwerkdiensten en de vereisten die ze stellen aan de onderliggende protocolstapel. De relevantie van dit onderzoek blijkt ondermeer uit drie belangrijke tendensen in netwerkdiensten en hun omgeving: de toenemende afhankelijkheid van het netwerk voor zowel bedrijven als particulieren, de zeer dynamische netwerkomgevingen, en de grote diversiteit qua geconnecteerde toestellen. Deze drie tendensen brengen een gemeenschappelijke uitdaging aan het licht: de ondersteuning van flexibiliteit in protocolstapels tijdens hun ontwerp, ontwikkeling en uitvoering, en dit afgestemd op applicatie- en/of netwerkspecifieke karakteristieken.

Deze thesis presenteert DiPS+ (Distrinet Protocol Stack+), een softwarearchitectuur en componentenraamwerk dat toelaat om op een flexibele manier protocolstapels te ontwikkelen die makkelijk aangepast kunnen worden aan applicatie- en omgevingsspecifieke vereisten. Dankzij DiPS+ kunnen drie belangrijke onderzoekspistes bij het ontwikkelen van protocolstapels aangepakt worden: zelfaanpasbaarheid en -beheer, testen, en raamwerkoptimalisatie.

De kracht van deze thesis ligt erin dat deze drie pistes niet door aparte oplossingen aangepakt worden, maar dat ze kunnen terugvallen op een gemeenschappelijk componentenraamwerk en software-architectuur. Het DiPS+ componentenraamwerk ondersteunt niet alleen programmeurs om protocolstapels te ontwikkelen, het legt ook de basis voor een robuust en soepel ontwikkelingsproces en voor een evenwichtige levenscyclus.

# Inhoudsopgave

**1 Inleiding** **i**
1.1 Protocolstapels . . . . . i
1.2 Context . . . . . i
1.3 Flexibele protocolstapels . . . . . iii
1.4 Belangrijkste bijdragen . . . . . iii
1.5 Overzicht . . . . . iv

**2 Achtergrond** **iv**
2.1 Sleutelproblemen en eigenschappen van protocolstapels . . . . . iv
2.2 Onderzoeksuitdagingen . . . . . v

**3 Het DiPS+ componentenraamwerk** **vi**
3.1 Van DiPS naar DiPS+ . . . . . vi
3.2 De architectuur van DiPS+ . . . . . vii
3.3 Het ontwerp van het datavlak . . . . . viii
3.4 Het ontwerp van het beheersvlak . . . . . x
3.5 Besluit . . . . . x

**4 Zelfaanpasbaar systeemgedrag** **xi**
4.1 De DMonA aanpak . . . . . xi
4.2 Een casestudy: beheer van de systeembelasting . . . . . xii

**5 Testen in DiPS+** **xv**
5.1 De testproblematiek vanuit een DiPS+ perspectief . . . . . xv
5.2 Het DiPSUnit testraamwerk . . . . . xvi
5.3 DiPSUnit in de praktijk . . . . . xvii
5.4 Besluit . . . . . xviii

**6 Optimalisatie van het DiPS+ raamwerk** **xviii**
6.1 Beperkingen van optimaliserende vertalers . . . . . xix
6.2 Optimalisatie volgens de DiPS+ aanpak . . . . . xix
6.3 Besluit . . . . . xx

**7 Gerelateerd werk** **xx**
7.1 Raamwerken voor protocolstapels . . . . . xx
7.2 Ondersteuning tijdens software-ontwikkeling . . . . . xxi

**8 Besluit** **xxii**
8.1 De belangrijkste resultaten . . . . . xxii
8.2 Validatie van het DiPS+ raamwerk . . . . . xxii
8.3 Toepasbaarheid van de DiPS+ aanpak . . . . . xxiii
8.4 De toekomst . . . . . xxiii

# 1 Inleiding

Na een korte schets van de taken en eigenschappen van een protocolstapel in Sectie 1.1, kadert Sectie 1.2 de context en probleemstelling van de thesis. Als oplossing voor de gestelde problemen wordt in Sectie 1.3 geopteerd voor het ontwikkelen van flexibele protocolstapels. Deze oplossing vertaalt zich in drie cruciale uitdagingen in het ontwikkelen van protocolstapels: zelfaanpasbaarheid, testen, en raamwerkoptimalisatie. Tenslotte bespreekt Sectie 1.4 de belangrijkste bijdragen die uitgewerkt worden in deze thesis, en geeft Sectie 1.5 een kort overzicht van de tekst.

## 1.1 Protocolstapels

Het doel van een protocolstapel bestaat erin om communicatie mogelijk te maken tussen machines die aangesloten zijn op een netwerk [Tan96]. De hoofdtaken van een protocolstapel zijn onderverdeeld in protocollagen die, zoals in een stapel, op elkaar liggen (vandaar de naam protocolstapel) (zie ook Figuur 1).

Elke laag in een protocolstapel steunt op de diensten van de onderliggende laag. De transportlaag, bijvoorbeeld, is verantwoordelijk voor het sturen van boodschappen tussen een zender naar een ontvanger die niet noodzakelijk rechtstreeks met elkaar verbonden zijn. Om een route te bepalen door het netwerk, maakt de transportlaag daarom gebruik van de netwerklaag, die in staat is om de volgende machine op de route van zender naar ontvanger te identificeren.

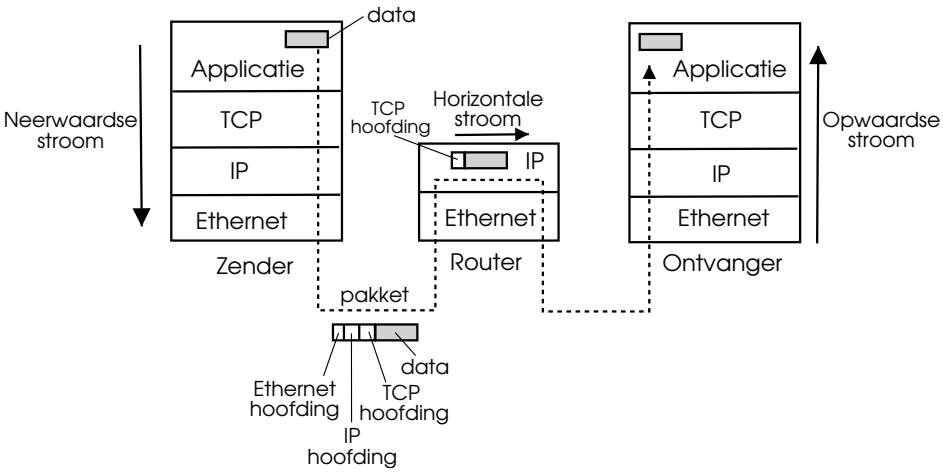
In een protocolstapel kunnen drie datastromen geïdentificeerd worden. Een neerwaartse stroom van uitgaande data die naar het netwerk gestuurd wordt, een opwaartse stroom van pakketjes die arriveren vanuit het netwerk, en een horizontale stroom die een verbinding legt tussen de neerwaartse en de opwaartse stroom.

Protocollagen op hetzelfde niveau in een protocolstapel, maar op verschillende machines, kunnen informatie uitwisselen door een protocolspecifieke hoofding aan een uitgaand pakket toe te voegen. Wanneer een pakket bij de ontvanger toekomt, zal elk protocol de beschikbare informatie in de hoofding interpreteren en verder verwerken.

## 1.2 Context

De context van deze thesis wordt gevormd door netwerkdiensten en de vereisten die ze stellen aan de onderliggende systeemsoftware, i.c. protocolstapels. Belangrijke voorbeelden van netwerkdiensten zijn Internettoegang, aanbieden van e-mailfaciliteiten, het opzoeken van informatie, of het beschikbaar stellen van de laatste nieuwsfeiten.

De relevantie van het onderzoek in deze thesis blijkt ondermeer uit drie belangrijke tendensen in netwerkdiensten en hun omgeving. Eerst en vooral rekent men meer en meer op de beschikbaarheid van het netwerk (zoals bijvoorbeeld het Internet). Als dusdanig is het netwerk een cruciale schakel geworden in het dagelijkse leven, zowel voor bedrijven die het gebruiken voor contacten




Figuur 1: Illustratie van een TCP/IP netwerk met twee eindknopen en een tussenliggende router. De zender stuurt een pakket via de router, die de Ethernet en de IP hoofding verwijdert, de volgende machine op de route identificeert (i.c. de ontvanger), en het pakket doorstuurt via de netwerk- en datalinklaag. Deze twee lagen voegen elk een nieuwe protocolhoofding toe aan het doorgestuurde pakket.

met andere (sub)bedrijven, als voor particulieren, die Internettoegang en e-mail als onmisbaar beginnen te beschouwen. Deze groeiende afhankelijkheid van netwerktoegankelijkheid benadrukt de behoefte aan betrouwbare en robuuste protocolstapels.

Ten tweede is het Internet een zeer dynamische omgeving wat betreft beschikbare bandbreedte, aantal te verwerken aanvragen, soort aanvragen, en verwachte kwaliteit. Het wordt daardoor moeilijk om één protocolstapel te ontwikkelen die voldoet aan deze variërende omstandigheden. Zelfs wanneer de protocolstapel aangepast zou kunnen worden (door een aantal parameters te wijzigen), dan nog zou een manuele interventie tekort schieten om onvoorspelbare veranderingen van korte duur op te vangen.

Ten derde is er een merkbare tendens naar meer variatie in de types van toestellen die aansluiten op het netwerk. Niet alleen computers, maar ook GSM's, persoonlijke digitale assistenten (PDA's), en voertuigen kunnen tegenwoordig beschikken over een protocolstapel. Het spreekt voor zich dat deze toestellen verschillen vertonen wat betreft beschikbare rekenkracht en geheugen. Het is daarom zeer belangrijk dat de protocolstapel makkelijk aangepast kan worden aan het systeem, door bijvoorbeeld overbodige functionaliteit te verwijderen of een module te vervangen door een versie die beter afgestemd is op de beschikbare middelen. Daarenboven is het cruciaal in dit soort omgevingen om de beschikbare middelen maximaal te benutten door de overhead te minimaliseren en de rekenkracht optimaal in te zetten.

*Deze drie tendensen – de groeiende afhankelijkheid van het netwerk, de zeer dynamische netwerkomgevingen, en de grote diversiteit qua geconnecteerde toestellen – brengen een gemeenschappelijke uitdaging aan het licht: de ondersteuning van flexibiliteit in protocolstapels tijdens ontwerp, ontwikkeling en uitvoering, en dit afgestemd op applicatie- en/of netwerkspecifieke karakteristieken.*

## 1.3 Flexibele protocolstapels

De uitdaging om flexibele protocolstapels te ontwikkelen vloeit voort uit twee belangrijke tekortkomingen in traditionele protocolstapelsoftware. Ten eerste worden protocolstapels doorgaans opgebouwd uit grote modules (protocollagen bijvoorbeeld), wat het testen en aanpassen van de stapel bemoeilijkt. Ten tweede kan het gedrag van een protocolstapel dikwijls niet meer aangepast worden aan de specifieke en veranderende omstandigheden tijdens de uitvoering.

Er is daarom nood aan ondersteuning van het software-ontwikkelingsproces op vlak van ontwerp en software-architectuur. Traditionele software-ontwikkelingsparadigma's (zoals object- [Boo94] en componentgeoriënteerd programmeren [Szy98] of open implementatie [KLL+97]) schrijven weliswaar richtlijnen voor om tot aanpasbare software te komen, maar deze richtlijnen worden zelden afgedwongen tijdens het ontwikkelingsproces, wat leidt tot moeilijk aanpasbare software [LH89].

Om tot flexibele protocolstapels te komen concentreren we ons op drie onderzoekspistes: zelfaanpasbaarheid, testen, en raamwerkoptimalisatie.

Daarnaast zijn er nog veel andere pistes, maar die vallen buiten het kader van deze thesis. Zo kan aanpasbaarheid ook bekeken worden vanuit functioneel perspectief (bijvoorbeeld het weglaten, vervangen, of toevoegen van componenten). Verder kan het beheer van protocolstapels uitgebreid worden naar meer dan één machine in plaats van enkel lokale aanpassingen toe te laten. Een probleem in het netwerk kan bijvoorbeeld omzeild worden door tegelijkertijd in verschillende protocolstapels een bepaalde component toe te voegen, te verwijderen, of te vervangen. Tenslotte is beveiliging van zulke aanpassingen cruciaal in een realistische netwerkomgeving.

## 1.4 Belangrijkste bijdragen

De voornaamste bijdrage van deze thesis is dat het een consistent antwoord biedt op de drie geformuleerde onderzoeksuitdagingen: zelfaanpasbaarheid, testen, en raamwerkoptimalisatie. De tekst refereert aan deze oplossing als DiPS+ (Distrinet Protocol Stack+), aangezien ze verder bouwt op het reeds bestaande DiPS raamwerk [Mat99]. De bijdragen in elk van de drie onderzoekspistes zijn als volgt samen te vatten:

Het DMonA (DiPS+ Monitoring Architectuur) systeem zorgt voor zelfaanpasbaarheid wat betreft de toekenning van processorkracht binnen in de protocolstapel. DMonA laat toe om problemen te detecteren, een oplossing te

suggereren, en deze oplossing toe te passen in het onderliggende DiPS+ systeem.

Het DiPSUnit testraamwerk ondersteunt de ontwikkeling en uitvoering van testen. DiPSUnit profiteert van de ontwerpbeslissingen die genomen zijn in DiPS+, zodat programmeurs op een eenvoudige manier complexe protocolstapels kunnen testen. Zowel elementaire bouwstenen kunnen getest worden, als een protocollaag of een volledige protocolstapel. Daarenboven vereenvoudigt DiPSUnit het simuleren van specifieke netwerkomstandigheden, zoals het verloren gaan van specifieke pakketten.

Tenslotte stelt de thesis een optimalisatiebeschrijving voor die toelaat om raamwerkondersteuning automatisch weg te filteren op die plaatsen waar ze overbodig is. Indien de omstandigheden veranderen tijdens de uitvoering kunnen optimalisaties ongedaan gemaakt worden zodat DiPS+ raamwerkondersteuning opnieuw gebruikt kan worden. Op deze manier kan een evenwicht bereikt worden tussen flexibiliteit enerzijds en performantie anderzijds.

De kracht van deze thesis ligt erin dat deze drie onderzoekspistes niet door aparte oplossingen aangepakt worden, maar dat ze kunnen terugvallen op een gemeenschappelijk componentenraamwerk [SN99] en software-architectuur [SG96] (i.e. DiPS+).

## 1.5 Overzicht

De tekst kan in drie delen worden opgesplitst. Hoofdstuk 2 biedt de nodige achtergrond bij de software-ontwikkelingstechnieken die gebruikt zullen worden in de rest van de tekst. Daarenboven wordt het doctoraatsonderzoek gesitueerd binnen de ruimere onderzoekscontext. De volgende vier hoofdstukken (3, 4, 5 en 6) beschrijven de kern van het onderzoek uitgevoerd in het kader van deze thesis. Het derde deel bestaat uit twee hoofdstukken. Hoofdstuk 7 vergelijkt DiPS+ met de belangrijkste raamwerken voor protocolstapels die de laatste jaren ontwikkeld zijn. Hoofdstuk 8 formuleert conclusies en toont de relevantie van DiPS+ aan voor protocolstapels en andere subsystemen zoals randapparaatsturingsprogramma's (device drivers) en bestandensystemen.

# 2 Achtergrond

Dit hoofdstuk identificeert de belangrijkste problemen in traditionele protocolstapels, en positioneert de thesis binnen een ruimer onderzoekskader. Hoewel sommige problemen en karakteristieken toepasbaar zijn op software in het algemeen, zal deze thesis ze toch telkens benaderen vanuit het standpunt van protocolstapels.

## 2.1 Sleutelproblemen en eigenschappen van protocolstapels

### 2.1.1 Software-ontwerp en -architectuur

**Zwakke scheiding van belangen.** Programmeurs worden niet alleen geconfronteerd met de functionele vereisten van het systeem (i.e. wat het systeem moet doen), maar ook met niet-functionele vereisten (i.e. hoe het wordt gedaan). Programmeurs van protocolstapels worden dikwijls geconfronteerd met gelijktijdigheid, performantie, testbaarheid van de broncode, of haar robuustheid. Jammer genoeg zit de niet-functionele code dikwijls totaal verspreid over de functionele code [KLM+97], wat de controle en implementatie ervan bemoeilijkt, en programmeurs afleidt van het functionele programmeerwerk.

**Geen hiërarchische visie.** Al te dikwijls ontbreekt het ontwikkelaars aan een duidelijke software-architectuur. Een goede software-architectuur moet toelaten om de algemene structuur en de werking van het systeem te begrijpen, en moet voldoende nauw aansluiten bij het ontwerp zodat men makkelijk kan afwisselen tussen een architectuur- en een ontwerpperspectief [GP95].

**Beperkte mogelijkheden voor hergebruik.** De beperkte aandacht voor algemene, steeds weerkerende patronen leidt tot ad-hoc software die moeilijk herbruikbaar is in een andere context of in een volgende versie van het systeem met lichtjes gewijzigde functionaliteit [Joh97]. Broncode opsplitsen in duidelijk afgelijnde modules om hergebruik te stimuleren is enkel effectief wanneer modules voldoende klein zijn om subtiele veranderingen mogelijk te maken.

### 2.1.2 Buitengewone vereisten

**Beperkte aanpasbaarheid aan de omstandigheden en vereisten.** Interne implementatiedetails worden veelal afgeschermd van applicaties, wat verhindert dat ze afgestemd worden op applicatie- of omgevingsspecifieke vereisten. Dit laatste wordt bijzonder relevant wanneer toestellen met zeer uiteenlopende karakteristieken voorzien worden van een protocolstapel (zoals GSM's, PDA's, voertuigen met draadloze netwerkverbindingen, enz.).

**Focus op performantie.** Het streven naar optimale performantie van protocolstapels gaat vaak gepaard met twee problemen. Ten eerste is het zeer moeilijk om de omstandigheden te voorspellen waarin een protocolstapel moet functioneren. Dit maakt dat statisch geconfigureerde systemen zeer performant zijn voor de doorsnee applicatie, maar minder voor applicaties met uitzonderlijke vereisten (zoals multimedia-applicaties). Ten tweede hebben optimalisaties dikwijls subtiele veranderingen in de broncode tot gevolg, wat de code onverstaanbaar en foutgevoelig maakt.

**Onvoorspelbare belasting.** Protocolstapels moeten kunnen weerstaan aan extreem variërende belastingen van inkomende pakketjes, en daarom voldoende intelligent zijn om te detecteren wanneer hun limiet bijna bereikt is. Een netwerk van servers die de toevloed aan pakketten in parallel kunnen verwerken is geen afdoende oplossing, omdat dergelijk netwerk geen piekbelastingen kan opvangen op een kost-efficiënte manier.

## 2.2 Onderzoeksuitdagingen

**Zelfaanpasbaarheid.** Het onderzoek naar zelfaanpasbaarheid in deze thesis concentreert zich op het voorzien van een software-architectuur die toelaat om het niet-functionele gedrag van een protocolstapel te controleren en bij te sturen. Concreet behandelt het geanticipeerde aanpassingen van de verdeling van de beschikbare processorkracht tijdens de uitvoering.

**Testen.** Wat testen betreft concentreert de thesis zich op het voorzien van een componentenraamwerk dat de ontwikkeling van testbare software ondersteunt. Hiertoe moeten softwarecomponenten ontworpen worden als testbare entiteiten, en de onderliggende principes die leiden tot de ontwikkeling van testbare componenten moeten afgedwongen worden door het raamwerk. Daarnaast moeten programmeurs een testraamwerk ter beschikking krijgen dat toelaat om op een efficiënte en effectieve manier componenten, protocollagen en protocolstapels te testen als een zwarte doos ("black-box").

**Raamwerkoptimalisatie.** Om de tegenstrijdige vereisten van aanpasbaarheid en performantie te verzoenen moeten het onderliggende raamwerk en de architectuur een "black-box" optimalisatie van de protocolstapel toelaten. Taalondersteuning moet aangeboden worden om optimalisaties te beschrijven op een hoog niveau, en het onderliggend raamwerk moet in staat zijn deze beschrijving te verwerken tot een geoptimaliseerde versie. Bovendien moeten de optimalisaties zeer lokaal en instantiespecifiek doorgevoerd, en indien nodig ongedaan gemaakt kunnen worden.

# 3 Het DiPS+ componentenraamwerk

Dit hoofdstuk beschrijft de basis van ons onderzoek, het DiPS+ componentenraamwerk. DiPS+ breidt het DiPS (Distrinet Protocol Stack [Mat99]) raamwerk uit en past het aan op cruciale plaatsen. Deze veranderingen ondersteunen niet alleen softwareprogrammatie, maar het gehele software-ontwikkelingproces [MMWV01], en in het bijzonder wat betreft aanpasbaarheid, testen en raamwerkoptimalisatie.

Sectie 3.1 identificeert eerst de belangrijkste objectieven die DiPS+ moet vervullen, en toont aan waarom de originele versie van DiPS+ hieraan niet kan voldoen. Sectie 3.2 stelt de DiPS+ architectuur voor; Secties 3.3 en 3.4 concentreren zich op het ontwerp van respectievelijk het data- en het beheersvlak. De belangrijkste conclusies worden samengevat in Sectie 3.5.

## 3.1 Van DiPS naar DiPS+

### 3.1.1 Objectieven

Het doel van DiPS+ bestaat erin een antwoord te bieden op de belangrijke tekortkomingen waarmee ontwikkelaars van protocolstapels geconfronteerd worden. Dit doel kan vertaald worden in het aanbieden van de volgende drie kenmerken.

**Uniformiteit.** Systemen moeten samengesteld worden uit fijnkorrelige entiteiten, die gegroepeerd kunnen worden tot één geheel. Nochtans moeten entiteiten van verschillende granulariteit op eenzelfde uniforme manier gebruikt, gecontroleerd, en getest kunnen worden.

**Duidelijke mapping tussen architectuur en ontwerp.** Een duidelijke overeenkomst tussen architectuur en ontwerp maakt het systeem makkelijker begrijp- en beheersbaar. Verschillende aspecten, zoals functionaliteit, interactie tussen componenten, en gelijktijdigheid moeten daarom als aparte entiteiten aangeboden worden in zowel architectuur als ontwerp. Zo kunnen softwareontwikkelaars zich concentreren op één aspect tegelijkertijd.

**Controleerbaarheid.** Twee belangrijke aspecten, gelijktijdigheid en de datastroom doorheen de protocolstapel, moeten gecontroleerd kunnen worden (als aparte entiteiten). Gelijktijdigheidscontrole houdt in dat de beschikbare processorkracht zo optimaal mogelijk verdeeld wordt over het systeem, afhankelijk van de belasting, het type van inkomende aanvragen, de gevraagde kwaliteit, enz. Het controleren van de datastroom, van zijn kant, houdt in dat datapakketjes kunnen onderschept en geïnterpreteerd worden in een beheerssysteem, dat op basis van deze informatie de onderliggende protocolstapel kan bijsturen.

### 3.1.2 Tekortkomingen in DiPS

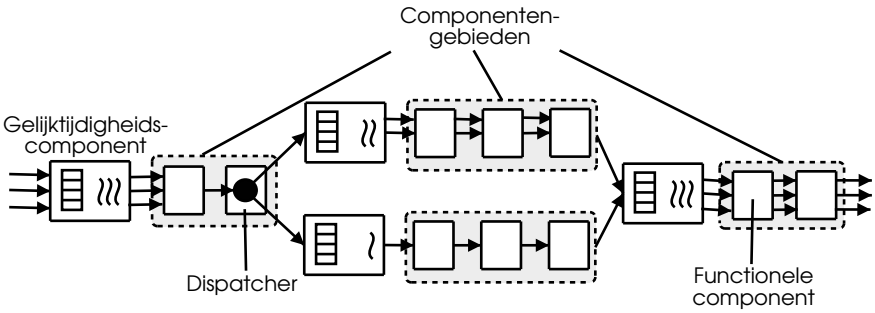
Om de gestelde objectieven te realiseren moet het bestaande DiPS raamwerk en de bijhorende architectuur aangepast en uitgebreid worden. Ten eerste moet het ontwerp van de basisabstracties in DiPS geüniformeerd worden zodat een consistente manier van eenheids-, integratie-, en systeemtesten kan aangeboden worden. Ten tweede moeten de in- en uitgangen van een component expliciet aanwezig zijn, niet alleen in het ontwerp maar ook in de architectuur. Dit laat een beheerssysteem toe om op een transparante manier informatie uit de protocolstapel te onderscheppen. Tenslotte moet de DiPS architectuur uitgebreid worden met een beheerslaag die verantwoordelijk is voor de controle en, indien nodig, bijsturing van het onderliggende systeem.

## 3.2 De architectuur van DiPS+

The DiPS+ architectuur is gelaagd: het verwerken van netwerkpakketjes gebeurt in het datavlak, de controle en het beheer van de onderliggende protocolstapel situeert zich in het beheersvlak. Elk van deze twee vlakken wordt voorgesteld door een combinatie van architecturale stijlen zoals de “pipe-en-filter”, de “blackboard”, en de gelaagde stijl [SG96].

De “pipe-en-filter” stijl in DiPS+ identificeert entiteiten zoals componenten (functionele blokken), connectoren (verbindingen tussen componenten), dispatchers (om een pakketstroom op te splitsen in deelstromen) en events (om een signaal te sturen naar meerdere componenten tegelijk).

De “blackboard” stijl laat componenten toe op een anonieme manier met elkaar te communiceren. In DiPS+ kan iedere component een pakket aanvullen




Figuur 2: Voorbeeld van een componentenpijplijn in DiPS+ met een dispatcher die de pijplijn in twee parallelle componentengebieden opsplitst.

met meta-informatie die geïnterpreteerd wordt door een andere component verderop in de pijplijn, zonder dat deze laatste moet weten van welke component de informatie komt.

Tenslotte worden gelijktijdigheidscomponenten aangeboden die in staat zijn om pakketten tijdelijk op te slaan in een buffer, en waarmee een verzameling van threads geassocieerd is. Aan elke thread is een selectiecriterium gekoppeld dat bepaalt welk pakket uit de buffer als volgende behandeld zal worden. Een gelijktijdigheidscomponent samen met de componenten erachter wordt in DiPS+ aangeduid als een "componentengebied" (zie Figuur 2).

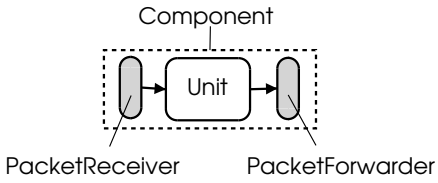
Gelijktijdigheidscomponenten bieden twee belangrijke voordelen. Ten eerste, door dergelijke componenten te verspreiden over de pijplijn kunnen pakketselectiecriteria rekening houden met de informatie die op dat moment beschikbaar is in een pakket. Deze informatie is niet noodzakelijk beschikbaar aan het begin van de pijplijn, wat een classificatie op basis van prioriteit op die plaats verhindert. Door zo een classificatie binnen in de pijplijn te maken kan de beschikbare processorkracht op een meer intelligente manier toegekend worden aan de belangrijkste pakketten. Het tweede voordeel is dat er niet alleen onderscheid gemaakt kan worden tussen pakketten, maar ook tussen belangrijke en minder belangrijke taken (i.e. componentengebieden).

## 3.3 Het ontwerp van het datavlak

### 3.3.1 Pakketten en componenten

De basisbouwstenen van het DiPS+ componentenraamwerk zijn `Packet`, `Unit`, `PacketReceiver` (PR), `PacketForwarder` (PF) en `Component`. Een `Packet` object laat toe om informatie uit te wisselen tussen componenten volgens de "blackboard" communicatiestijl. Een pakket in DiPS+ encapsuleert daartoe zogenaamde meta-informatie, naast de pure data en de protocolhoofdingen.

De PR en de PF vormen respectievelijk de ingang en de uitgang van een DiPS+ component. Deze laten toe om pakketten te onderscheppen naar het beheersvlak, en ontkoppelen bovendien `Units`. Een unit stelt de kern voor van




Figuur 3: Een DiPS+ component met een `PacketReceiver`, een `Unit`, en een `PacketForwarder`.

een DiPS+ component. Een unit, samen met zijn PR en PF vormt de DiPS+ `Component` (zie Figuren 3 en 4).

### 3.3.2 Verschillende types van units

Elke DiPS+ unit biedt dezelfde vaste interface aan: `incomingPacket(Packet p)`. Om het onderscheid tussen verschillende units en om de link met de entiteiten in de DiPS+ architectuur te benadrukken, worden drie types units onderscheiden: `FunctionalUnit`, `DispatchingUnit`, en `ConcurrencyUnit`. Het eerste type beschrijft een afgelijnd stuk functionaliteit, zoals het aanmaken van een protocolhoofding, encrypteren van gegevens, of fragmenteren van pakketten. Functionele units worden meestal geschreven door een programmeur of geselecteerd uit een bibliotheek van herbruikbare componenten. Het tweede type, dispatching unit, kan inkomende pakketten naar verschillende uitgangen sturen, afhankelijk van de meta-informatie die gekoppeld is aan het pakket. Op deze manier wordt het gevolgde pad van een pakket doorheen de protocolstapel beïnvloed. Gelijktijdigheidsunits, tenslotte, laten toe om pakketten te bufferen en ze (later) op een gecontroleerde manier te verwerken door een onafhankelijke pakketbehandelaar.

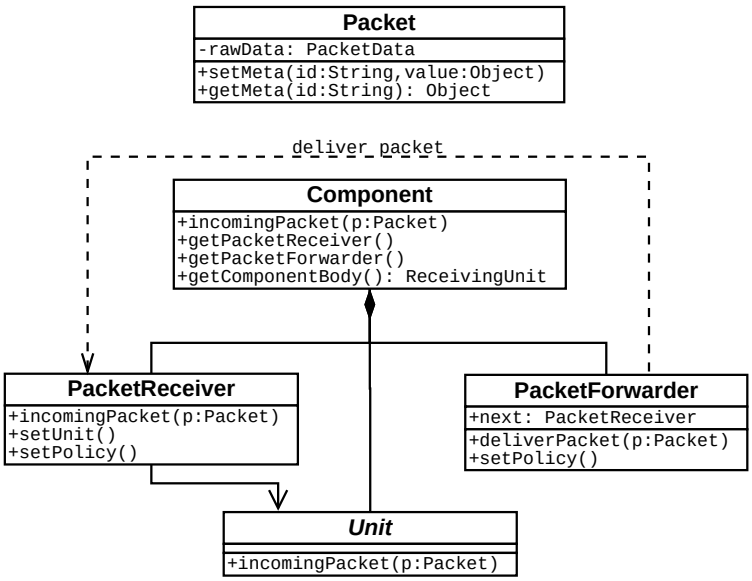
## 3.4 Het ontwerp van het beheersvlak

### 3.4.1 Onderscheppen van pakketten

Om pakketten op een transparante manier te kunnen onderscheppen vanuit het datavlak naar het beheersvlak, zijn PR's en PF's voorzien van een `Policy`. Een dergelijke policy biedt de interface naar het beheersvlak en beschrijft wat er met elk binnenkomend pakket moet gebeuren. De afhandeling van een pakket in het beheersvlak wordt verzorgd door een pijplijn van `ManagementModule`s die elk een specifieke taak vervullen (bvb. het aantal pakketten tellen, de lengte van de pakketten bewaren, pakketten markeren, enz.). Door de management modules aan te passen kan de behandeling van elk pakket makkelijk veranderd worden.

### 3.4.2 Gelijktijdigheidsbeheer

Om het gedrag van een gelijktijdigheidscomponent te kunnen aanpassen vanuit het beheersvlak, biedt de gelijktijdigheidsunit, naast de `incomingPacket(Packet`




Figuur 4: Het ontwerp van twee essentële DiPS+ abstracties: component en pakket.

p) interface, ook een management-interface. Op deze manier krijgt het beheersvlak op een gecontroleerde manier toegang tot de interne toestand van een gelijktijdigheidsunit. Via de management-interface kan, bijvoorbeeld, de strategie die bepaalt wat moet gebeuren als een interne buffer overloopt, makkelijk aangepast worden aan de specifieke vereisten van een bepaalde applicatie.

## 3.5 Besluit

### 3.5.1 Bijdragen van DiPS+

Dit hoofdstuk beschreef hoe het DiPS componentenraamwerk en de bijhorende architectuur zijn aangepast en uitgebreid. De eerste bijdrage wordt geleverd door het uniforme ontwerp van functionele, dispatching- en gelijktijdigheidsunits. De tweede bijdrage zijn de expliciete in- en uitgangen (PR en PF) in het ontwerp van een DiPS+ component. Via de policy kunnen pakketten op een transparante manier afgeleid worden naar het beheersvlak. Ten derde biedt DiPS+ een beheersvlak dat volledig gescheiden is van het datavlak, zodat onderliggende protocolstapel transparant gecontroleerd en bijgestuurd kan worden.

### 3.5.2 Samenvatting

Tenslotte gaan we na op welke manier DiPS+ de in Sectie 3.1.1 voorgestelde objectieven bereikt of tegemoet komt. Het uniforme ontwerp van componenten

laat toe verschillende types componenten (functionele, dispatching of gelijktijdigheid) op gelijke voet te behandelen. De afzonderlijke vlakken voor data en beheer zorgen bovendien voor een duidelijke mapping tussen architectuur en ontwerp. Ook de architecturale componenten komen overeen met de units. Tenslotte kan een DiPS+ protocolstapel makkelijk gecontroleerd worden door specifieke management modules in te pluggen in het beheersvlak.

# 4 Zelfaanpasbaar systeemgedrag

De ontwikkeling van aanpasbare protocolstapels speelt in op drie recente tendensen: de groeiende afhankelijkheid van netwerkdiensten, het dynamisch karakter van netwerkverkeer, en de grote variëteit aan toestellen die aangesloten kunnen worden op een netwerk.

Dit hoofdstuk beschrijft DMonA, de DiPS+ Monitor Architectuur. DMonA is een concrete invulling van het beheersvlak in DiPS+ en laat toe performantieknelpunten te detecteren, oplossingen voor te stellen om zulke knelpunten op te lossen, en deze oplossingen door te voeren tijdens de uitvoering [MDJ+02]. Bovendien verandert het DiPS+ van een manueel in een zelfaanpasbaar systeem. Met andere woorden, via DMonA kan een protocolstapel zelf (in plaats van een administrator bijvoorbeeld) pakketbehandelaars verplaatsen en toekennen aan gelijktijdigheidsunits.

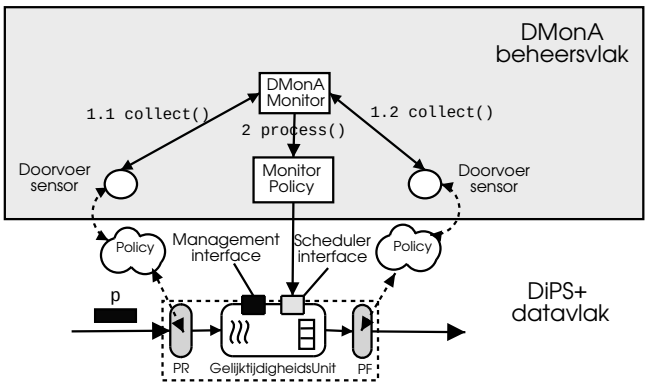
DMonA is ontworpen als een algemeen raamwerk dat kan toegepast worden voor beheer en controle. Ter validatie past dit hoofdstuk DMonA toe op twee complementaire benaderingen voor het beheer van overbelasting. Enerzijds wordt overbelasting aangepakt door pakketbehandelaars optimaal te spreiden over de verschillende componentengebieden in de protocolstapel. Anderzijds worden inkomende pakketten gecontroleerd en geclassificeerd, zodat de belangrijkste pakketten voorrang kunnen krijgen. Dit kan door de selectiecriteria van de pakketbehandelaars te baseren op de meta-informatie die beschikbaar wordt tijdens de verwerking van een pakket.

## 4.1 De DMonA aanpak

DMonA steunt op de architectuur en het ontwerp van het datavlak in DiPS+. Enerzijds wordt informatie uit het datavlak verzameld via policies die geassocieerd zijn met de PR en PF. Anderzijds laat de management interface van de gelijktijdigheidsunits toe om het datavlak bij te sturen.

DMonA zelf is opgevat als een algemeen raamwerk, en biedt dus zelf geen specifieke detectie- of oplossingsstrategieën. Het is daarentegen een open platform waaraan specifieke policies en algoritmes kunnen toegevoegd worden. Het DMonA raamwerk zorgt er voor dat gegevens uit het datavlak worden verzameld en verwerkt, en dat de voorziene oplossingsstrategie wordt doorgevoerd wanneer een probleem zich voordoet (zie Figuur 5).

DMonA is opgebouwd rond vier basisklassen: de `Monitor`, de `Collector`, de `MonitorPolicy`, en de `TargetId`. De monitor is de actieve motor van DMonA en




Figuur 5: DMonA (DiPS+ Monitoring Architectuur) als een beheersvlak bovenop het DiPS+ datavlak.

beschrijft het algemene gedrag als een "Template" ontwerppatroon [GHJV95]: informatie verzamelen in de collectoren, en ter verwerking doorsturen naar de monitor-policy. De collectoren verwerken informatie die verzameld is in een `Collectable` object (een management module). Collectoren kunnen bijvoorbeeld de gemiddelde doorvoersnelheid berekenen, of de gemiddelde pakketgrootte, of ze kunnen zelf een patroon herkennen uit de beschikbare informatie in een collectable. De monitor-policy beslist op basis van de informatie in de collectoren of een aanpassing nodig is. Een dergelijke policy kan daartoe ondermeer vergelijken hoeveel pakketten toekomen aan een componentengebied en hoeveel er verwerkt kunnen worden gedurende een bepaald tijdsinterval. Een concrete monitor-policy kan bijvoorbeeld beslissen om een pakketbehandelaar te migreren van één gelijktijdigheidsunit naar een andere, of een extra pakketbehandelaar toe te kennen aan een gelijktijdigheidsunit, op voorwaarde dat er nog vrije systeembronnen beschikbaar zijn. De gelijktijdigheidsunits worden geïdentificeerd door hun `TargetId`.

## 4.2 Een casestudy: beheer van de systeembelasting

Belastingsbeheer kan op verschillende manieren benaderd worden. Vanuit een gelijktijdigheidsperspectief omvat belastingsbeheer het optimaal spreiden van de beschikbare processorkracht over de verschillende componentengebieden. Tezelfdertijd kan men belastingsbeheer ook aanpakken vanuit het perspectief van toegangscontrole. Hierbij wordt de pakketinvoer gecontroleerd en beperkt tot een haalbaar niveau. Dit kan door inkomende pakketten te classificeren op basis van hun bestemming, de geëncapsuleerde protocollen, of applicatiespecifieke vereisten meegegeven via meta-informatie. Op basis van die classificatie kunnen pakketten met een lage prioriteit afzonderlijk behandeld worden, of simpelweg weggegooid als de belasting te hoog wordt.

### 4.2.1 Allocatiemodellen voor threads

Multiprogrammatie is een algemeen bekende en aanvaarde techniek die toelaat om meerdere programma's (of processen) uit te voeren op één enkele processor, zodat het lijkt of al deze programma's tegelijk in uitvoering zijn. Multiprogrammatie biedt echter *virtueel* parallellisme, gezien er hoogstens één programma tegelijkertijd de processor kan gebruiken. Daarom gebruiken monoprocessorbesturingssystemen typisch een onderbreekbaar ("pre-emptive") selectie-algoritme (scheduling) dat er voor zorgt dat het actieve proces onderbroken wordt ten voordele van één van de andere processen.

Multiprogrammatie en onderbreekbare selectie-algoritmes verbergen het aspect gelijktijdigheid in het besturingssysteem, zodat het transparant wordt voor de toepassingen die erboven draaien. Hoewel die transparantie het voordeel biedt dat toepassingen er zich niets van moeten aantrekken, heeft het ook een groot nadeel: het gedrag van het onderliggende systeem kan niet aangepast worden aan applicatiespecifieke vereisten.

Twee traditionele modellen voor multiprogrammatie zijn het thread-per-aanvraag en het threadpoolmodel. Het eerste model associeert een nieuwe thread met elke nieuwe aanvraag. Omdat deze oplossing niet uitbreidbaar is naar een groot aantal parallelle threads, wordt het threadpoolmodel het meest frequent gebruikt. In dit model kan geput worden uit een beperkte verzameling threads.

Deze modellen zijn niet controleerbaar en aanpasbaar aan applicatiespecifieke vereisten om twee fundamentele redenen. Ten eerste laten ze niet toe om de stroom van aanvragen te inspecteren tijdens hun verwerking. Bijgevolg kan men selectiecriteria niet afstemmen op informatie die pas beschikbaar wordt tijdens de verwerking van een pakket. In beide modellen wordt een thread geassocieerd met een binnenkomende aanvraag, waardoor een selectiecriterium enkel rekening kan houden met de op dat moment beschikbare informatie. Ten tweede laten deze modellen geen controle over de uitvoering van een thread toe tijdens de verwerking van een aanvraag, wat bijvoorbeeld verhindert om de kern van een belastingsknooppunt te detecteren en op te lossen.

Dankzij DiPS+ kan men elk componentengebied associëren met een specifieke selectiestrategie, die gebruik kan maken van alle (meta-)informatie die op die plaats in de pijplijn beschikbaar is. Daarenboven laat de opsplitsing van de pijplijn in deeltaken (componentengebieden) enerzijds toe om knelpuntgebieden te detecteren. Een knelpuntgebied is een componentengebied waar veel meer pakketten toekomen dan er verwerkt kunnen worden gedurende een bepaalde periode. Anderzijds maken componentengebieden het ook mogelijk om belangrijke van minder belangrijke taken te onderscheiden, wat bijzonder relevant kan zijn voor parallelle componentengebieden.

Ondanks de voordelen die de DiPS+ aanpak biedt, blijft het essentieel dat het systeem zichzelf kan aanpassen aan de gegeven omstandigheden. Het invoeren van componentengebieden zorgt voor meer controle en aanpasbaarheid, maar is pas efficiënt als de beschikbare rekenkracht optimaal verdeeld wordt over de verschillende gebieden. Daarenboven is het belangrijk dat het systeem

zelf kan detecteren wanneer het haar verwerkingsgrens bereikt, zodat het overbelasting op een intelligente manier kan aanpakken. Het doel van DMonA is om deze twee aspecten, gelijktijdigheid en toegangscontrole, te ondersteunen.

### 4.2.2 Balanceren van threads

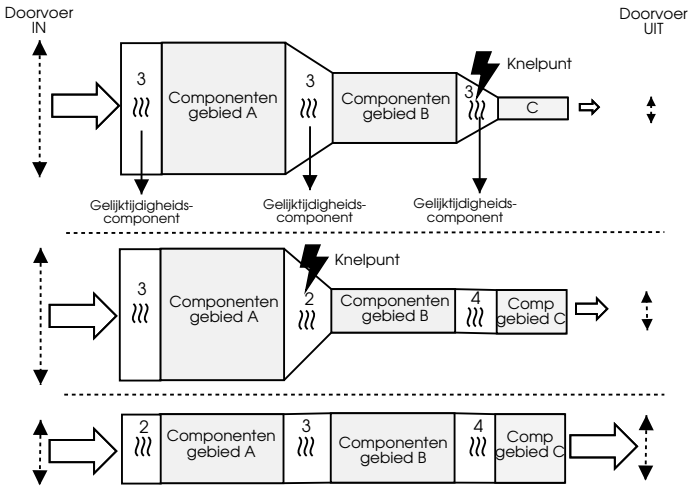
Figuur 6 toont drie deeltaken als een pijplijn samengesteld uit componentengebieden, elk met een specifieke diameter die de doorvoer voorstelt. De figuur illustreert de werking van DMonA om rekenkracht zo optimaal mogelijk te spreiden over verschillende componentengebieden. In het voorbeeld wordt verondersteld dat negen pakketten per tijdseenheid verwerkt kunnen worden, wanneer de pijplijn niet opgesplitst zou zijn in componentengebieden. Bijgevolg is het streefdoel van DMona om een doorvoer te bereiken van negen pakketten per tijdseenheid, wat enkel benaderd kan worden door de rekenkracht optimaal te verdelen over de componentengebieden. Met andere woorden, gebieden die meer rekenkracht vereisen moeten relatief gezien meer pakketbehandelaars toegekend krijgen. Het voorbeeld in Figuur 6 verdeelt de rekenkracht (negen pakketbehandelaars) in eerste instantie evenredig over de drie gebieden, maar gebied C vereist meer rekenkracht dan de andere gebieden. De monitor-policy in DMonA zorgt er daarom voor dat pakketbehandelaars verplaatst worden van minder naar meer belaste gebieden.

# 5 Testen in DiPS+

In de voorbije decennia zijn er verschillende methodes en toepassingen ontwikkeld om het testen van objectgeoriënteerde software te ondersteunen [Bin99]. Componentgeoriënteerde software zorgt echter voor nieuwe uitdagingen. Om componenten efficiënt en betrouwbaar te kunnen gebruiken in diverse producten en omgevingen, is er nood aan een manier van werken die niet alleen toelaat om de componenten makkelijk te testen, maar die ontwikkelaars ook verplicht om ze zodanig te ontwerpen dat ze makkelijk te testen zijn [Wey98].

De complexiteit van protocolstapels maakt het testen ervan moeilijk, vervelend en tijdrovend. Dit heeft grotendeels te maken met de manier waarop traditionele protocolstapels ontworpen zijn, nl. als een monolithische blok code met meerdere parallelle threads. Ten eerste maakt het monolithisch karakter van de code het moeilijk tot onmogelijk om specifieke bouwstenen van de protocolstapel (een protocollaag bijvoorbeeld) te isoleren van de rest. En zelfs een gemodulariseerd ontwerp garandeert niet dat individuele modules ontkoppeld en onafhankelijk zijn. Ten tweede wordt het testen van een protocolstapel bemoeilijkt wanneer verschillende threads actief kunnen zijn, omdat testscenario's in dat geval moeilijk te reproduceren zijn. Idealiter wordt eerst de functionele code getest, en wordt pas later, wanneer de code stabiel is, de gelijktijdigheidscode toegevoegd.

Het doel van dit hoofdstuk is om enerzijds aan te tonen dat het ontwerp van DiPS+ componenten leidt tot makkelijk testbare software, en anderzijds




Figuur 6: Intuïtieve illustratie van het verdelen van de beschikbare processorkracht over meerdere componentengebieden om een lokaal performantieknelpunt op te lossen. In het getoonde voorbeeld vereist gebied A 20% van de processortijd, gebied B 30%, en gebied C 50%. In de bovenste figuur zijn de threads evenwichtig verdeeld over de drie componentengebieden (i.e. 33% van de beschikbare rekenkracht voor elk gebied). Gezien gebied C meer rekenintensief is dan A en B, vormt het een performantieknelpunt. De middelste figuur toont hoe DMonA één pakketbehandelaar verhuist naar gebied C. Aangezien deze verhuis een nieuw knelpunt in gebied B creëert, verplaatst DMonA een pakketbehandelaar van A naar B. Hierdoor wordt een quasi-optimale verdeling bereikt voor de momentele belasting.

een testraamwerk (DiPSUnit [MWJV, MWJV02]) voor te stellen dat DiPS+ ontwikkelaars ondersteunt bij het schrijven en uitvoeren van testen.

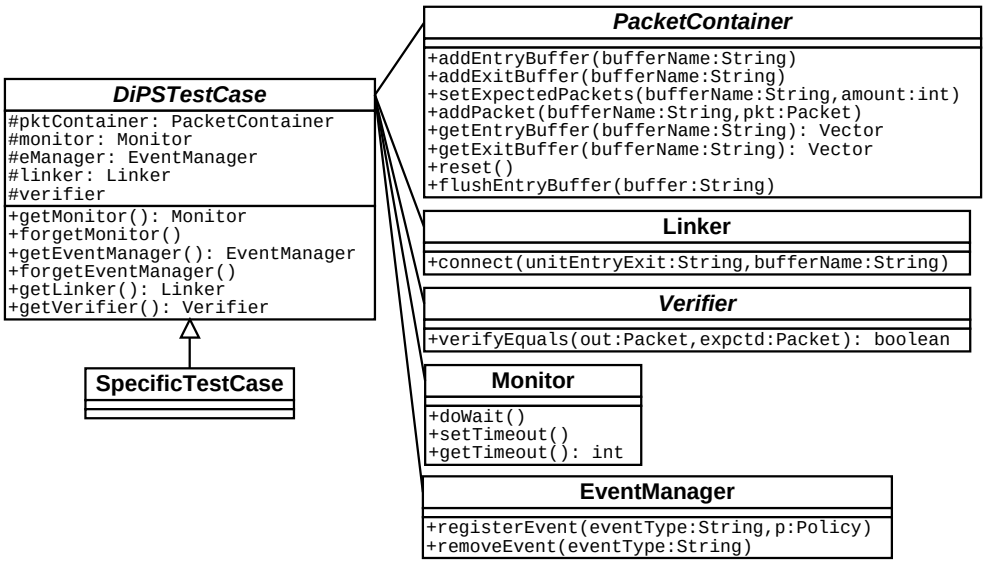
## 5.1 De testproblematiek vanuit een DiPS+ perspectief

**Controleerbaarheid.** De controleerbaarheid van een softwarecomponent (of een programma) geeft aan hoe makkelijk de in- en uitvoer, de operaties, en het gedrag ervan gecontroleerd kunnen worden. DiPS+ software is controleerbaar door de pakketklasse, meta-informatie, en de architectuurbeschrijving. Pakketten worden aangenomen, verwerkt, en doorgestuurd op een voorgedefinieerde manier en via een vaste interface, wat de controle over de in- en uitvoer van een component vanzelfsprekend maakt. Via meta-informatie kan het gedrag van een component beïnvloed worden van buitenaf. De beschikbaarheid van een architectuurbeschrijving naast de effectieve broncode laat toe om de configuratie van het geteste systeem te controleren en aan te passen, onafhankelijk van de testcode.

**Observeerbaarheid.** Het observeren (en testen) van een stuk software (bijvoorbeeld een component, een protocollaag, of een protocolstapel) houdt in dat de software geïsoleerd kan worden van de rest en van de omgeving (bijvoorbeeld het netwerk). In DiPS+ kan men componenten isoleren via connectiecontrole, controle-events, en meta-informatie. Connectiecontrole wordt ondersteund door pakketten altijd via het DiPS+ raamwerk (i.e. via de expliciete in- en uitgang van een component) te sturen. Hierdoor hebben units geen directe referentie naar mekaar, wat een testraamwerk toelaat om een component te isoleren van de andere componenten, zelf invoer te genereren en de uitvoer op te vangen. Controle-events ontkoppelen van nature hun zender van de ontvanger(s). Bijgevolg kan men een ontvanger vervangen door een zogenaamde stub [Bin99], die het gedrag van de echte component simuleert. Meta-informatie laat componenten toe om op een anonieme manier informatie te verspreiden. Zo blijven componenten onafhankelijk van elkaar wat ook weer toelaat om ze te isoleren van de rest.

**Uniforme aanpak.** Uniformiteit slaat enerzijds op het feit dat verschillende types van componenten op een consistente manier behandeld kunnen worden. Anderzijds houdt uniformiteit in dat unit-, integratie-, en systeemtesten door een zelfde aanpak ondersteund worden. Het ontwerp van DiPS+ ondersteunt dergelijke uniforme aanpak door drie aspecten: het uniforme ontwerp van alle unit types, de vaste interface die aangeboden wordt door deze units, en de scheiding tussen de pakket- en de beheersinterface.

**Incrementele aanpak.** In het ideale geval is testen een zeer incrementeel proces [Amb02, Bin99]. Daarenboven moeten alle beschikbare testen opnieuw worden uitgevoerd wanneer nieuwe code wordt toegevoegd om er zeker van te zijn dat de nieuwe code geen fouten introduceert in de bestaande [Bec99]. De incrementele aanpak moet ook toegepast worden op de broncode zelf door eerst de functionele code te testen, en daarna pas de code voor gelijktijdigheid toe te voegen. DiPS+ maakt een incrementele aanpak mogelijk door het systeem op te splitsen in fijnkorrelige units, en door gelijktijdigheidscode te isoleren in




Figuur 7: Het ontwerp van het DiPSUnit testraamwerk.

gelijktijdigheidsunits.

**Omgevingscontrole.** De omgeving (het netwerk) is zeer belangrijk voor het testen van een protocolstapel. Een protocolstapel testen in een echte netwerkomgeving is echter zeer moeilijk omdat het gedrag van het netwerk onvoldoende kan gecontroleerd worden (bijvoorbeeld verlies van pakketten, vertraging, duplicatie, enz.). Daarom is het beter om het gedrag van het netwerk te simuleren zodat specifieke fouten op een controleerbare manier geproduceerd kunnen worden. Een netwerksimulatie moet ook toelaten om pakketten te manipuleren in eender welke laag in de protocolstapel, zodat specifieke fouten in elk protocol op een eenvoudige manier gesimuleerd kunnen worden. DiPS+ ondersteunt omgevingscontrole door de aparte architectuurbeschrijving en doordat componenten onafhankelijk zijn van elkaar.

## 5.2 Het DiPSUnit testraamwerk

Het DiPSUnit testraamwerk (Figuur 7) laat toe om op een uniforme manier unit-, integratie- en systeemtesten te ontwikkelen en uit te voeren.

**Testen van componenten.** De belangrijkste abstracties voor het opzetten en uitvoeren van een test zijn gemodelleerd door drie klassen: de `PacketContainer`, de `Linker`, en de `Verifier`. De `PacketContainer` beheert de buffers die dienen om testpakketten te bewaren (invoerbuffers) en de resulterende pakketten op te vangen (uitvoerbuffers). Op die manier kan een test verschillende testpakketten aanbieden aan verschillende ingangen van de geteste component. De `Linker` zorgt er voor dat een component, een protocollaag of een ganse protocolstapel gekoppeld wordt aan het DiPSUnit raamwerk. De `Verifier`

tenslotte laat toe om na te gaan of het verkregen pakket overeenkomt met het verwachte resultaat.

**Asynchroon gedrag.** Wanneer het geteste systeem asynchroon is opgebouwd (i.e. met een interne gelijktijdigheidsunit), moet een test in uitvoering expliciet wachten tot alle pakketten verwerkt zijn. Het blokkeren van een test gebeurt door de `Monitor` klasse, die aan elke uitgang van het geteste systeem controleert of het verwachte aantal pakketten is ontvangen.

**Controle-events.** Wanneer het geteste systeem controle-events uitstuurt, kan een `Policy` object geregistreerd worden bij de `EventManager`, samen met het event type waardoor de `Policy` geactiveerd wordt. De `Policy` simuleert het gedrag van de echte ontvanger of introduceert foutief gedrag om na te gaan hoe het systeem hierop reageert.

## 5.3 DiPSUnit in de praktijk

**Foutendetectie.** Het spreekt voor zich dat het gebruik van een testraamwerk helpt om fouten te detecteren in de geteste software. We kunnen echter drie niet-triviale conclusies trekken uit het gebruik van DiPSUnit. Ten eerste worden fouten zeer vroeg in het ontwikkelingsproces gedetecteerd. Dit komt doordat alle testen telkens opnieuw worden uitgevoerd wanneer een component wordt gewijzigd of een nieuwe component wordt toegevoegd. Ten tweede, door de testen herhaaldelijk uit te voeren kan ook de bron van de fout sneller ontdekt worden. Dit komt doordat het verschil met de vorige, werkende versie zeer klein is en de fout (of de oorzaak ervan) hoogstwaarschijnlijk in het toegevoegde stukje broncode zit. Ten derde laat DiPSUnit toe om het systeem in een consistente toestand te houden tijdens de ontwikkeling omdat elke verandering geverifieerd wordt door een steeds groeiende verzameling testen.

**Ervaringen.** Uit de praktijk blijkt DiPSUnit makkelijk te gebruiken en aan te leren. Zowel onderzoekers als studenten gebruiken DiPSUnit zonder noemenswaardige problemen. Blijkbaar krijgen ontwikkelaars meer vertrouwen om bestaande software aan te passen, vermits het uitvoeren van alle beschikbare testen onmiddellijk feedback geeft over de correctheid van het aangepaste systeem. Een ander merkwaardig fenomeen is dat programmeurs elkaar beginnen aan te moedigen om testen te schrijven nadat ze een fase van scepticisme overwonnen hebben. DiPSUnit werd gebruikt voor het testen van protocollen zoals TCP, UDP, IP, Ethernet en SIP. Onze ervaringen bevestigen ook dat DiPSUnit de trend naar zeer iteratieve software-ontwikkelingsmethodologieën ondersteunt (bijvoorbeeld de “agile” methodologie [Aoy98, Amb02]).

## 5.4 Besluit

Dit hoofdstuk stelde het DiPSUnit testraamwerk voor, en beschreef hoe DiPS+ er voor zorgt dat de geproduceerde software makkelijk te testen is. DiPSUnit biedt een uniforme testinfrastructuur, laat toe om asynchrone software te testen, biedt de mogelijkheid om controle-events te onderscheppen of te sturen, en laat toe om de netwerkomgeving te simuleren door pakketten te manipuleren op

verschillende niveaus in de protocolstapel. DiPSUnit bevestigt bovendien dat het ontwerp van DiPS+ componenten tot testbare software leidt. De combinatie van DiPSUnit en DiPS+ laat toe om complexe protocolstapels toch op een eenvoudige en efficiënte manier te testen.

# 6 Optimalisatie van het DiPS+ raamwerk

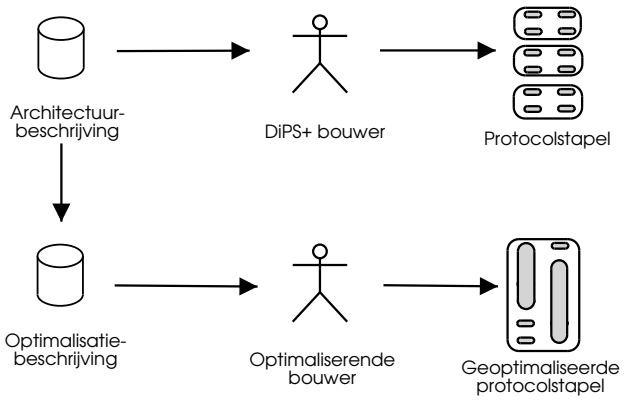
Dit hoofdstuk plaatst twee belangrijke maar dikwijls tegenstrijdige aspecten tegenover elkaar: performantie en dynamische aanpasbaarheid. Hoewel DMonA zeer effectief kan zijn om een maximale performantie te garanderen tijdens overbelasting, brengt het ook een deel overhead met zich die niet altijd kan gecompenseerd worden door het aanpasbare gedrag.

Overhead kan verschillende oorzaken hebben. Enerzijds kan het adaptief gedrag zelf de oorzaak zijn van overhead, bijvoorbeeld doordat het systeem moet gecontroleerd worden tijdens de uitvoering. Anderzijds kunnen het ontwerp en de architectuur impliciet overhead introduceren. De expliciete in- en uitgangen van een DiPS+ component bijvoorbeeld zorgen voor overhead, maar zijn nodig om het systeem te kunnen controleren.

De uitdaging ligt erin om een gulden middenweg te vinden tussen het gebruik van een raamwerk en architectuur die dynamische aanpassingen toelaten, en de overhead die dit met zich brengt. Daarom biedt DiPS+ naast de zuivere architectuurbeschrijving ook een optimalisatiebeschrijving, die een administrator toelaat om die plaatsen aan te duiden waar geen ondersteuning nodig is tijdens de uitvoering. Bijgevolg kan vanuit deze twee beschrijvingen een geoptimaliseerde protocolstapel gecreëerd worden. Deze optimalisaties concentreren zich louter op de structuur en niet op de functionaliteit van de protocolstapel.

## 6.1 Beperkingen van optimaliserende vertalers

Vandaag de dag zijn vertalers een zeer krachtig instrument om optimalisaties door te voeren. Sommige Just-In-Time (JIT) vertalers houden zelfs rekening met dynamische aanpassingen in de broncode [Sun, Sun03]. Ondanks hun grote voordelen kunnen zulke vertalers echter geen rekening houden met de specifieke eigenschappen van het gebruikte raamwerk, en kunnen ze geen onderscheid maken tussen de functionele code (de protocolstapel bijvoorbeeld) en het raamwerk zelf. Maar zelfs indien ze dit onderscheid konden maken, dan nog zouden vertalers niet geschikt zijn voor zeer lokale en instantiespecifieke aanpassingen. Technieken als AspectJ [Asp03] laten toe om optimalisaties apart te beschrijven in een zogenaamd Aspect dat dan later in de functionele code “geweefd” wordt. Hierdoor kunnen zeer lokale veranderingen en optimalisaties beschreven en uitgevoerd worden. Het is echter niet mogelijk om veranderingen aan specifieke instanties van een bepaalde klasse door te voeren, aangezien AspectJ, net als traditionele vertalers, typegeoriënteerd is (in plaats van instantiegeoriënteerd).




Figuur 8: Het scenario voor het beschrijven en bouwen van een standaard protocolstapel en een geoptimaliseerde versie ervan.

## 6.2 Optimalisatie volgens de DiPS+ aanpak

**Granulariteit.** Optimalisaties kunnen doorgevoerd worden op verschillende niveaus. Het hoogste niveau van optimalisaties zijn die tussen verschillende protocollagen. Indien geweten is dat er geen lagen zullen toegevoegd worden aan de protocolstapel, kan het selectiemechanisme dat beslist naar welke laag een pakket gestuurd moet worden vereenvoudigd worden. Ten tweede kan men de interne structuur van een protocollaag samensmelten als er tijdens de uitvoering geen controle nodig is. Ten derde kan een protocollaag ook gedeeltelijk geoptimaliseerd worden. Dit is bijvoorbeeld relevant wanneer alleen een gelijktijdigheidscomponent moet voorzien worden van expliciete in- en uitgangen en de andere componenten niet. Tenslotte kunnen optimalisaties zich concentreren op één specifieke instantie van een component. Dit gebeurt bijvoorbeeld wanneer niet alle gelijktijdigheidscomponenten beheerd moeten worden.

**Taalondersteuning.** Naast de beschrijving van de architectuur biedt DiPS+ ook de mogelijkheid om de gewenste optimalisaties te beschrijven (zie Figuur 8). Dit verlost systeemadministratoren van de moeilijke taak om optimalisaties manueel door te voeren.

**Optimaliserende bouwer.** Net zoals de DiPS+ bouwer een protocolstapel creëert op basis van een architectuurbeschrijving, wordt een bouwer voorzien die vanuit een optimalisatiebeschrijving een geoptimaliseerde stapel instantieert (zie Figuur 8).

## 6.3 Besluit

Dit hoofdstuk stelde een oplossing voor die de gulden middenweg zoekt tussen dynamische aanpasbaarheid en optimalisatie van het DiPS+ raamwerk. De twee belangrijkste bijdragen zijn een beschrijving voor lokale en instantiespecifieke optimalisaties en een geassocieerde bouwer.

# 7 Gerelateerd werk

Dit hoofdstuk bespreekt de vier meest relevante raamwerken voor het ontwikkelen van protocolstapels en vergelijkt deze met de DiPS+ aanpak wat betreft aanpasbaarheid, testen, en raamwerkoptimalisatie.

## 7.1 Raamwerken voor protocolstapels

**Scout.** Het Scout besturingssysteem [MMOP95] vertoont een gelaagde architectuur, vergelijkbaar met de lagen in het het DiPS+ datavlak, maar biedt geen fijnkorrelige componenten. Scout is ontwikkeld rond een communicatiegeoriënteerde "pad" abstractie [MP96]. Een pad stelt een communicatiekanaal doorheen een gelaagde structuur voor, en verlengt als het ware een netwerkverbinding doorheen de protocolstapel. Een pad encapsuleert de specifieke attributen van dergelijk communicatiekanaal en geeft toegang tot deze attributen over protocollagen heen. Een pad in Scout is vergelijkbaar met een DiPS+ pakket in combinatie met de bijhorende meta-informatie. DiPS+ componenten kunnen attributen doorgeven aan componenten verderop in de pijplijn, eventueel in een andere protocollaag. Op die manier laat een DiPS+ pakket een spoor na, vergelijkbaar met een pad in Scout.

**Click.** De Click router [Koh01, KMC+00] is gebaseerd op een ontwerp dat vergelijkbaar is met DiPS+. Een Click router is opgebouwd uit fijnkorrelige modules om pakketten te verwerken, "elementen" genaamd, vergelijkbaar met DiPS+ units. Een Click element heeft net zoals een DiPS+ component expliciete in- en uitvoerpoorten die respectievelijk pakketten aannemen van het vorige en doorgeven aan het volgende element. Elementen communiceren ofwel via een directe functie-oproep, ofwel indirect via een buffer die twee naburige elementen loskoppelt. Click ondersteunt twee manieren om een pakket te transporteren: al "duwend" of al "trekkend". Bij de eerste manier stuurt een element een pakket naar het volgende element, en wordt het pakket dus als het ware door het systeem geduwd. Bij de tweede manier vraagt een element aan de voorganger om een pakket door te geven, zodat het pakket door het systeem getrokken wordt. DiPS+ componenten duwen pakketten door de pijplijn, waarbij gelijktijdigheidscomponenten de pijplijn ontkoppelen en pakketten op een onafhankelijke manier verder sturen. Click elementen laten niet toe om informatie te verspreiden zoals een pad in Scout of meta-informatie in DiPS+.

**Cactus.** Het Cactus project biedt een geïntegreerd ontwerp- en implementatieraamwerk dat aanpasbare en fijnkorrelige kwaliteitsattributen ondersteunt met betrekking tot afhankelijkheid (dependability), reële tijd, en beveiliging in gedistribueerde systemen [Hil96, HS00] en netwerkprotocollen [Bha96, WHS01]. Het Cactus model bouwt protocollen op uit fijnkorrelige microprotocollen [Bha96] met een vaste interface, vergelijkbaar met DiPS+ units. Microprotocollen communiceren via events die onderschept en gedelegeerd worden via het Cactus raamwerk. DiPS+ biedt een combinatie van architecturale stijlen die de precieze werking van een samengesteld protocol makkelijker begrijpbaar maakt.

**Staged Event-Driven Architectuur (SEDA).** SEDA [WCB01, Wel02] biedt een architectuur ter ondersteuning van zwaar belaste concurrente systemen, hoofdzakelijk webservers. SEDA bouwt een systeem op uit zogenaamde "stages" die met elkaar kunnen communiceren via events. Elke stage bestaat uit een buffer voor inkomende events, een eventbehandelaar die de events verwerkt, en een verzameling threads. Stages zijn vergelijkbaar met componentengebieden in DiPS+. SEDA laat weliswaar toe om een systeem te controleren en bij te sturen tijdens de uitvoering, maar biedt daartoe geen duidelijk gescheiden beheersvlak aan zoals in DiPS+.

## 7.2 Ondersteuning tijdens software-ontwikkeling

**Aanpasbaarheid.** Conceptueel komen SEDA stages [WCB01, Wel02] overeen met gelijktijdigheidsunits in DiPS+. Beide laten toe om afgebakende gebieden individueel aan te passen aan de belasting. Functionele logica is gescheiden van niet-functionele aspecten zoals threadbeheer en -scheduling. Het thread-controlemechanisme in SEDA is sterk gekoppeld met de stage, terwijl beheer en functionaliteit strikt gescheiden zijn in DiPS+. Enerzijds onderscheidt DiPS+ functionele en gelijktijdigheidscomponenten, anderzijds is het beheersvlak gescheiden van het datavlak.

**Testen.** Hoewel geen van de vier voorgestelde raamwerken het testen van software expliciet aanpakt, kan men zich toch een beeld vormen in hoeverre ze software-ontwikkelaars reeds de nodige ondersteuning bieden. Click, Cactus, en SEDA bieden modules aan met vaste interfaces, en houden modules bovendien onafhankelijk van elkaar, wat een groot voordeel is om een module geïsoleerd van de rest van de broncode te testen. Cactus en SEDA houden modules gescheiden door een event-gebaseerde communicatiestijl; Click van zijn kant ontkoppelt modules door expliciete in- en uitgangen; Scout is eerder grofkorrelig van structuur, en bijgevolg minder geschikt voor het testen van elementaire karakteristieken.

**Raamwerkoptimalisatie.** Het Scout besturingssysteem laat structurele optimalisaties toe, zoals het opsplitsen van connecties [MMOP95, SHHP00]. Click optimaliseert het systeem door componenten te herstructureren op basis van applicatie-, domein-, of omgevingsspecifieke kennis.

# 8 Besluit

## 8.1 De belangrijkste resultaten

Deze thesis presenteerde DiPS+, een software-architectuur en componentenraamwerk dat toelaat om op een flexibele manier protocolstapels te ontwikkelen die makkelijk kunnen aangepast worden aan applicatie- en omgevingsspecifieke vereisten. Flexibiliteit werd gerealiseerd doordat het raamwerk een strikte scheiding van belangen afdwingt, niet alleen tussen het beheer en de eigenlijke dataverwerking, maar ook tussen functionaliteit en gelijktijdigheid, en tussen functionaliteit en componentinteractie.

Het DiPS+ componentenraamwerk speelt een centrale rol in de ontwikkeling van protocolstapels. Niet alleen ondersteunt het programmeurs om protocolstapels te ontwikkelen, het legt ook de basis voor een robuust en soepel ontwikkelingsproces, en voor een evenwichtige levenscyclus. Drie grote uitdagingen bij het ontwikkelen van protocolstapels werden in deze thesis geïdentificeerd en aangepakt: zelfaanpasbaarheid, testen, en raamwerkoptimalisatie.

## 8.2 Validatie van het DiPS+ raamwerk

DiPS+ werd gevalideerd door het te gebruiken in de context van onderzoeksprojecten, en door studenten ermee te laten werken tijdens hun thesis. Ons onderzoeksteam probeert betrokken te zijn bij zowel fundamentele als eerder toepassingsgerichte projecten, gezien dit de ideale combinatie oplevert tussen innovatie en validatie in een realistische omgeving.

Ten eerste werd in het kader van een onderzoeksproject[1] (onder andere) het RADIUS protocol [RRSW97, Rig00] ontworpen en geïmplementeerd om de mogelijkheden van gelijktijdigheidscontrole te valideren. DiPS+ werd daarbij ook vergeleken met een bestaande Java-implementatie van RADIUS. Uit deze vergelijking bleek de flexibiliteit van DiPS+ en DMonA een groot voordeel om de belasting van het systeem op een applicatiespecifieke manier onder controle te houden, en daardoor betere prestaties te leveren voor bepaalde categorieën van pakketten.

Ten tweede werd DiPS+ in een ander onderzoeksproject[2] gebruikt om protocolstapels op aanvraag samen te stellen. De modulaire aanpak in DiPS+ laat toe om protocolstapels te assembleren op maat van een toestel met specifieke karakteristieken qua beschikbare systeembronnen (zoals processorkracht en geheugen).

Tenslotte werd DiPS+ gebruikt in verschillende thesissen, die specifieke protocollen implementeren, het raamwerk uitbreiden zonder de kern te veranderen, of suggesties formuleren om het raamwerk zelf aan te passen.

## 8.3 Toepasbaarheid van de DiPS+ aanpak

Hoewel DiPS+ is voorgesteld in de context van protocolstapels, zijn we ervan overtuigd dat het ook bruikbaar is in andere domeinen, zoals randapparaatsturingsprogramma's[3] en bestandensystemen. De eerste onderzoeksresultaten in die richting zijn zeer veelbelovend. Aan de basis van deze onderzoekspiste ligt het feit dat dergelijke systemen dikwijls opgebouwd zijn als een gelaagde architectuur, zoals de DiPS+ architectuur.

[1] IWT project "Service Centric Access Networks" (SCAN) #010319.

[2] IWT project "Platform for Enhanced Provisioning of Terminal-independent Applications" (PEPiTA) #990219.

[3] IWT project "Component Framework for Device Drivers" (CoFraDeD) #990280.

## 8.4 De toekomst

Deze thesis geeft aanleiding tot verscheidene onderzoekspistes, die zullen onderzocht worden in de toekomst. Enerzijds zullen die pistes de DiPS+ aanpak verder valideren. Anderzijds zullen ze DiPS+ uitbreiden om andere vereisten te ondersteunen.

Een eerste piste betreft het toevoegen, verwijderen, en vervangen van componenten tijdens de uitvoering. Waar DiPS+ momenteel enkel toelaat om het gedrag te manipuleren (via DMonA), is de ondersteuning van functionele ingrepen minstens even belangrijk.

Hiermee gekoppeld is het gedistribueerde beheer van protocolstapels. Momenteel concentreert DiPS+ zich op één enkele protocolstapel. Een functionele verandering zal echter vaak gelijktijdig toegepast moeten worden op de protocolstapels van zowel zender als ontvanger (en eventueel ook in de tussenliggende machines).

Tenslotte moet DiPS+ ondersteuning bieden om aanpassingen tijdens uitvoering op een veilige en betrouwbare manier te kunnen doorvoeren. Samen met het dynamisch vervangen van componenten en het gedistribueerde beheer, vervolledigt het beveiligen van zulke aanpassingen de flexibiliteitspuzzel.

# Referenties

[Amb02] S. W. Ambler. *Agile modeling: Effective Practices for eXtreme Programming and the Unified Process.* J. Wiley & Sons, New York, NY, USA, 2002.

[Aoy98] M. Aoyama. Agile software process and its experience. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 3–12, Kyoto, Japan, 1998. IEEE Computer Society, Washington, DC, USA.

[Asp03] AspectJ project. AspectJ – crosscutting objects for better modularity. http://www.eclipse.org/aspectj/, 2003.

[Bec99] K. Beck. *Extreme Programming explained: embrace change.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Bha96] N. T. Bhatti. *A System for Constructing Configurable High-level Protocols.* PhD thesis, Department of Computer Science, University of Arizona, Tucson, AZ, USA, December 1996.

[Bin99] R. V. Binder. *Testing object-oriented systems: models, patterns and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin-Cummings, Redwood City, CA, USA, second edition, 1994.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GP95] D. Garlan and D. E. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4):269–274, 1995.

[Hil96] M. A. Hiltunen. *Configurable Fault-Tolerant Distributed Services.* PhD thesis, Department of Computer Science, University of Arizona, Tucson, AZ, USA, July 1996.

[HS00] M. A. Hiltunen and R. D. Schlichting. The cactus approach to building configurable middleware services. In *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000), in conjunction with the 19th IEEE Symposium on Reliable Distributed Systems*, Nurnberg, Germany, October 2000.

[Joh97] R. E. Johnson. Frameworks = (components + patterns): How frameworks compare to other object-oriented reuse techniques. *Communications of the ACM*, 40(10):39–42, October 1997.

[KLL+97] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, and G. C. Murphy. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, pages 481–490, Boston, MA, USA, 1997. ACM Press, New York, NY, USA.

[KLM+97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, Jyväskylä, Finland, June 1997.

[KMC+00] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[Koh01] E. Kohler. *The Click Modular Router.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, February 2001.

[LH89] K. J. Lieberherr and I. M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, 6(5):38–48, September 1989.

[Mat99] F. Matthijs. *Component Framework Technology for Protocol Stacks*. PhD thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, December 1999.

[MDJ+02] S. Michiels, L. Desmet, N. Janssens, T. Mahieu, and P. 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, SC, USA, 2002. ACM SIGSOFT, ACM press.

[MMOP95] A. B. Montz, D. Mosberger, S. W. O'Malley, and L. L. Peterson. Scout: A communications-oriented operating system. In *Proceedings of Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 58–61, Orcas Island, WA, USA, May 1995. IEEE Computer Society Press.

[MMWV01] S. Michiels, F. Matthijs, D. Walravens, and P. Verbaeten. Position summary: DiPS: A Unifying Approach for Developing System Software. In A. D. Williams, editor, *Proceedings - The Eigth Workshop on Hot Topics in Operating Systems (HoTOS-VIII*, page 175. University of Karlsruhe, IEEE Computer Society Press, 2001.

[MP96] D. Mosberger and L. L. Peterson. Making paths explicit in the scout operating system. In *Proceedings of USENIX Second Symposium Operating Systems Design and Implementation (OSDI'96)*, pages 153–167, Seattle, WA, USA, October 1996. USENIX Association, Berkeley, CA, USA.

[MWJV] S. Michiels, D. Walravens, N. Janssens, and P. Verbaeten. DiPS: Filling the Gap between System Software and Testing.

[MWJV02] S. Michiels, D. Walravens, N. Janssens, and P. Verbaeten. DiPS-Unit: A JUnit Extension for the DiPS Framework. In *Proceedings of Third International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP2002)*, Alghero, Italy, May 2002.

[Rig00] C. Rigney. Radius accounting. http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc2866.html, 2000.

[RRSW97] C. Rigney, A. Rubens, W. Simpson, and S. Willens. Remote authentication dial in user service specification (radius). http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc2138.html, 1997.

[SG96] M. Shaw and D. Garlan. *Software Architecture - Perspectives on an emerging discipline*. Prentice-Hall, 1996.

[SHHP00] O. Spatscheck, J. S. Hansen, J. H. Hartman, and L. L. Peterson. Optimizing tcp forwarder performance. *IEEE/ACM Transactions on Networking (TON)*, 8(2):146–157, 2000.

[SN99] J-G. Schneider and O. Nierstrasz. Components, scripts and glue. In J. Hall L. Barroca and P. Hall, editors, *Software Architectures – Advances and Applications*, pages 13–25. Springer-Verlag, 1999.

[Sun] Sun Microsystems, Inc. The java hotspot performance engine architecture. Available at http://java.sun.com/products/hotspot/whitepaper.html.

[Sun03] Sun Microsystems, Inc. The java hotspot virtual machine. Available at http://java.sun.com/products/hotspot/docs/whitepaper/, 2003.

[Szy98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.

[WCB01] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP-18)*, pages 171–180, Banff, Canada, 2001. ACM Press, New York, NY, USA.

[Wel02] M. F. Welsh. *An Architecture for Highly Concurrent, Well-Conditioned Internet Services*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, August 2002.

[Wey98] E. J. Weyuker. Testing component-based software. *IEEE Software*, 15(5):54–59, 1998.

[WHS01] G. T. Wong, M. A. Hiltunen, and R. D. Schlichting. A configurable and extensible transport protocol. In *Proceedings of INFOCOM 2001*, pages 319–328, 2001.