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

Mobile Sessions in Heterogeneous Networks

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

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Tom MAHIEU

September 2006

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

Mobile Sessions in Heterogeneous Networks

Jury :
Prof. Dr. ir. L. Froyen, voorzitter
Prof. Dr. ir. P. Verbaeten, promotor
Prof. Dr. ir. W. Joosen, promotor
Prof. Dr. ir. G. Janssens
Prof. ir. C. Huygens
Prof. Dr. ir. B. Dhoedt
Prof. Dr. ir. E. Van Lil

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Tom MAHIEU

U.D.C. 681.3∗C21

September 2006

D/2006/7515/78
ISBN 90–5682–744–8

Abstract

Computers connected to a network have become an integral part of our society and the idea of being continuously connected to the Internet is gaining more acceptance. Contemporary computer networks are realized using a large number of heterogeneous access technologies, such as Ethernet, DSL, wireless Ethernet, UMTS, etc. Additionally, network devices have become small enough to be carried around and are used to communicate in public places using publicly available access networks.

Despite the availability of all these access technologies, well-equipped network devices and the use of carefully designed communication software, applications still run into problems when running in such mobile, heterogeneous network environments: network addresses and protocols change, network characteristics (bandwidth, jitter,...) fluctuate and network disconnections occur frequently.

These problems lead to four major challenges for the next generation mobility solutions: First, the mobility solution must support both address and protocol changes. Secondly, when desired, applications must be kept aware of mobility events. Thirdly, switching to another access network must happen in a secure way. Fourthly, it must be possible to deploy the solution in a heterogeneous network where access technologies and communication protocols evolve quickly.

This dissertation contributes a mobility solution architecture that addresses these challenges by introducing a session layer protocol in the protocol stack. This architecture is realized by two subsystems: the Connection Abstractions System (CAS) and the Address Management System (AMS). The CAS defines a session as a logical communication channel between two applications. Communication for a CAS session is realized using the transport protocols that are available at that time. Transport protocol connections that are aborted as a consequence of mobility are replaced by new connections. If this happens, the CAS maintains communication reliability and optionally informs the application. The CAS's session protocol can authenticate the moving applications if that is desired. Protocol changes are enabled by the AMS, which introduces the necessary concepts for developing network application development without prior knowledge of the available communication protocols. Both systems are implemented and evaluated using the DiPS+ protocol stack framework.

Voorwoord – Preface

Computernetwerken zijn dankzij het Internet alsmaar belangrijker geworden en hebben het laatste decennium een explosieve groei gekend. De manier waarop computernetwerken gerealiseerd en gebruikt worden verandert voortdurend: de toegangsmethoden veranderen (draadloze communicatie, Internet toegang via de TV-kabel,...) en de applicaties veranderen (van e-mail en bestandentransfer naar online gaming, video streaming, instant messaging,...). Deze tekst beschrijft een mobiliteitsoplossing die toelaat dat computers, die alsmaar kleiner en krachtiger worden en steeds vaker draagbaar zijn, flexibeler kunnen communiceren in hedendaagse mobiele computernetwerken. Deze mobiliteitsoplossing is tot stand gekomen tijdens de jaren die ik in de DistriNet onderzoeksgroep aan het departement computerwetenschappen van de K.U.Leuven gewerkt heb.

Toen ik bij DistriNet begon te werken had ik helemaal geen idee dat dit het resultaat zou zijn van mijn doctoraatsonderzoek. Mijn onderzoek had initieel immers niks met computernetwerken te maken maar met raamwerktechnologie voor natuurlijke taalverwerkingssystemen. De weg om van mijn initieel doctoraatsonderwerp te komen tot de tekst die je nu aan het lezen bent was een lange weg, voorzien van de nodige frustratiemomenten. Ik voel me dan ook verplicht iedereen te bedanken die zowel op professioneel als persoonlijk vlak bijgedragen heeft tot dit werk.

Ik wil eerst mijn promotoren, professor Pierre Verbaeten en professor Wouter Joosen, bedanken. Hun kritische geest, veeleisende ingesteldheid, geduld en hun vermogen om mensen te motiveren waren voor mij van onschatbare waarde. Daarnaast wil ik de leden van mijn begeleidingscommissie, professoren Gerda Janssens en Christophe Huygens, bedanken. Hun andere invalshoek op dit onderzoeksdomein vormden een belangrijke bijdrage tijdens het tot stand komen van deze tekst. Ook de voorzitter en de andere leden van de jury, professoren Bart Dhoedt en Emmanuel Van Lil wil ik bedanken voor hun interesse in dit werk.

Ik wil ook de leden van de DistriNet onderzoeksgroep bedanken. In het bijzonder bedank ik de leden van de networking taskforce voor een aangename werkomgeving, Sam Michiels, Nico Janssens, Lieven Desmet, Thomas Delaet en Bart Elen. Ook voormalige networking taskforce en DistriNet leden, Frank Matthijs, Dirk

Walravens en Bart Vanhaute ben ik mijn dank verschuldigd: Frank, omdat hij de eerste aanzet gaf tot het Address Management System, Dirk, vooral omwille van zijn onvoorwaardelijke inzet om TCP in DiPS+ geïmplementeerd te krijgen en Bart die me mijn eerste paper wellicht 3 keer heeft doen herschrijven (maar hij was dan ook aanvaard).

Daarnaast heb ik aan het departement computerwetenschappen heel wat mensen leren kennen. Bij Jean Huens en/of Bart Swennen kon ik altijd terecht toen we het eerste netwerklabo uit de grond stampten of als er zich obscure netwerkproblemen voordeden. Ze hebben me doen inzien dat systeembeheer geen triviale taak is, die al te vaak onderschat wordt. Ook heb ik een groot aantal bureaugenoten gehad omdat DistriNet er toch wel in slaagde om ongeveer elke 2 jaar een bureauverhuis te organiseren. Bedankt voor de toffe werksfeer de afgelopen jaren, David Goelen, Bart De Win, Jan Van den Bergh, Romain Slootmaekers (sindsdien heb ik nooit meer bisonwodka gedronken), Stefan Raeymakers, Yves Younan, Liesje Demuynck, Steven Gevers, Bart Jacobs, en degenen die ik verder nog vergeten ben. Verder wil ik nog Remko Tronçon, Andrew Wils bedanken voor de aangename alma momenten. Een speciale 'thanks' gaan naar Eddy Truyen en Pieter Bekaert. We waren, denk ik, experten in het klagen over hoe lastig het leven als doctoraatstudent wel was, vooral aan de toog.

Naast werk was er ook ontspanning. Bert, Geert en Natasja, Lies en Lorenzo, Rob en Elke waren altijd wel voor een gezelschapspelletje te vinden. De laatste vier jaar was ik ook vaak op de dansvloer terug te vinden. Ik wen**s** dan o**o**k **a**lle dansers en de trainers van de Ha**p**py Faces en Dance Connection formatieteams te bedanken voor de ont**s**pann**e**nde momenten op de t**r**ainingen, en de soms toch ook spannende en emot**i**onele momenten op de dansw**e**dstrijden. Ik wil vooral Hilde Delrue bedanken om met zoveel toewijding deze tekst na te lezen terwijl ze er wellicht niks van begreep. Het doet je ook inzien dat termen zoals piggybacking (zwijntjesrug?) en garbage collection (de vuilkar?) voor een buitenstaander best wel grappig klinken in een technische tekst zoals deze.

Ik wil mijn ouders en mijn broer bedanken om me te blijven steunen doorheen de jaren. Het heeft lang geduurd, maar 't is er toch nog van gekomen.

Tenslotte, en vooral, wil ik Mie bedanken. Ze was er bij van in het begin en is me doorheen de jaren blijven steunen. Ze deed me de dingen relativeren, vooral tijdens momenten van frustratie en demotivatie. Ook Anika, mijn dochtertje, bedank ik. Ze is pas 5 maand op deze wereld, maar ze heeft me al zoveel mooie momenten bezorgd. Aan haar draag ik ook dit werk op. Ze gaf me dat extra duwtje in de rug... en 't heeft geholpen, zoals je kunt zien.

Tom Mahieu, september 2006.

Aan Anika.

Contents

1 Introduction **1**
- 1.1 Dynamic networks . . . 1
- 1.2 Dynamic network paradigms . . . 3
  - 1.2.1 Open wired networks . . . 3
  - 1.2.2 Wireless networks . . . 3
  - 1.2.3 Mobile ad-hoc networks (MANETs) . . . 4
  - 1.2.4 Overlay networks . . . 5
  - 1.2.5 Service centric access networks (SCANs) . . . 5
- 1.3 Software support in dynamic networks . . . 6
- 1.4 Contribution . . . 7
- 1.5 Organization of the text . . . 8

2 Mobility solution challenges and taxonomy **11**
- 2.1 Challenges . . . 11
  - 2.1.1 Address and protocol changes . . . 11
    - 2.1.1.1 Handling address changes . . . 11
    - 2.1.1.2 Handling protocol changes . . . 12
    - 2.1.1.3 Coping with protocol diversity/proliferation . . . 13
  - 2.1.2 Application awareness of mobile behavior . . . 14
    - 2.1.2.1 Keeping the application involved . . . 15
    - 2.1.2.2 Handling disconnection as a way of life . . . 15
  - 2.1.3 Securing mobile endpoint behavior . . . 16
  - 2.1.4 Openness of heterogeneous networks . . . 17
- 2.2 A Taxonomy of Mobility Solutions . . . 18
  - 2.2.1 Network Layer Mobility Solutions (NLMSs) . . . 19
  - 2.2.2 Transport Layer Mobility Solutions (TLMSs) . . . 21
  - 2.2.3 Socket Layer Mobility Solutions (SoLMSs) . . . 24
  - 2.2.4 Proxy Mobility Solutions (PMSs) . . . 26
  - 2.2.5 Application Specific Mobility Solutions (ASMSs) . . . 28
  - 2.2.6 Session Layer Mobility Solutions (SeLMSs) . . . 29
  - 2.2.7 Summary . . . 31

3 A session based networking environment **33**
3.1 Motivation for a session layer approach . . . 33
3.2 Best practices and design strategies . . . 34
3.2.1 Limit lower layer dependencies from higher layers . . . 35
3.2.2 Handle unexpected disconnections gracefully . . . 35
3.2.3 Do not restrict the choice of naming techniques for mobile nodes . . . 35
3.2.4 Provide support at the endpoints . . . 36
3.2.5 Optimize for the static case . . . 36
3.3 Session definition . . . 37
3.4 Session Layer Mobility Solution architecture . . . 40
3.5 Session management tasks in dynamic networks . . . 42
3.5.1 Session support detection . . . 43
3.5.2 Transport and network protocol independent session identification . . . 43
3.5.3 Protocol and address hiding . . . 44
3.5.4 Session state management . . . 44
3.5.5 Session negotiation protocol . . . 46
3.5.6 Transport protocol management . . . 47
3.5.7 Maintaining communication channel semantics . . . 47
3.5.8 Offering application feedback . . . 48
3.6 Evaluation . . . 48
3.7 The Connection Abstraction System and Address Management System . . . 49

4 The Connection Abstraction System **51**
4.1 CAS session definition . . . 51
4.2 Designing a session layer in the protocol stack . . . 52
4.2.1 Interaction of CAS with the application layer . . . 54
4.2.2 The relation between the CAS and the transport layer . . . 54
4.3 The CAS header format . . . 55
4.4 Description of the session protocol . . . 57
4.4.1 Session establishment . . . 58
4.4.1.1 Transition diagram for session establishment . . . 58
4.4.1.2 Protocol message exchange for session establishment 61
4.4.2 Session suspension . . . 62
4.4.2.1 Transition diagram for session suspension . . . 62
4.4.2.2 Protocol message exchange for session suspension 65
4.4.2.3 Using reliable transport protocols . . . 66
4.4.3 Session resumption . . . 68
4.4.3.1 Transition diagram for session resumption . . . 68
4.4.3.2 Protocol message exchange for session resumption 71
4.4.3.3 Using reliable transport protocols . . . 72

4.4.4 Session termination . . . 73
4.4.4.1 Transition diagram for session termination . . . 74
4.4.4.2 Protocol message exchange for session termination 76
4.5 Transport connection management . . . 76
4.6 Network status feedback for the application . . . 78
4.7 Security measures . . . 80
4.7.1 Protocol security . . . 80
4.7.1.1 Attack-equivalence . . . 80
4.7.1.2 CAS vulnerabilities . . . 81
4.7.1.3 Authentication of CAS protocol requests . . . 83
4.7.1.4 Preventing denial-of-service . . . 84
4.7.2 Network security . . . 86
4.7.2.1 NATs and firewalls . . . 87
4.7.2.2 Using IPsec . . . 90

5 The Address Management System 93
5.1 Introduction . . . 93
5.2 High level overview of the AMS . . . 94
5.3 Technical realization of the AMS . . . 95
5.3.1 Generic addresses . . . 95
5.3.2 Reducing generic addresses . . . 97
5.4 The CAS and AMS synergy . . . 99

6 Realization and Evaluation 101
6.1 Architecture compliance . . . 101
6.2 DiPS+ overview . . . 105
6.2.1 DIPS+ packets . . . 106
6.2.2 DiPS+ components . . . 106
6.2.3 DiPS+ connectors . . . 107
6.2.4 DiPS+ layers . . . 108
6.2.5 DiPS+ layer resources . . . 109
6.2.6 DiPS+ framework communication mechanisms . . . 109
6.2.7 DiPS+ in dynamic networks . . . 110
6.3 CAS design in DiPS+ . . . 110
6.3.1 Maintaining session data . . . 111
6.3.2 The CAS layer design . . . 111
6.3.3 CAS state machine design . . . 114
6.3.4 Transport layer interaction . . . 116
6.3.5 Application interaction . . . 117
6.4 AMS design in DiPS+ . . . 117
6.4.1 Reduction: stack and layer address managers . . . 117
6.4.2 Plugging AMS in the DiPS+ stack . . . 119
6.5 Implementation and protocol evaluation . . . 120

6.5.1 CAS overhead . . . 120
6.5.1.1 Processing overhead . . . 120
6.5.1.2 Protocol overhead . . . 125
6.5.1.3 Header overhead . . . 126
6.5.1.4 Memory usage . . . 127
6.5.1.5 Code size . . . 128
6.5.2 Detecting disconnection . . . 129
6.5.2.1 Disconnection detection with connection-oriented protocols . . . 130
6.5.2.2 Disconnection detection with connectionless protocols . . . 130
6.5.3 Network traffic during immediate handover . . . 131
6.5.3.1 UDP . . . 132
6.5.3.2 TCP . . . 132
6.6 Validation in industry projects . . . 132
6.6.1 Project PEPiTA . . . 132
6.6.1.1 AMS role in PEPiTA . . . 134
6.6.2 Project SCAN . . . 134
6.6.2.1 The role of the AMS and CAS in SCAN . . . 135

7 Related work 137
7.1 Existing solutions to mobile computing . . . 137
7.1.1 Network Layer Mobility Solutions . . . 137
7.1.1.1 Unicast based solutions . . . 138
7.1.1.2 Multicast based solutions . . . 141
7.1.1.3 General NLMS evaluation . . . 141
7.1.2 Transport Layer Mobility Solutions . . . 142
7.1.2.1 TCP-R . . . 142
7.1.2.2 TCP Migrate . . . 144
7.1.2.3 Endpoint migration for improved service availability 146
7.1.2.4 General TLMS evaluation . . . 148
7.1.3 Socket Layer Mobility Solutions . . . 149
7.1.3.1 Rocks and Racks . . . 149
7.1.3.2 MobileSocket . . . 151
7.1.3.3 Mobile Socket Layer (MSL) . . . 153
7.1.3.4 Persistent Connection . . . 154
7.1.3.5 General SoLMS evaluation . . . 156
7.1.4 Proxy Mobility Solutions (PMSs) . . . 157
7.1.4.1 MSOCKS . . . 158
7.1.4.2 Indirect TCP . . . 159
7.1.4.3 PMS evaluation . . . 161
7.2 Session layer solutions . . . 162
7.2.1 OSI model . . . 162

7.2.1.1 Discussion . . . 163
7.2.2 Migrate . . . 163
7.2.2.1 Discussion . . . 165
7.2.2.2 Session management tasks . . . 166
7.2.3 TESLA . . . 168
7.2.3.1 Discussion . . . 169
7.2.3.2 Session management tasks . . . 169
7.2.4 SLM . . . 170
7.2.4.1 Discussion . . . 171
7.2.4.2 Session Management Tasks . . . 172
7.3 Mobility in the GSM world . . . 173
7.3.1 Overview of the GSM system . . . 173
7.3.2 Discussion . . . 174

8 Conclusion **177**
8.1 Summary and contributions . . . 177
8.2 Future work . . . 181
8.3 Some final considerations . . . 182

Glossary **185**

Bibliography **191**

List of Figures

2.1 Network layer mobility solutions 20
2.2 Transport layer mobility solutions 22
2.3 Application layer mobility solutions: the socket layer approach . . 24
2.4 Application layer mobility solutions: the proxy approach 26
2.5 Application specific mobility solutions 28
2.6 Session layer mobility solutions 30
2.7 Solution taxonomy summary 32

3.1 A session is a logical communication channel between two endpoints owned by two applications. 37
3.2 A session is associated with a transport protocol connection that is replaced by a new one each time the session's endpoint moves. . . 39
3.3 A Session Layer Mobility Solution resides in the session layer in the protocol stack. Session functionality is offered to the application by means of a session socket. The Session Layer Mobility Solution uses the services of the lower transport, network and data link layer protocols to establish physical communication channels. 41
3.4 Session state management 45
3.5 the CAS and AMS with respect to the system's protocol stack . . 50

4.1 The CAS is realized in the session layer of the OSI reference model [Zim80] 52
4.2 The CAS header format 56
4.3 Simplified transition diagram of the CAS protocol. The square boxes represent the four CAS protocol actions. The grey ovals represent the states that the detailed state transition diagrams of the connected protocol actions have in common. 57
4.4 Session establishment transition diagram 59
4.5 Session establishment protocol 61
4.6 Session suspension transition diagram 63
4.7 Session suspension protocol 66

4.8 Session resumption transition diagram 70
4.9 Session resumption protocol 72
4.10 Session closing transition diagram 74
4.11 Session termination protocol 77
4.12 Session establishment with Diffie-Hellman key agreement. 84
4.13 Verifying the validity of a session resumption request by sending a challenge to the client. The client must reply with a valid response. 85

5.1 An example of a generic address 95
5.2 The generic address from Figure 5.1 after a) protocol reduction and b) address reduction. 98

6.1 DiPS+ component anatomy 106
6.2 DiPS+ components are connected to each other in a pipeline . . . 107
6.3 DiPS+ Layers. The left hand side of the figure shows that a DiPS+ layer internally possesses two component pipelines: a downgoing path, which goes from the upper entry point to the lower exit point and an upgoing path, which goes from the lower entry point to the upper exit point. This layer also contains one layer resource. The right hand side shows that DiPS+ layers can be stacked on top of each other and exist next to each other on the same OSI level. Layers between levels are interconnected by means of layer glue. . 108
6.4 DiPS+ design of the CAS 112
6.5 UML Diagram of the CAS state machine in DiPS+ 114
6.6 DiPS+ design of the CAS state machine 115
6.7 Using the socket layer in the DiPS+ framework 116
6.8 Address management in the DiPS+ framework 118
6.9 Timing results for 10 testruns for UDP (left hand side) and TCP (right hand side). The stars on the blue lines show the timing results for a DiPS+ stack that contains the CAS layer, the circles on the green lines show the results for a stack that does not contain the CAS layer. 121
6.10 Timing results for 10 testruns for UDP (top figure) and TCP (bottom figure). The timing values depicted are the same as in Figure 6.9 without the timing values from the warmup phase. The red line depicts the mean time to run a test with CAS/AMS, the purple line depicts the mean time to run a test without CAS/AMS. 123
6.11 Boxplot of the timing results for every protocol stack type. The figure indicates the medians for every stack tested, and also indicates the dispersion of the time values measured by means of upper and lower quartiles and whiskers that extend to maximally 1.5 times the interquartile range. The plus signs indicate values that fall outside the range of the whiskers. 124

6.12 Proportional CAS header size with respect to the amount of data in the CAS packer. 126
6.13 The PEPiTA software platform 133

7.1 General architecture of network layer mobility solutions 139
7.2 The general architecture of network layer mobility solutions applied to Mobile IP 140

Listings

4.1 A Java Session Socket API. 53
4.2 Example API for connection state exporting and importing. 78
4.3 CAS feedback API. 79
4.4 CAS socket methods to avoid denial of service attacks 86

Chapter 1

Introduction

Computer network technology has known enormous progress since the Internet started being used commercially. Existing network hardware is constantly being improved and new communication technology continues to be developed. Typical LAN communication hardware, like for example Ethernet, has become a lot faster. Network backbones consist of fast optical networks. Home networks are no longer a single PC with a slow modem connected to a telephone line, but consist of a number of interconnected devices that can communicate on the internet 24/7 using a DSL or cable modem. Wireless communication technologies have increased computing comfort at home and have introduced public hot spots, facilitating internet access outside the domestic environment.

Together with the trend that computing hardware becomes smaller and portable, this network technology progress has led to dynamic networks. First, the characteristics of a dynamic network are in explained in Section 1.1. We then shortly describe a number of popular network computing paradigms that possess dynamic network properties in Section 1.2. Section 1.3 shortly outlines existing software support to cope with dynamic networks. Section 1.4 positions this work in the broad domain of dynamic network software. Section 1.5 explains how this text is organized.

1.1 Dynamic networks

A dynamic network is a *heterogeneous network* in which *endpoint mobility* is supported. This section describes in greater detail what a heterogeneous network is and what endpoint mobility is.

A *heterogeneous network* is a computer network that consists of a large number of access networks, equipped with a multitude of communication technologies, ranging from traditional wired network technologies such as telephone modems

and Ethernet to wireless network technologies such as Wifi, GSM, GPRS and UMTS. These network technologies have different technical properties such as different bandwidth, throughput, latency and network coverage. Access networks are often managed by different authorities. In this work, we will refer to a set of access networks that are managed by the same authority as a management domain. Management domains determine additional technical and administrative characteristics of their access networks. Management domain specific technical characteristics are for example the used communication protocols. Example administrative characteristics are among others communication cost, bandwidth limits and security policies.

A heterogeneous network is populated by a heterogeneous set of computing devices, like laptops, palmtops, cellphones, etc. To communicate, a device must be connected to an access network using a *network attachment point*. To be able to connect to a network attachment point, the device must be equipped with a compatible network interface. A connection with a network attachment point can be realized using a physical connection, i.e. a wire, but can also be a wireless connection.

The devices in a dynamic network have become sufficiently small and portable to be carried around easily. Laptops can easily be unplugged from the network and could be taken from the work floor to the user's home. To remain connected to the network it will be necessary to change to another network attachment point. Contemporary computing devices are typically equipped with multiple network interfaces of a different type. A device can switch to an access point realized with a different network technology, potentially located in another management domain. For example, a laptop can be moved from its home, DSL network to a wireless GPRS connection. Network connectivity hence does not depend on the availability of one particular type of access network.

The second characteristic of dynamic networks is the support for *endpoint mobility*. In this work, we define an endpoint as a software concept that denotes the termination of a transport layer communication channel. Endpoints are used by applications and are traditionally offered by the operating system as sockets. Endpoint mobility then denotes the ability of an endpoint to seamlessly continue communication in the event of network attachment point changes. From the perspective of the endpoint, network attachment point changes can be the consequence of moving the hosting device to another location or of moving an application to another host. In this work, endpoint mobility will always refer to device movement, unless stated otherwise.

Endpoint mobility can be particularly hard to achieve if the endpoint is moved to another management domain. Changes in technical and administrative characteristics may interfere with the business logic of the application that is using the endpoint. For example, technically, it may not be possible to send a video stream after switching to a network attachment point with lower bandwidth char-

acteristics. Administratively, a bank account management program will not be (or should not be) able to operate on a network other than the bank's local network because of the absence of security protocols and policies.

Dynamic networks have led to new ways of network computing. Such dynamic network paradigms allow applications to benefit from a heterogeneous, mobile network environment. The next section describes a number of different dynamic network paradigms.

1.2 Dynamic network paradigms

This section describes five networking paradigms for dynamic networks. These paradigms are named open wired networks, wireless networks, mobile ad-hoc networks, overlay networks and service centric access networks.

1.2.1 Open wired networks

The open wired network paradigm is one of the earliest dynamic network ideas and became popular when the mobile computing devices appeared in the predominantly wired networks. This paradigm deals with the *consequences* of mobile behavior. Open wired networks assure network connectivity and device reachability *after* the device moved to another location. The corresponding devices of a mobile device must be able to send data to the mobile device regardless of its location. To realize this, all traffic destined for a mobile device should be routed to its new location after the device is unplugged from the network, moved to another location and reconnected to the network.

Device reachability is the main concern of a mobile network. The capability to deal with network heterogeneity is subordinate and typically depends on application requirements. A decade ago the main network applications were email, web browsing and file transfer. The impact of the technical characteristics of the used communication technology on those application is a lot smaller than on contemporary applications such as multimedia applications that require real time audio and video streams.

In an open wired network, a mobile device is not expected to communicate during movement. Because computer network were mainly wired, it was simply not possible to keep the device connected to the network during movement. This changed since the ubiquitousness of wireless networks, which is the next paradigm that is discussed.

1.2.2 Wireless networks

The wireless network paradigm found its inception in wireless communication technology like cellular phone networks, WaveLAN and WiFi; the wire disappeared.

Contrary to open wired networks, a wireless network also allows communication during device mobility. Consequently, applications are developed with that capability in mind. For example, a palmtop can be used to communicate with a tourist information office while sightseeing a city.

Wireless networks require a significant amount of infrastructure. A network attachment point in a wireless network is a wireless base station with a limited geographical coverage. To cover a large area, a number of interconnected, adjacent base stations are needed, and it must be possible for the device to switch between base stations. The event of switching to another base station is typically called a handover. Handovers between base stations in the same management domain are typically supported by the technology and are transparent for the user. Wireless networks are hence homogeneous. Nevertheless, the sensitivity of wireless transmission technology to interference can result in loss of bandwidth.

1.2.3 Mobile ad-hoc networks (MANETs)

Mobile ad-hoc networks are also wireless networks, but do not depend on the availability of network infrastructure. Every device in an ad-hoc network communicates directly with other devices in the network using wireless communication technologies. These networks are called ad-hoc because the network topology is formed on the spot with the devices that are in range at the moment. Devices can enter and leave the network freely. Every wireless device functions as a router in the network; every device is expected to forward packets on behalf of the other devices in the network. A MANET can cover a large area if the devices are evenly distributed over that area. Also, MANETs are not often pure ad-hoc networks, but are hybrid wireless/ad-hoc networks; they use network infrastructure when it is available.

In a MANET, network characteristics such as bandwidth, propagation delay and other communication characteristics constantly change for three reasons. First, because MANET devices are mobile routers, the network topology often changes and the connectivity of a device constantly changes. If a mobile device moves out of range of the last reachable device in the ad-hoc network, the device becomes disconnected. Secondly, MANETs can use heterogeneous communication technology. A device can for example establish a Bluetooth connection with one device and use its wireless ethernet interface in ad-hoc mode to communicate with another device. Thirdly, computing device heterogeneity in the ad-hoc network affects network performance. The computing capacity of a mobile device in a MANET largely determines its packet routing speed. For instance, a laptop will be a faster MANET router than a cellphone.

Summarized, compared to mobile networks and wireless networks, MANETs are very dynamic networks because the network topology changes frequently. MANETs consist mainly of mobile devices that use wireless communication technologies. There are no restrictions on the used communication technologies, and

the devices can range from simple cellphones to powerful portable computers.

1.2.4 Overlay networks

Overlay networks [KB96] are wireless networks that ensure continuous network connectivity for a mobile device by automatically switching to another wireless communication technology if connection is lost. For example, when moving out of the range of a WiFi hotspot (a publicly available, wireless Ethernet access point), the device could switch to a GPRS network to remain connected. Switching to a different network technology in an overlay network is called a vertical handover, as opposed to a horizontal handover, which is switching to a different network attachment point of the same technology.

The geographical areas covered by the different wireless communication technologies in an overlay network are different in size and lay over each other[1]. For example, the area covered by a WiFi hotspot is typically completely overlapped by a GPRS network. The functioning of an overlay network hence relies on the diversity of the available wireless communication technologies, mainly with respect to area coverage. By consequence, other technical characteristics, such as bandwidth, and non-technical characteristics, such as communication cost, are also very different. For instance, GPRS, on the one hand, covers a large area, is low bandwidth and expensive to use. Wireless ethernet, on the other hand, is cheap to use, offers high bandwidths but is very short range.

The difference between MANETs and overlay networks is twofold. First, overlay networks mainly uses infrastructure dependent network technologies. MANETs do not depend on network infrastructure, devices that participate in a MANET provide the network infrastructure themselves. Secondly, devices in an overlay networks communicate using one wireless communication technology at a time and switch to another technology when the old one is no longer feasible. MANET devices use all communication technologies simultaneously.

1.2.5 Service centric access networks (SCANs)

In a service centric access network, mobility is not the consequence of device mobility, but of application preference. A device switches to another access network when an application running on that device can perform better on that access network. Different access network characteristics are better suited for different services. The type of service determines the type of access network that will be used. For example, when using a video on demand service, an access network will be selected that supports a QoS protocol that allows an application to reserve the

[1] The literature also refers to an overlay network as a network routing technology that is an application layer overlay of the network layer substrate[ABKM01]. However, in this work, an overlay network always refers to a network that is built using wireless network technologies that geographically overlap.

necessary bandwidth to watch the movie. If one wants to download a large file on a peer to peer network, like BitTorrent [Coh03] for example, an access network will be selected that does not limit the upstream bandwidth of a user.

The SCAN paradigm is interesting when multiple access networks are available simultaneously. Also in the case of home networks, non-mobile desktop computers can benefit from a SCAN because it is possible that multiple internet access providers are available on the existing wire infrastructure. In that case, service centric access networks introduce logical endpoint mobility: although a device remains physically connected to the same network attachment point(s), it is logically moved to another access network.

1.3 Software support in dynamic networks

A lot of system software has been and is still being created to support application development in dynamic networks. Such software is mainly developed in the system's protocol stack or on the middleware level. This section shortly discusses software support for dynamic network behavior and argues that this software support is still very limited.

Protocols and protocol stacks have followed the trend to support more flexibility in the network. Software that allows mobile terminal behavior in the network has been developed on all levels in the protocol stack. On the data link layer, horizontal handover techniques have been developed. The most prominent solution in the network layer is Mobile IP [Per96, JPA04]. Software solutions in the transport layer exist but are less popular. They are often adaptations or extensions of existing protocols, such as TCP Migrate [SB00] or SCTP dynamic address reconfiguration [SRX+05]. Additionally, protocol stack *implementations* are no longer a monolithic chunk of system software. Protocol stack composition frameworks, such as DiPS+[Mat99, Mic03], and Click [KMC+00] have been developed that allow protocol stack composition based on application needs and on the current network situation [ŞMBV03, ŞVB02]. Other solutions offer support to alter [JMMV02, MJD+05] or program [CDK+99, CBZS98] the protocol stack at runtime.

On the middleware level, there exist numerous solutions that help the application to run in dynamic network environments, such as network configuration mechanisms (DHCP), service discovery protocols (UPnP, Jini,...), dynamic naming systems (Dynamic DNS [TRB97]), mobile network service models (J2EE, Embouchure [Myh03], Coda filesystem [MES95], Rover mobility toolkit [JTK97]). Next to application support there is also middleware that realizes different communication models for network paradigms that diverge from the traditional fixed infrastructure based networks. For example, ad-hoc networks or delay tolerant systems [Fal03, BHT+03] may consist of devices that are only connected to the network intermittently or have limited resources (power) to communicate or com-

pute. Examples of such middleware are communication coordination models (e.g. tuple spaces [PMR99, Gel85]).

Despite these developments, software support for dynamic networks is still very limited. It is especially remarkable that a lot of dynamic network applications and middleware still depend on protocol stack solutions that were developed to cope with the open wired networks paradigm. For example, Mobile IP was initially developed to realize open wired networks; it copes with IP address changes that are the consequence of network attachment point changes. Mobile IP is still used as supporting technology in wireless network solutions and overlay network solutions, even though Mobile IP offers no support when a device becomes disconnected.

Next to that, also middleware solutions are hardly developed for heterogeneous network environments. Middleware solutions are typically developed for a particular development and/or execution environment. They are also often targeted at a specific application domain and are not designed to interoperate well with other solutions in that domain. For example, Jini service discovery is only available to Java applications and does not interoperate with other service discovery implementations. Furthermore, there exist many ways to describe a service, such as normal naming schemes like DNS, intentional naming systems [AWSBL99], etc. Service discovery systems that use different service description mechanisms are not interoperable.

1.4 Contribution

The goal of this work is the realization of a mobility solution for dynamic networks. We use the term *mobility solution* to refer to a software solution that realizes endpoint mobility, i.e. the ability of an endpoint to seamlessly continue communication in the event of mobile terminal behavior. The mobility solution presented in this work supports applications running on a mobile device operating in a heterogeneous network environment. The developed solution cannot be applied for every possible dynamic network paradigm. Given the diversity of the network paradigms described in section 1.2, that would not be realistic. The mobility solution described in this work targets more traditional dynamic networking paradigms that rely on a communication infrastructure, like wireless networks and overlay networks, containing devices that have enough processing power, memory and storage capacity to run a modern protocol stack. The applications that run in this network are typical client-server applications or peer-to-peer network applications. More specifically, the mobility solution is very relevant to applications that use long living transport protocol connections, like the ones used for audio/video streaming applications, file transfer applications, instant messengers and login sessions. Transport connections, like those used for fetching a web page, doing a DNS request, etc. have typically such a short life span that they don't suffer from mobility events, and if they do, the user typically retries. Infrastructure-less dynamic

network paradigms that consist of low capacity devices which only communicate intermittently are also not the target of this mobility solution.

The contribution of this work is fourfold. The first contribution is the identification of the challenges that must be addressed by mobility solutions deployed in a dynamic network. We have identified four main challenges: mobility solutions must be able to handle the consequence of both address and protocol changes, the application should be made aware of changes in the network if desired, endpoint identity must be ensured when they become mobile and communication technology must possess a degree of openness to be able to deal with network heterogeneity.

The second contribution of this dissertation is a taxonomy of mobility solutions. We have limited ourselves to categorize solutions that are realized in the protocol stack or cooperate closely with the protocol stack. We prefer protocol stack solutions because they are part of the operating system and are therefore applicable to all applications. It is not necessary to force the application to use a specific middleware solution. Mobility solutions in the protocol stack are often also better equipped to inspect and monitor the network situation and can therefore more adequately deal with consequences of mobility.

Thirdly, we have developed a Session Layer Mobility Solution architecture. The mobility solutions taxonomy indicates that endpoint mobility is realized best in the protocol stack's session layer. The architecture defines the concept of a mobile session, determines how the Session Layer Mobility Solution interacts with the other layers in the protocol stack and determines the tasks it must accomplish to realize endpoint mobility.

Fourthly, we have realized a Session Layer Mobility Solution that adheres to this architecture. The solution consists of two parts: the Connection Abstraction System (CAS) and the Address Management System (AMS). The CAS is the actual Session Layer Mobility Solution. The AMS is an add-on to the protocol stack that enables application development without prior knowledge of the available protocols in the operating system's protocol stack. We implemented the CAS and AMS in the DiPS+ protocol stack framework and did some elementary performance evaluations.

1.5 Organization of the text

The remainder of this text is organized as follows. In Chapter 2, we identify the challenges that are raised by dynamic networks. We also describe how these challenges are met by different categories of protocol stack mobility solutions.

Chapter 3 describes our concept of a mobile session, the developed session layer architecture and describes how this architecture can realize these challenges by means of eight session management tasks.

Chapter 4 describes the developed Connection Abstraction System which we developed as an instance of the proposed session layer architecture. The Con-

nection Abstraction System is designed in the protocol stack as a session layer protocol. The chapter describes the design of the protocol in the protocol stack and discusses some security concerns for the protocol.

The Address Management System is explained in Chapter 5. The chapter defines the concept of Generic Addresses which allows application development independent from the used communication protocols in the network. When the Connection Abstraction System and Address Management System are used together, they address the formulated challenges and realize endpoint mobility in heterogeneous networks.

Chapter 6 describes an implementation of both systems in the DiPS+ protocol stack framework [Mic03], discusses the operational overhead of the implementation and describes how the systems have been applied successfully in industry projects.

Chapter 7 describes related work. We evaluate a number of mobility solutions from the different mobility solution categories identified in Chapter 2. For each solution we verify how they address the dynamic network challenges from Chapter 2. For Session Layer Mobility Solutions, we also verify if they realize the tasks of the proposed session layer architecture.

Chapter 8 concludes this dissertation and describes future work.

Chapter 2

Mobility solution challenges and taxonomy

In his chapter we specify the challenges for mobility solutions deployed in a dynamic networks. We identify four main challenges, which are described in Section 2.1. Existing mobility solutions already deal with some of these challenges, but not all of them. We categorize mobility solutions according to their location in the protocol stack, and discuss how each solution category can address the challenges.

2.1 Challenges

We have identified four challenges in a dynamic network. First, mobility solutions must support for address and protocol changes. Secondly, the application should be aware of mobile behavior if that is desired. Thirdly, mobile behavior should be secure. Fourthly, heterogeneous networks are open networks in which it must be easy to deploy new protocols and mobility solutions. Mobility solutions, protocols and also protocol stacks must be designed in a way that promotes open networks. The following sections discuss these challenges in greater detail.

2.1.1 Address and protocol changes

2.1.1.1 Handling address changes

When a terminal moves to another network attachment point, the network layer address of the host may change. It is possible that the device's new attachment point is configured in another network subnet. If the host is moved to an attachment point of another management domain, the device will most certainly receive another network layer identifier. In the case of the Internet, different organizations

are assigned different, non-overlapping sets of addresses by IANA[Int06]. At the end of the working day, when a user goes home, his laptop is moved from the corporate network to his home network. Both the corporate network and user's ISP will configure the laptop with a different IP address.

Network address changes are not transparent for the higher transport and application layer. This can be explained by the double function of network layer addresses[BPT96]. On the one hand, they are used as a routing directive. The network layer uses the address to route packets to the correct destination. On the other hand, the application and transport layer use the network layer address as an endpoint identifier for transport protocol sockets and transport layer connections. Moving to another location requires changing the network layer address so traffic can be routed to the device's new location. Additionally, the transport and network layer must use the new address to correctly identify the existing open network connections and sockets. However most transport layer protocols and applications do not anticipate such address changes and hence cannot cope with them. TCP connections, which are identified using the IP addresses of both endpoints, will break because TCP does not support IP address changes on an open connection. If the IP address of one endpoint is changed, the connection is reset because the peer is not able to identify the old connection with the newly assigned IP address. Applications that depend on a device's network layer address to realize a service are not resistant to address changes either, unless they provide additional means to identify a session. For example, a web application that uses a device's network layer address to track the actions performed during a session will fail after the address changes. The web application will have to store a network layer address-independent identifier in a browser cookie [Net99] instead.

Protocol stacks that are used in dynamic networks should offer support for address changes that result from mobile behavior[1]. They should tackle the technical consequences of address changes and allow the application to focus on the influence of such mobile behavior on the business logic. Mobile IP [Per96, JPA04] is the most popular technology used to handle address changes. They solve the address changing problem by giving a host two IP addresses. One is used as a routing directive and the other is used for identification. Only the address that functions as routing directive changes. The application and transport layer always use the IP address that is used for identification, because that address never changes.

2.1.1.2 Handling protocol changes

Not only addresses but also *address schemes* can change when a terminal moves between networks. Different protocols typically use different address schemes. In a dynamic network, address scheme changes are a common event. Different

[1]Note that there is also mobile application behavior: an application may be moved to another terminal. Next to address changes, this also requires the migration of application state between the terminals. This is not covered in this work.

organizations may use other protocol stack instances to realize their networks. For example, the company's network can use IPv6 as a network layer protocol while the personal broadband connection still uses IP version 4. If a device moves to a network attachment point managed by another organization, the composition of protocols in the protocol stack might change. Moving to a network that uses other protocols means that not only addresses will change but also that network address schemes will change. In the given example, when going home from work, the user's laptop must switch to IPv4 addresses because the home network cannot function with the IPv6 address scheme.

To ensure continuous communication, the transport and application layer must allow the address scheme to change. In current TCP/IP protocol stacks, transport layer protocol (TCP and UDP) implementations can already handle both IPv4 and IPv6 protocols to identify transport protocol endpoints. However, it is not possible to change the address and therefore the identification scheme of an active transport connection or open socket. By consequence, applications break when the terminal they run on is moved to a network attachment point that uses another network layer protocol, unless they know how to cope with address scheme changes themselves.

Similar to address changes, protocol stacks that are deployed in dynamic network environments must be able to handle the technical consequences of address schemes on behalf of the application. It should be noted that existing mobility solutions such as Mobile IP can not handle address scheme changes because they only target address changes. Although Mobile IPv4 and Mobile IPv6 are capable of handling address changes in their respective networks, they can not cooperate to obtain support for changing address schemes.

2.1.1.3 Coping with protocol diversity/proliferation

Currently, applications are developed with a fixed protocol stack in mind. This not only hinders protocol changes but also limits the deployment of applications in a heterogeneous network where it is unpredictable what protocols will be used when attached to a particular network access point. Different access networks may use different network layer protocols (which results in address scheme changes) or different QoS protocols. In those networks, the services may be reachable by using different transport layer protocols that have been tuned for optimal performance. Moreover, it is possible that over time new protocols will be available that can handle a particular network service better than the currently existing protocols.

The fast evolution of network hardware has made possible a whole new range of applications such as video and audio streaming, peer-to-peer networking, etc. As a result new protocols are developed to facilitate the new gamut of applications. The examples are countless: the Stream Control Transmission Protocol (SCTP) [SXM+00] has been developed to offer improved streaming services, where applications had to use UDP or TCP in the past. Recently Mobile-SCTP [SRX+05] has

been proposed to offer better support in mobile networks. The Session Initiation Protocol (SIP) [HSSR99] offer support to set up and negotiate about multimedia network connections. These protocol developments are performed in the transport and application layer. There is however a lot of work done on the the network layer too. Efforts are done to deploy IPv6 worldwide and means are developed to make it interoperate with the current base of IPv4 hardware [WC02].

The protocol stack should offer the necessary concepts and abstractions to allow application development independent of a particular protocol stack instance. Protocol stack support should encompass application deployment on a multitude of computer networks, without knowing beforehand what data link and network layer protocols will be used to provide network access, and what transport layer protocols will be used to realize the requested service. Additional to address changes and address scheme changes, the protocol stack should also allow protocols to change at application runtime because the optimal composition of protocols in the stack may depend on other factors, such as the type of network and application preferences.

Not only applications, but also mobility solutions are often developed with a fixed protocol stack in mind. Instead, mobility solutions should be *generally applicable*: they should not depend on the availability of particular protocols or protocol features. It should be noted that general applicability encompasses the challenge for address scheme changes. For example, Mobile IPv4 and Mobile IPv6 are not generally applicable because they depend on the respective protocol's address scheme. They are no longer usable if another network layer protocol is used. However, general applicability covers more than address schemes. For example, mobility solutions that are developed in the transport layer are typically only applicable to one transport protocol because they rely on particular protocol features like option headers or protocol usage scenarios (see Section 2.2.2 on Transport Layer Mobility Solutions and Section 7.1.2 for example Transport Layer Mobility Solutions). Applications that use other transport layer protocols can consequently not benefit from those mobility solutions.

2.1.2 Application awareness of mobile behavior

It is possible that network characteristics change when migrating to another network attachment point. These changes may affect the application and the application should be notified if such changes occur. Section 2.1.2.1 elaborates on application involvement.

An often overlooked characteristic of mobile networks is the fact that mobile devices may become disconnected from the network. Section 2.1.2.2 discusses how mobility solutions should notify the application of network disconnection events.

2.1.2.1 Keeping the application involved

Certain classes of applications may behave strangely or stop functioning when they are moved to another network attachment point. Moving to a different network domain can impose a problem because of the impact changes in network characteristics can have on the business logic of some applications. For example, multimedia applications such as video or audio streaming applications may suddenly be moved to a network that is characterized by a lower bandwidth, variable latency or unpredictable jitter. It is possible that such changes make it impossible to watch the video feed or to listen to the audio stream. Other applications may even be unable to function when they are deployed in a network where other protocols are used because they do not fulfill the application's network requirements. For example, an online banking application may require the presence of IPSec[KA98] to ensure secure communication. Bandwidth changes and encryption availability respectively are crucial for the service offered by such applications. It must be mentioned, however, that a change in network characteristics does not have to have an impact on all applications. A drop in bandwidth is not a fatal change for a file transfer application, for example.

Since the operating system can not anticipate what applications will be affected when attaching to a different network access point, the application will have to be involved when addressing the occurring network changes. There are two possible ways to address this problem. First, the application can be involved in the selection process of the new network attachment point. This way, the application can ensure it is never placed in an inadequate network. There is a risk, however, that an adequate network to realize the service cannot be found. With the second approach, the application has no decision power, but needs to tune its business logic when it is informed about a particular change in network characteristics. Both approaches require the exchange of network status information between the application and protocol stack. Currently, the status information that is exchanged is limited to the reporting of fatal network errors to the application.

Most popular mobility solutions try to keep mobile behavior transparent for the application, mainly not to existing applications. Protocol stack software should allow applications to optionally exchange application feedback. In case of legacy applications, the default of transparent mobile behavior is used.

2.1.2.2 Handling disconnection as a way of life

In a dynamic network it is always possible that a device loses network access, because the end user is in an area with little or no network coverage, or simply because communicating is too expensive using the available network technology. The consequence is that sometimes applications must operate without network connection. Applications like network file systems can continue operation while disconnected from the network (e.g. Coda [MES95]). Other applications, like file

transfer applications, can suspend operation until network connection becomes available again. For applications like video streaming, network suspension will probably be fatal unless the application has buffered enough video data to bridge the time of disconnection.

Current mainstream protocol stack implementations offer no support for disconnected operation. Each application itself must implement recovery mechanisms to handle the potential loss of network access, which is difficult to realize with current application feedback, network access loss is reported as a fatal network error. It is impossible for the application to distinguish between a genuine network failure (e.g. a router crashed) and the loss of network access because the terminal moved out of the vicinity of an adequate network access point. This again stresses the need for semantically richer application feedback.

Protocol stack implementations should be able to notify the application when network connection is no longer available. The application can then adapt its business logic or inform the end user that the access connection is lost. In the case of legacy applications, it should be possible to suspend the application when the access connection is lost, and resume the application when a new access point is encountered. Although this will not address disconnection with every legacy application, it will improve the service quality, also for legacy applications that are deployed in a dynamic network.

2.1.3 Securing mobile endpoint behavior

There are security implications when moving around in a dynamic network, in particular when switching between different access points that are managed by different organizations. When addresses and address schemes change, it is possible that a third party might, maliciously or not, interfere with an existing transport connection. For example, suppose that host A suddenly loses network access while communicating with host B. Host C enters the network and receives the former network layer address of host A from the DHCP server, which is a realistic situation if lease times are short. If host C starts to communicate with host B, host B should be able to differentiate host C from host A, notwithstanding they are both identified with the same network layer address in a short period of time. A malicious host C can claim that it is host A and try to hijack the communication channel that was set up between A and B. Using the network layer address to identify an endpoint in a dynamic network can be problematic.

There must be a way to establish the identity of an endpoint without using the endpoint's network layer address. Furthermore, a mechanism is required to ensure that one is communicating with the same endpoint after that endpoint moved to another network access point or returns from disconnected operation. It should not be possible for a third party to assume, by accident or intentionally, the identity of another host.

The security measures taken to establish the identity of an endpoint should provide an *attack-equivalent* network environment [Sno03]. Attack-equivalence means that, by applying the security measures, network vulnerabilities introduced by a mobility solution can be reduced to vulnerabilities that already existed in a system without the mobility solution. It is hence sufficient to provide enough security measures to ensure the issue of endpoint identification and authentication. Security measures should not try to solve other security aspects than endpoint identification and authentication.

2.1.4 Openness of heterogeneous networks

An open network is a network in which it is easy to deploy new protocols and mobility solutions. Dynamic networks are open networks because they are built using a large set of heterogeneous communication technologies and protocols. Adding a new protocol to a dynamic network should not meet a lot of problems. To obtain such open networks, protocol stacks must become more flexible, the deployment and implementation techniques of mobility solutions should be generally applicable and network infrastructure should remain as simple as possible. This section elaborates on these three aspects.

First, to be able to function in a dynamic network, a protocol stack must support the dynamic addition, upgrading and removal of protocols. If a client and server must be able to communicate with each other using every possible protocol combination, all existing protocols should be available in the protocol stack. This is not only impossible due to memory constraints, it is also unwieldy to support all possible protocols, and would require constant upgrades to support new protocols or new protocol versions. The protocol stack therefore needs to support the plugging in and out of protocols to keep the memory footprint of the protocol stack low. It should be possible to evaluate the composition of protocols in the stack so the used protocol stack is the most efficient according to application requirements and the network situation of both communicating peers.

There exist a lot of protocol stack frameworks such as Click [KMC+00], Cactus [Bha96] and DiPS+/CuPS [Mat99, Mic03, JMMV02, MJD+05] that possess or allow some or all of these properties. Therefore, these protocol stack challenges are not the goal of this work; to obtain these properties this work will appeal on DiPS+. DiPS+ is a protocol stack component framework that allows protocols to be implemented as components, which are building blocks with which a protocol stack can be implemented. DiPS+ has been evaluated in the context of protocol stack building and composition [ŞMBV03, ŞVB02]. With the support of CuPS [JMMV02] also runtime component replacement, and thus protocol replacement, is possible.

Secondly, the implementation and deployment techniques of a mobility solution must not limit its applicability. The used implementation and deployment techniques must not change the expected behavior of the computer system or introduce

unexpected side effects. *General applicability* hence does not only encompass independence of protocols and protocol features (see Section 2.1.1.3), but also requires carefully designed implementation and deployment techniques. An example of a deployment technique with problematic side effects and limited applicability is the following. Mobility solutions that are realized in the application layer are often deployed as a dynamically linked library (DLL). This library realizes the mobility solution by altering the semantics of the system calls that are called on a transport protocol socket. The mobility solution's DLL intercepts the system calls, executes mobility solution functionality for that system call, and then forwards the system call to the operating system's kernel. This approach becomes problematic when the socket's file descriptor is passed to another process that not linked with the mobile solution's DLL. To avoid this side effect, all system calls that pass file descriptors to other processes must be altered too. Also, applications that are statically linked cannot use DLLs. Section 7.1.3 discusses a number of existing mobility solutions that use this deployment technique and suffer from that side effect.

Thirdly, protocols and mobility solutions should not depend on *specialized network infrastructure*, i.e. infrastructure that is not available in the network before adding the solution. Mobile IP is an example of a mobility solution that requires specialized infrastructure. It requires a home agent on the home network router. This home agent acts on behalf of a mobile device when the device is communicating using another network attachment point. Deploying Mobile IP hence is difficult because it requires access to the core network infrastructure (the router) and requires the privileges to change the infrastructure. Mobility solutions developed in the transport layer typically do not depend on extra network infrastructure (See section 2.2.2). In some cases they depend on existing infrastructure such as a name service like DNS, but since this type of network infrastructure is typically already present in the network we do not perceive it as *specialized* infrastructure.

2.2 A Taxonomy of Mobility Solutions

This section presents a taxonomy of mobility solutions. Solutions can be categorized according to their location in the protocol stack: in the network layer, in the transport layer, in the session layer or in the application layer. Application layer solutions can be further categorized as follows: socket layer mobility solutions, proxy mobility solutions and application specific mobility solutions. All endpoint mobility solutions found in the literature can be subdivided in these categories.

This section evaluates these six solution categories with respect to the challenges introduced Section 2.1. We discuss if the solution category can cope with **network layer address changes**. When applicable, we also discuss whether and how the solution realizes **virtual circuit continuity**. Virtual circuit continuity is the capability of keeping a virtual connection that is established between two

applications alive when the location of the host changes [OMTT00], or when the host is disconnected for a longer time. Virtual circuit continuity is a property that is pursued by mobility solutions that operate in the transport layer or higher. If the virtual circuit offers reliable data transport, virtual circuit continuity also encompasses **byte stream consistency**: changes to the virtual circuit should not compromise the transport protocol's reliability service.

We also discuss if the mobility solution category supports **protocol changes**; the solution should allow applications to deal with address scheme changes. The ability to deal with address scheme changes usually automatically means that the solution category can also offer **programming abstractions and concepts** necessary to deal with protocol diversity. Whether these abstractions and concepts are offered to the application developer is solution implementation dependent. It is therefore not discussed in the scope of a solution category.

For every mobility solution category, we discuss whether it offers **application awareness** in the form of application feedback or if it aims to be **transparent** for the application. We also verify if the solution category offers support for **disconnected operation**.

We handle the security implications on the endpoint when device moves around. The **security measures** that those protocols should take to prevent abuse are shortly highlighted.

Finally, we evaluate whether the mobility solution category can be deployed easily in a heterogeneous network by examining its dependence on **additional network infrastructure**. Additional network infrastructure is additional hardware and/or software that is required for the solution to operate. Examples are foreign agents, home agents and proxy servers. Name servers are not considered to be additional infrastructure since these are already ubiquitous in contemporary networks. We also discuss whether the solution category is **generally applicable**, i.e. applicable independently of the presence of particular protocols or protocol features and free of unexpected side effects. We do not discuss whether the protocol stack supports the **dynamic addition, upgrading and removal of protocols** because that is a characteristic of a particular protocol stack implementation. It is not a property of a mobility solution category.

Before describing the solution categories, it must be noted that this section describes how the different solution types *can possibly* address these challenges. Existing mobility solutions do not usually address all challenges. A number of solutions that fit into each category are described in Chapter 7, which treats related work. These existing solutions were also used to validate the taxonomy presented in the following sections.

2.2.1 Network Layer Mobility Solutions (NLMSs)

Since host addressing is one of the tasks of the network layer, and since host address changes are one of the most acknowledged problems of mobile behavior,

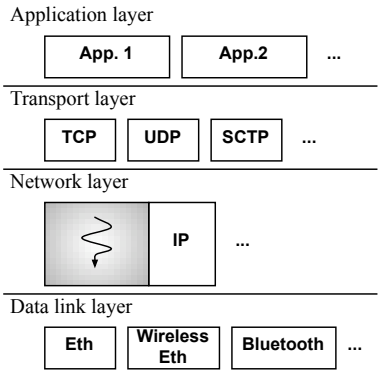


Figure 2.1: Network layer mobility solutions

the network layer is a straightforward place in the protocol stack to handle **address changes**. Figure 2.1 shows a schematic representation of a protocol stack in which a NLMS is depicted as a grey box with a curved arrow. The representation of a NMLS as a separate box attached to a network layer protocol illustrates the fact that NLMSs are usually designed as a network layer protocol add-on. Because the solution is located in the network layer, network and transport layer protocol changes are obviously not supported.

The goal of NLMSs is to be **transparent** for the upper transport and application layer and hence do not provide **application feedback**. Transparency is one of the main reasons why NLMSs are so popular. There are no adaptations required on the protocol stack's application programming interface (API) because applications use the normal network layer addresses. Consequently, this promotes every application to a mobile application. Existing network protocols also do not need to be adapted because NLMSs are an add-on. However, moving to a different network attachment point may result in degraded performance of some transport protocols [CI94]. Due to network layer solution transparency, transport layer protocols typically cannot fully adapt to the new network conditions. For example, when moving to another network, parameters, like the round trip time (RTT) and the maximum transmission unit (MTU), may change. TCP adapts to the new RTT because it continuously measures changes in the RTT. TCP typically cannot adapt to a different MTU, however. The MTU is used to calculate the connection's maximum segment size (MSS). The MTU is typically determined at connection establishment time by consulting the host's routing tables. Because NLMSs keep mobility events transparent, TCP typically is never aware of MTU changes. Consequently, the network layer may have to fragment packets if the MTU of packets on the new network is smaller.

Network layer mobility solutions do not support **disconnected operation**.

Disconnected operation is equivalent to the absence of a network address. This is also kept transparent for the layers above the network layer. Higher layer transport and application protocols may fail when they are disconnected from the network for a longer time period. For example, transport layer protocols or applications that use timers to drive their operation, e.g. to resend data when the receipt of that data has not been acknowledged by the peer, will usually fail when these timers keep expiring.

Network layer mobility solutions usually require **additional network infrastructure** to be able to route packets to the mobile hosts. In their survey of Network Layer Mobility Solutions, Bhagwat et al. [BPT96] described that all Network Layer Mobility Solutions require agent software on the edge routers of the network, operating on behalf of the mobile device. The agent software that must be deployed on the edge routers of the network is the only required adaptation for Network Layer Mobility Solutions. The mobile devices themselves do not require additional adaptations apart from the installation of the NLMS in the device's protocol stack.

Network layer solutions are not **generally applicable**. Although network layer mobility solutions are usually designed as network layer protocol add-ons without requiring adaptations to the original protocol, they depend on the used address scheme of the network layer protocol. consequently, NLMSs can not be used in a network where different network layer address schemes are used (no support for protocol diversity).

NLMSs require additional **security** measures. A malicious host could assume the identity of the mobile host. There is nothing that prevents an attacker from using the mobile host's identifying network layer address and claiming that the mobile host just moved to the attacker's location. A network layer address is hence not a sufficient means to prove one's identity. Other authentication mechanisms are required. An example authentication technology available in the network layer is IPSec[KA98]. IPSec offers packet payload encryption and endpoint authentication as an add-on to the IP Protocol. IPSec is a mandatory part of IPv6.

2.2.2 Transport Layer Mobility Solutions (TLMSs)

Transport Layer Mobility Solutions (TLMSs) handle the consequences of endpoint mobility in the transport layer, as indicated by the grey box in the transport layer on Figure 2.2, and are typically developed as an extension of an existing transport protocol. In the literature, TLMSs are mainly applied to connection-oriented protocols that offer reliable data stream services because such protocols show a higher sensitivity to failure in a mobile environment. For connectionless protocols that offer no reliability, the more popular NLMSs are often sufficient.

The main goal of transport layer mobility solutions is to offer **virtual circuit continuity**. If the used transport protocol offers reliable communication, virtual circuit continuity also encompasses **byte stream consistency**. There

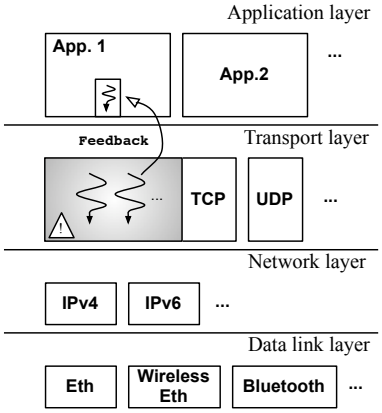


Figure 2.2: Transport layer mobility solutions

are a number of problems to ensure virtual circuit continuity. First, transport protocol connections are often identified by means of network layer and transport layer addresses. When the network layer address of one of the endpoints changes, the transport layer connection breaks because packets that carry the new address can no longer be associated with the connection. For example, a TCP connection is identified using the IP addresses of both peers and a pair of TCP ports. If one of the IP addresses changes, the TCP connection is typically reset. Secondly, when disconnected for a longer period of time, some transport layer connections are aborted when the peer party does not respond within a certain time. If byte stream consistency must be ensured, there must be a way to resynchronize the data stream.

All TLMSs deal with the fundamental problem of **network layer address changes**. When new network layer addresses are configured, transport layer protocols using network layer addresses to identify connections must reconfigure active connections with the new address. On the mobile host, all transport protocol connections are affected. The multiple curved arrows in the grey box in Figure 2.2 indicate that the mobility solution is connection based and not host based. Contrary to NLMSs, TLMS must be applied separately for each transport connection and maintain separate state for each transport protocol connection. On the correspondent host, only the transport connections that are established with the mobile host must be managed using the TLMS. NLMS only affects to host configuration state, which is the same for all transport connections. **Protocol changes** can be supported but are limited to network layer protocol changes. Transport layer protocol changes are not possible because TLMSs are part of a transport layer protocol. To support network layer protocol changes, TLMSs must be able to use

the addressing schemes of different network layer protocols for identifying transport layer connections and must be able to change these addressing schemes for active transport connections.

Contrary to network layer mobility solutions, transport layer mobility solutions can have support for **disconnected operation**. For example, where a normal transport protocol would fail because data has been transmitted too many times without response from the peer communication partner, a TLMS can suspend the connection until the peer notifies its reconnection to the network.

Transport layer mobility solutions are mostly **transparent** for the application layer. TLMSs can hide a connection failure from the application by pretending that the connection is still open but that the peer party is not sending any data. However, in case the application wishes to be informed of mobility events, the TLMS can optionally provide network status feedback. This is shown in Figure 2.2 by the feedback arrow to the application. The TLMS can inform the application when the hosting computer becomes disconnected, or when it has not received any data from the peer communication partner for a longer time period. The application can then act correspondingly and adapt its business logic (displayed by means of a curved arrow). It must be noted that an application must be altered if it wishes to receive feedback. Traditional protocol stack APIs do not offer any means to provide richer network status feedback to the applications.

TLMSs are not normally **generally applicable**. When TLMSs extend an existing protocol, they often change the protocol specification. The exclamation mark in the figure points out that the protocol specification may not be compatible with the extended transport protocol's original specification. Consequently, both communicating peers must run these extensions, otherwise the peers may not be able to communicate at all. Also, TLMSs sometimes depend on transport protocol specific functionality such as exchanging options using option headers or protocol specific protocol usage scenarios. Examples are given in Section 7.1.2.

As with NLMSs, **security measures** are necessary if a peer that participates in a transport protocol connection moves to another location. There must be a way to verify that the peer is the same peer that established the connection. Security measures must prevent a malicious party from hijacking a transport protocol connection though a security hole in the mobility solution, i.e. a malicious third party should not be able to pretend that it is the peer that initiated the transport protocol connection. Because TLMSs operate in the transport layer, these security checks must happen for every transport protocol connection that has a mobile network endpoint.

TLMSs do not require special **network infrastructure** because they are end-to-end solutions. All mobility functionality is realized by the two communicating partners. Only in the special case that both communicating endpoints move simultaneously, a name service is required. When an endpoint tries to resume communication, a name service must be available to retrieve the new network

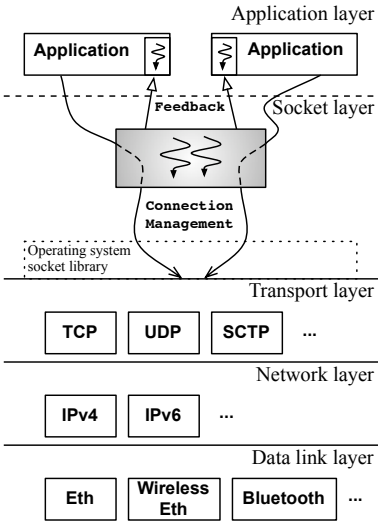


Figure 2.3: Application layer mobility solutions: the socket layer approach

layer address of the peer communication partner. The mobile host cannot use the previous network layer address because the peer is no longer reachable on the old address. The occasional need for a name service is not usually considered to be *specialized* infrastructure because name services are widely available.

2.2.3 Socket Layer Mobility Solutions (SoLMSs)

The goal of Socket Layer Mobility Solutions (SoLMSs) is to offer virtual circuit continuity to the application layer. An SoLMS is realized in the application layer as a library that offers the same API to the application as the operating system's socket API. To communicate, the application uses the operating system's socket API but calls into the mobility solution's library instead of into the operating system. The library typically does some bookkeeping in order to ensure virtual circuit continuity in the case of a mobility event. After bookkeeping, the calls are forwarded to the operating system's socket API, because SoLMSs do not implement a new protocol. Figure 2.3 shows the location of SoLMSs in the protocol stack. Note that the OSI reference model [Zim80] normally does not contain a socket layer. However because a SoLMS introduces an additional level of indirection between the application and the protocol stack, we introduce an intermediate socket layer in the application layer. The border between the application and socket layer is drawn as a dashed line to stress that the socket layer is not a real protocol stack layer. The forwarding of socket API calls is depicted by the

`Connection Management` arrows which go from the application to the mobility solution and then to the system's socket library.

Socket Layer Mobility Solutions cope with **network layer address changes**. This is realized in a very different way than transport layer or network layer mobility solutions. SoLMSs can not access the state of the transport layer protocol because they are realized above the transport layer. Hence, the SoLMS cannot make changes to the transport protocol connection identification parameters, disable timers or access buffer contents. Therefore, when the address of one communicating peer changes, the transport connection breaks. The SoLMS handles the address change by setting up a new transport layer connection using the new addresses. To ensure **virtual circuit continuity** and **byte stream consistency**, the SoLMS must take additional measures, such as double buffering [Sno03]: the content transport connection buffer that may be lost when a transport connection is aborted must also be buffered by the SoLMS. **Protocol changes** can not be supported because SoLMSs offer the system's socket API to the application. Contemporary socket APIs require the application to specify what transport protocol it wishes to use. The SoLMS is therefore restricted to use the transport layer protocol that the application requested. It can only use another transport protocol, if the SoLMS masks the differences between the protocol used to communicate and the protocol requested by the application.

SoLMSs can cope with **disconnection**. If a connection breaks and a replacement connection cannot be immediately established, the SoLMS can hide this from the application by blocking all socket API calls that are affected by the absence of a transport connection.

The aim of SoLMSs is to be **transparent** for the application. This is accomplished by using the same socket API as the one offered by the operating system. The operating system's protocol stack typically does not offer feedback about the network status to the application, apart from network errors. The SoLMS can optionally provide richer application feedback and offer it to the application by extending the system's socket API with a feedback API. In the figure, this is depicted as an arrow that goes from the SoLMS to the application.

There are some problems with the **general applicability** of this solution type [ZM02]. Applications must link with the application layer library if they wish to have mobility support. Otherwise, the normal system socket implementation will be used. This means that every application to be deployed in a mobile environment must be rebuilt. Dynamic linking techniques solves this problem but not every application binary supports dynamic linking. Keeping the SoLMS approach transparent for the application is a challenge as well. Because the SoLMS's state is not managed by the kernel a number of problems may arise when certain system calls, such as `fork` and `exec`, are called. All calls must be virtualized by the SoLMS so the application keeps working as expected when it is using the SoLMS.

Because SoLMSs establish a new replacement connection when they resume

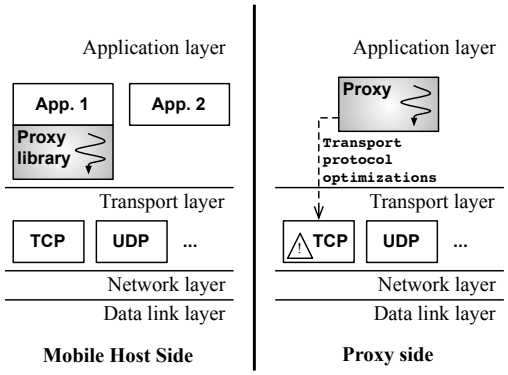


Figure 2.4: Application layer mobility solutions: the proxy approach

communication from a different location, additional **security** measures must be provided. It must be possible to verify the identity of the peer communication partner that tries to establish the replacement connection. Otherwise, nothing prevents an attacker from hijacking the communication channel established between two communicating parties.

The need for additional **network infrastructure** is limited to a name service, which is only needed when both endpoints moved. The name service provides a way to relocate the peer host when both hosts are identified using a different network layer address. This is a similar situation as with TLMSs (see Section 2.2.2).

2.2.4 Proxy Mobility Solutions (PMSs)

Proxy Mobility Solutions use a proxy server in the network that enables client mobility in a client-server networking model. PMSs are an application layer solution because a proxy server is implemented in the application layer. Like every mobility solution that is realized above the network layer, the goal of PMSs is to ensure **virtual circuit continuity** in the case of mobile clients. The proxy scheme splits a virtual circuit into two transport connections: one from the mobile client to the proxy (referred to as the client connection), and a second one from the proxy to server (referred to as the server connection). The server connection is realized on a non-mobile network and remains open until the client doesn't need the connection to the server anymore. The client uses a proxy protocol (similar to e.g. SOCKS [LGL+96]) to communicate with the proxy server. The left side of Figure 2.4 shows a mobile host with an application (application 1) that is linked with a library that implements the proxy protocol. The right side shows the proxy server with the

proxy server process running in the application layer. The client's connection can break due to mobility events. When this happens, the client's proxy library sets up a new connection to the proxy, and asks the proxy server to resume communication. The proxy server confirms this request and takes the necessary measures to ensure virtual circuit continuity. Sometimes, the used transport protocol is adapted to simplify the maintenance of **byte stream consistency** and improve the performance of the proxy server. This is indicated with the dashed arrow on the proxy side in figure 2.4. The exclamation mark indicates that the transport protocol has been adapted on the machine that runs the proxy server.

Proxy mobility solutions handle **network layer address changes** of the client. The client connection breaks when the client moves to another network attachment point. When the client notices reattachment to the network it resumes the client connection. The proxy takes the required measures to ensure **byte stream consistency**. The server connection is not affected by the client's mobile behavior because of the proxy's mediation. **Protocol changes** that may occur when the client moves between heterogeneous networks can in principle be supported as long as the proxy can bridge the gap of the semantic differences between the client and server connection. Data that is received on the client connection must be forwarded on the server connection and vice versa. If the client connection and the server connection use different protocols, the proxy must be able to handle the differences between these protocols. PMSs can also support **disconnected operation**. The proxy library can block communication operations when the client connection absent.

PMSs can easily offer optional **application awareness** because they are realized in the application layer. Every time the client connection breaks, the PMS can notify the application when desired. This does not require adaptations to the used transport protocol. The PMS can of course also keep mobility events transparent for the application.

General applicability with PMS is poor. Applications that are not designed to run in a proxy environment will have to be adapted to use the proxy solution, must be linked with the proxy library (see the left side of Figure 2.4), and must be configured to use the appropriate proxy server. Applications that already have proxy support, e.g. support for SOCKS proxy servers, will be a lot easier to adapt to the solution.

There is a proxy server required in the network, which categorizes as specialized **network infrastructure**. Because the proxy server usually is an application layer service that can be run on a normal fixed host, deployment of this infrastructure is not as intrusive as for instance the deployment of the home and foreign agents that are required for NLMS. It should be noted that a proxy server introduces triangular routing in the network: data is not exchanged directly between the two communicating peers, but must pass through the third party proxy server. The proxy server is not necessarily placed on a direct path between the mobile client

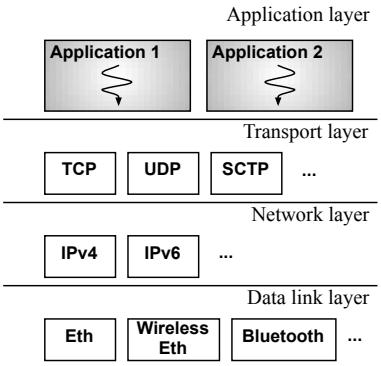


Figure 2.5: Application specific mobility solutions

and the non-mobile server. PMSs are therefore not the most performant mobility solution.

Additional **security** measures are required to prevent a malicious client from hijacking the client connection. Every time the client establishes a new client connection to the proxy server because the old one broke, it must be possible to verify that the client that establishes the new connection is the same client.

2.2.5 Application Specific Mobility Solutions (ASMSs)

Application Specific Mobility Solutions (ASMSs) are realized in the application layer as part of an application and are in the strictest sense realized without additional system support. In Figure 2.5, applications are depicted as grey boxes to indicate that they implement a mobility solution themselves without further dependencies on the system's protocol stack. The slightly different arrows in Figure 2.5 indicated that each application detects mobility events and addresses the consequences in a different way. An application that is deployed in a mobile environment handles the consequences of mobility itself. When the mobile device moves to another network attachment point or remains disconnected for a longer time, transport connections break. The application is responsible for detecting disconnection and establish a new transport connection to resume communication. It is possible however that ASMSs cooperate with other mobility solutions. These ASMSs are referred to as hybrid mobility solutions in comparison with pure ASMSs that do not depend on other mobility solutions. For example, an ASMS may cooperate with a NLMS to simplify the handling of network layer address changes. The ASMS is still required to cope with consequences of mobile behavior that are not anticipated in the NLMS like, for example, transport connection failure caused by longer periods of disconnection.

ASMSs handle **address changes** by establishing a new connection when the old connection breaks due to a network layer address change. To ensure **byte stream consistency**, the application must implement a double buffering technique [Sno03]. Since it can not access the transport protocol buffer state, the application itself must remember what data it has sent and what data was received by the peer. ASMSs can handle **protocol changes**. When a new connection must be established, this can be done using another protocol combination provided that the application has been designed to use that combination. For example, an application will not be able to use IPv6 if it has not been developed with IPv6 support. It will not be able to use SCTP is it was only designed to use TCP. The protocol changes are consequently limited by the changes that are anticipated for in the application and also by the protocols that are available in the system's protocol stack.

ASMSs can handle **disconnection**. When the application notices a disconnection it can act correspondingly and try to continue operating in a disconnected mode, or block until network access returns. Because the application must handle disconnection itself, it is also fully qualified to deal with the consequences of network disconnection on the business logic.

Because the application must handle the consequences of mobility itself, ASMSs are by definition **not transparent** for the application. Solutions are also application specific and are therefore not **generally applicable**. On the other hand, ASMSs confirm Saltzer's end-to-end argument [SRC84]: because these solutions are tailored to the application, they are the most optimal mobility solutions.

ASMSs must be able to verify the identify of peer applications that moved to a different location. Such **security** measures will be different for all applications, because the application designer can tailor them to the application.

ASMSs typically do not require specialized **network infrastructure** to operate. The solution involves the mobile endpoints only. However, in case both mobile devices that host the applications move, a name service is required so the ASMS can retrieve the new network layer address of the peer mobile device.

2.2.6 Session Layer Mobility Solutions (SeLMSs)

Session layer mobility solutions (SeLMS) introduce a session layer in the protocol stack, as described in the OSI reference model [Zim80]. Because in contemporary protocol stacks there is no presentation layer, the session layer is located between the transport layer and the application layer (see Figure 2.6). It should be noted that, although they are realized in the application layer, SoLMSs and PMSs also introduce a (virtual) layer between the transport layer and the actual applications. However, SeLMSs cope with mobile behavior in a different way: SeLMSs introduce the notion of a session to the application. Applications that run on a protocol stack containing a SeLMS explicitly establish sessions with each other instead of transport protocol connections. A session is designed to survive network failures

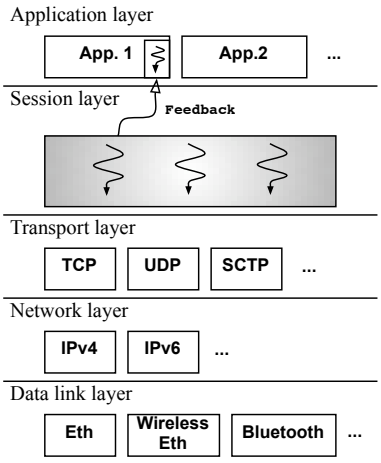


Figure 2.6: Session layer mobility solutions

that stem from mobile behavior (e.g. network layer address changes). SoLMSs and PMSs change the semantics of a transport protocol connection so they can cope with endpoint mobility. Like SoLMSs and PMSs, SeLMSs use the services from the lower transport layer. However a session is realized by *normal* transport protocol connections. SeLMSs do not have to change the semantics of the transport layer connections to be able to realize endpoint mobility. Also, a session can be realized by multiple transport connections that are active simultaneously. Contrary to PMSs, SeLMSs are pure end-to-end solutions. They do not need proxy servers.

Address changes are handled when the session's transport connection breaks because of the movement of one of the communicating endpoints. A new transport connection is established and the session layer takes care of **byte stream consistency**, for example by double buffering [Sno03] the data. **Protocol changes** can be supported because a session represents a logical connection. The used communication protocols and access technologies to realize that session can change during the session's lifetime. The session is responsible for masking the protocol differences and offering the required network service, such as a reliable data stream service or a datagram service, to the applications.

SeLMS can handle **disconnection** for the same reason they support protocol changes. The session survives connection failures, even when such failures cover a large time period. For the application the session is still active even if there is no network connection available.

To establish a session, applications must use a session socket to interface with the system's protocol stack. The session socket offers an API that resembles the API of a connection-oriented transport protocol. The session socket offers the same

communication primitives as a connection oriented transport protocol socket to establish sessions. The session socket is extended with functionality to configure application awareness.

The session layer can also offer support for **applications awareness**. If the application wishes to be informed of mobility events, it can optionally receive network status feedback from the session layer. In Figure 2.6 this is depicted using a feedback arrow to an application that chooses to receive feedback. All mobility events that happen during the lifetime of a session can also be kept **transparent** for the application. A SeLMS can offer mobility support for legacy applications by transparently and automatically creating a session when a transport protocol connection is requested. The SeLMS hides mobility events from the application.

SeLMSs are **generally applicable** because they do not depend on particular protocols or protocol features. Because they are realized in the protocol stack, there is no need to need for virtualization of system calls, like with SoLMSs. The main reason for such virtualization is because SoLMSs are not located in the operating system's protocol stack but in the application layer. **Security** is again required for this solution to prevent malicious users from hijacking the session. Because SeLMS are end-to-end solutions, security must be enforced for every session. When a mobile device moves and wishes to resume communication from its new location, it first establishes a new transport protocol connection for every session. When successful, the mobile devices must prove to all its peers that it was the host that established the session.

SeLMS, like all other end-to-end solutions, do not require specialized **network infrastructure**. A name service is needed to find the network layer address of the peer host in case both mobile devices move simultaneously.

2.2.7 Summary

The matrix in Figure 2.7 summarizes how the different solution types realize the challenges specified in Section 2.1. It can be seen that the higher a mobility solution is applied in the protocol stack, the more challenges it can realize. Too high in the protocol stack, in the application layer, appears to be problematic. All mobility solutions realized in the application layer (SoLMSs, PMSs and ASMSs) have problems coping with general applicability. PMSs also depend on additional infrastructure. The SeLMS category is the only mobility solution type that addresses the general applicability challenge.

Challenge \ Solution Category	Network layer	Transport layer	Socket layer	Application layer proxy	Application specific	Session layer
Address changes	yes	yes	yes	yes	yes	yes
Protocol changes	no	network layer protocols only	yes	yes	yes	yes
General applicability	protocol dependent	modified protocol specification	no	no	application specific	yes
Application awareness	no	yes	yes	yes	yes	yes
Disconnected operation	no	yes	yes	yes	yes	yes
Security measures	per host authentication	virtual circuit authentication	virtual circuit authentication	virtual circuit authentication	application specific	session authentication
Network infrastructure	home/foreign agent	no	no	yes	no	no

Figure 2.7: Solution taxonomy summary

Chapter 3

A session based networking environment

This chapter describes a general architecture for Session Layer Mobility Solutions (SeLMSs). This general architecture together with the realization of the SeLMS form the main contribution of this work. Section 3.1 shortly motivates our choice for a SeLMS. This motivation is mainly based on the taxonomy of mobility solutions presented in Section 2.2. Before describing the session layer architecture, Section 3.2 first outlines a number of best practices and design strategies for mobility solutions that can be found in the literature. Section 3.3 then defines the concept of a session. Section 3.4 discusses the required architectural properties of Session Layer Mobility Solutions. Section 3.5 describes the tasks that must be accomplished in the proposed architecture. Section 3.6 evaluates if and how these tasks address the challenges discussed in Chapter 2. This chapter concludes with an overview of the SeLMS that is realized in this work and adheres to the proposed architecture in Section 3.7.

3.1 Motivation for a session layer approach

The end-to-end argument of system's design [SRC84] states that communication functions in a layered system must be placed intelligently in the appropriate layer. Placing functionality in lower layers may be redundant or of little value compared to the cost of providing it at that level. Certain communication functionality can best be implemented by the application at the endpoints because the application is aware of all the requirements. A session layer solution is an end-to-end solution that is still not realized in the application layer. We motivate in further detail why a mobility solution above the transport layer but below the application layer was chosen.

The taxonomy of mobility solutions described in Section 2.2 indicates that it is easier to support the challenges described in Section 2.1 when introducing a specialized session layer in the protocol stack. We again stress the advantage of the general applicability property of Session Layer Mobility Solutions. First, when working in a dynamic multi-protocol environment, it is not feasible to use solutions that depend on specific protocols in the stack because one can not rely on the availability of these protocols. Secondly, non-session layer solutions which offer session layer functionality and are realized above the transport layer can not be used by all possible applications. We elaborate on these limitations.

Solutions that address mobility in the network and transport layer are realized as extensions of existing protocols. For example, Mobile IPv4 [Per96, Per02] or Mobile IPv6 [JPA04] depend on IPv4 and IPv6 respectively. Interoperation between the two solutions is problematic. Migratory TCP is only applicable to TCP. Other transport protocols can not benefit from that solution.

Other, also network and transport layer protocol independent, mobility solutions are located above the transport layer with the valid reason that no adaptations are required to the operating system's protocol stack. Such adaptations always impede backward compatibility. However, the literature teaches us that such solutions (e.g. Rocks [ZM02], see Section 7.1.3.1) are still not generally applicable. These solutions mainly augment the communication API that is offered by the operating system with session layer functionality in an ad-hoc way by means of library interpositioning techniques: the system calls are intercepted, and additional bookkeeping is done to offer a session concept to the application. A number of technical implementation issues arise which hinder the applicability of such approaches on existing and new applications. Applications use special operating system constructs such as pipes, sockets, shared memory which do not interoperate well with such system call interception approaches.

These two reasons lead to the choice of addressing the dynamic network challenges using a transport and network protocol independent session layer in the protocol stack. On the session layer it is possible to define a mobility solution that is independent of the used transport and network layer protocols. Consequently, this mobility solution can be applied in practically every network environment because they do not assume the use of particular communication protocols. By putting the functionality in the protocol stack instead of offering it in the application layer, one does not have to cope with applications that suddenly break because the semantics of the used operating system mechanisms changed.

3.2 Best practices and design strategies

Although a lot of research has already been conducted on systems that enable mobile endpoint behavior, session layer mobility solutions (SeLMS) are still rare. Nevertheless, a number of best practices and design strategies to an appropriate

network architecture to support mobile Internet services have been introduced in [SBK01, Sno03]. This section summarizes these practices and strategies, which are closely related to the formulated challenges in Section 2.1.

3.2.1 Limit lower layer dependencies from higher layers

Higher layers in the protocol stack depend too much on state that resides in lower layers of the protocol stacks. The most prominent example of this problem is TCP that uses the network layer's IP addresses to identify its transport protocol connections. This impedes the identification of TCP connection when the IP address changes.

Better cooperation is required between the higher and lower layers in the protocol stack. On the one hand, if a transport layer protocol uses network layer protocol state, the transport layer protocol must allow this state to change. On the other hand, network layer protocols should expose relevant changes to the higher layers. In the context of mobile computing, this means that TCP must allow IP addresses to change in the TPC connection identification scheme, and IP must not keep IP address changes hidden or transparent from TCP.

3.2.2 Handle unexpected disconnections gracefully

Disconnected operation is an area that has received little attention. Disconnections are unexpected and can last longer periods of time. Most protocol stack mobility solutions, do not address disconnected operation.

Mobility systems should provide support for disconnections and reconnections, even when they happen unexpectedly. This allows applications to do more appropriate resource management, releasing system resources and reallocating them when network access returns, and to respond quicker to changing network conditions.

3.2.3 Do not restrict the choice of naming techniques for mobile nodes

Naming services are a fundamental part of mobility solutions because they allow to locate the mobile host. The problem is that mobility solutions and their naming solutions are often tightly coupled. Without that particular name service, the mobility solution can not function properly.

One name service is usually not suitable for all application classes. This is proven by the large amount of name services that is available. The used naming scheme should be determined by the application and not by the used mobility solution.

Name services are also often misused to obtain mobility support. Mobility solutions realize two operations in which name services can be involved: a *location*

operation and a *tracking* operation. The location operation is the process of finding a network endpoint when wanting to communicate with it. The tracking operation is the process of preserving the communication while the endpoints are mobile. The location task always requires a name service. The tracking task should be done by the communicating endpoints themselves. Only in the rare case that both endpoints move simultaneously, a third party (the name service) must be consulted to relocate the other endpoint. Mobility solutions that depend inexorably on a location resolution services during the entire communication session suffer from significant overhead. For example, Mobile IP depends on a home agent to track the whereabouts of the mobile host. In the absence of route optimization [PJ01], each packet destined for the mobile host is processed by the home agent.

3.2.4 Provide support at the endpoints

Mobility solutions that require extra network infrastructure usually suffer from degraded performance. Proxy based solutions (a.o. MSOCKS, ...) and also Mobile IP [Per96] are easy to deploy and can transparently provide mobility functionality without interfering with legacy systems. However, for such solutions to be performant, the proxy and the home and foreign agents must be well engineered and located appropriately in the network. In the case of mobility, the location of the proxy and agents must even be able to change locations if it were to be performant solutions.

These problems can be avoided if mobility functionality is provided on the mobile host itself. This reduces dependencies on extra network infrastructure which often offers a solution with a suboptimal performance.

3.2.5 Optimize for the static case

Endpoints that rarely move should not suffer significant performance penalties caused by a mobility solution. Many mobility solutions add a significant amount of network communication, even if endpoints do not move during communication.

Such solution designs stem from over-generalization of endpoint mobility. Many endpoints do not change network attachment points more often than is generally assumed and are mainly static in the network. If they do change attachment points a lot, data-link layer mobility techniques are often available to speed up mobility: horizontal handovers between access points in the same management domain are handled on the data-link layer. Consequently, the endpoint still assumes it is connected to the same network attachment point and did not move at all.

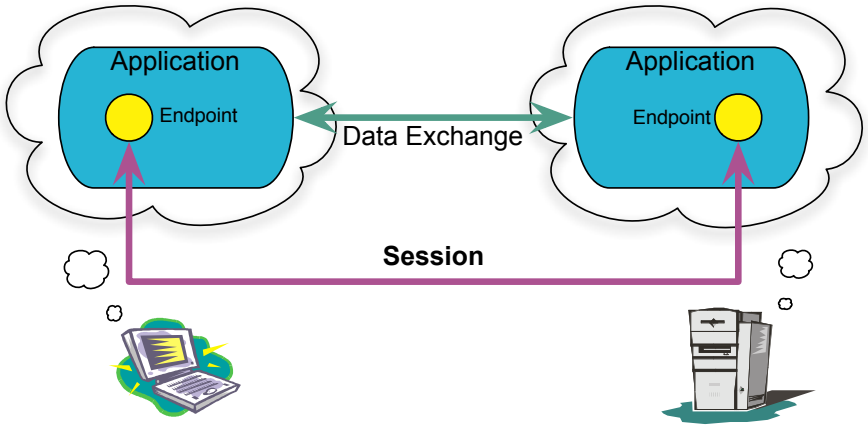


Figure 3.1: A session is a logical communication channel between two endpoints owned by two applications.

3.3 Session definition

This work specifies a session based approach to cope with the challenges outlined in the previous chapter. We first define what a session is. We then describe the static and dynamic properties of a session and outline how an application can use such a session.

A session is a logical communication channel between two communicating endpoints in a heterogeneous network. A communication channel is used to exchange data on a computer network. A *logical* communication channel defines the type of communication that will be used to exchange data, but does not specify the actual (transport, network and data link layer) communication protocols that will be used to realize the data exchange. Example communication types are datagram or stream-oriented communication and reliable or best-effort communication. An endpoint terminates a communication channel and is created and owned by an application on a network device. Two applications that wish to communicate each create an endpoint and establish a session with them. This is depicted in Figure 3.1. In this dissertation, an endpoint is always owned by the same application. That application is never moved to another host. Endpoint movement is always the consequence of device mobility. Theoretically, sessions can involve more than one party, for example, when multiple network endpoints are using multicast technology. In this dissertation, a session will only be set up between two endpoints. A session is *not* an application specific abstraction such as a session on a shopping website, an online bank transaction, an FTP session or a video conference session.

A session has three *static* properties, properties that never change throughout its lifetime. First, a session is identified using an identifier that is unique in the entire heterogeneous network and is independent of the identifiers used by transport and network layer protocols. This identifier never changes in the lifetime of a session, not even after one of the endpoints moved. Secondly, a session is characterized by its two endpoints. The endpoints never change during the session's lifetime. Thirdly, the communication type of the communication channel is a static property of a session. When a session is created, a communication type must be chosen. If another communication type is preferred, a new session must be created.

A session has two *dynamic* properties that can change during the session's lifetime: a physical communication channel and the session's state. The physical communication channel is the most important dynamic property and will be explained first. A session is associated with a physical communication channel that realizes the session's communication type. This physical communication channel is realized by a transport protocol connection that is established between the devices that host the session's endpoints.

In this work, a transport protocol connection (TPC) is defined as a communication channel between two endpoints that is realized using a transport layer protocol. For example, a TCP connection is a transport protocol connection that offers a reliable data stream. UDP packets exchanged between two endpoints is an example of a transport protocol connection that offers a best effort, datagram communication service. Note that transport protocol *connections* also include communication channels that are realized by non-connection-oriented protocols. The use of datagram protocols does not require establishing a connection first. Although it is possible to send data to more than one endpoint from a single datagram protocol socket, datagram protocols are often used instead of connection-oriented protocols to exchange data between only two network endpoints. Datagram protocols, such as UDP, are used instead of popular connection-oriented transport protocols, such as for example TCP, because their services are too restrictive for the application's business logic. For example, TCP's reliability is not required when playing a network game or when watching a video stream. On the contrary, TCP's resends can make the game impossible to play or the video stream impossible to watch. Packet loss can be tolerated in such applications.

A session is associated with *at most one* TPC. At a certain moment in time, a session temporarily may not have an associated TPC. In a dynamic network, TPCs fail if the used transport layer protocol does not contain functionality to cope with mobile endpoint behavior. For example, if one of the session's endpoints moves to another network attachment point, address changes can occur that abort the TPC. The TPC may also be aborted if one of the endpoints has been disconnected from the network for too long (see Section 2.1.1.1). If the TPC is aborted, the session is no longer associated with a TPC and a new TPC must be established

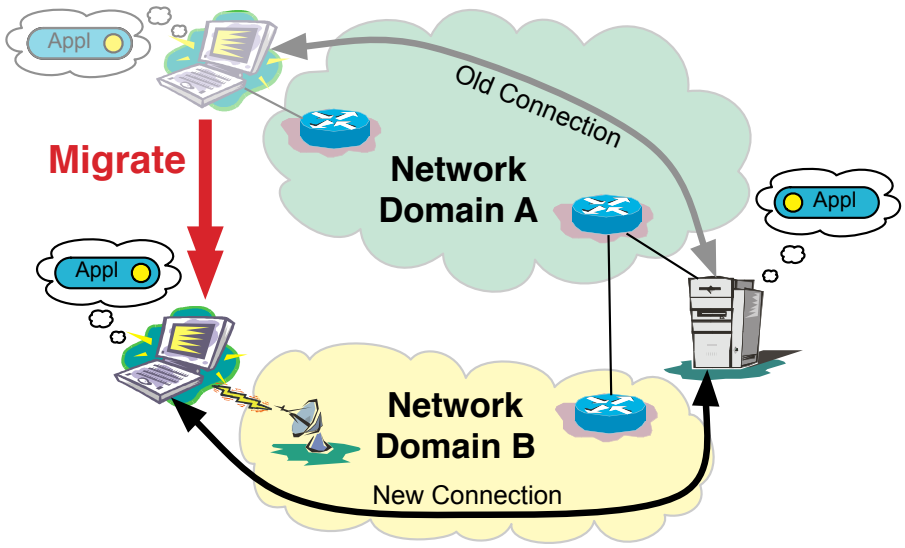


Figure 3.2: A session is associated with a transport protocol connection that is replaced by a new one each time the session's endpoint moves.

to resume communication. It must be noted that an aborted connection cannot guarantee that all transmitted data was successfully received. If the session's communication type ensures reliable communication, some data that was sent on the aborted connection may need to be resent on the new TPC. Figure 3.2 shows an endpoint movement scenario. When the laptop from Figure 3.1 was attached to network domain A, a TPC was established with the server to realize data communication for the session with the server. The laptop was then moved to network domain B (indicated by the Migrate arrow), which broke the TPC. A new TPC was subsequently established so communication could continue.

The second dynamic property is the session's state. A session's state is related to the status of the associated TPC. Generally, a session can be in the `Closed`, `Connecting`, `Active`, `Suspended` or `Reconnecting` state. If a session is created it is in the `Closed` state. If a TPC is being established for a new session, the session is in the `Connecting` state. During communication, a session is `Active` and has a valid TPC. If a session is not associated with a TPC it cannot communicate and is subsequently `Suspended`. If a new TPC is being established after the session was suspended, the session is in the `Reconnecting` state. If the reconnection is successful, the state moves to the `Active` state. Otherwise, it moves back to the `Suspended` state.

An application must be able to perform a number of operations with a session. An application must be able to *create* a session with another party and *destroy* a session. It must be possible to *send* and *receive* data during a session. Although a session is suspended when network access is lost and resumed when access returns, an application may wish to *suspend* or *resume* an session explicitly, for example, because it can anticipate network access loss. When desired, the application must be *notified* when suspension or resumption happens.

3.4 Session Layer Mobility Solution architecture

The mobility solution proposed in this work is a session layer mobility solution (SeLMS) which offers an *end-to-end solution* to cope with the dynamic network challenges outlined in Chapter 2.

A system that offers session functionality to the application is, according to the OSI reference model [Zim80], located in OSI layer 5: the *session layer*, located between the transport layer and the presentation layer. Hence, we place the solution in the session layer. The popular TCP/IP protocol stack model does not contain a session nor a presentation layer. However, the functionality of the SeLMS contributed by this work does not fit in the transport layer nor in the application layer. We therefore add a session layer to the TCP/IP model between the transport and the application layer.

Figure 3.3 depicts the developed SeLMS architecture. The *System View* column in the picture illustrates the responsibilities of a SeLMS with respect to their location in the protocol stack, shown in the *Protocol Stack* column. The left column indicates the communicating channel type the respective protocol stack layers realize: logical communication or physical communication. The remainder of this section will discuss this figure in greater detail.

Sessions are established between applications (application layer in Figure 3.3) that wish to exchange data with each other. These applications create an endpoint and can then establish a session using those endpoints. The protocol stack offers this service to the applications by means of a *session socket*, depicted on the figure as a box on the edge of the application and the session layer. This session socket represents an endpoint and offers an interface that is similar to the traditional socket API to communicate. Using this session socket, an application can create a session with a particular communication type, establish and close a session, create server (listening) sessions, send and receive data, etc. Additionally, the socket can optionally offer function calls to control the session layer solution's behavior in case dynamic network events happen: the application can optionally suspend and resume the session explicitly. The session socket also offers the means to provide the application feedback in case of network status changes (network access drops, returns, bandwidth drops, ...). Such feedback is optional: some applications may not wish to receive feedback or may not be capable of handling feedback.

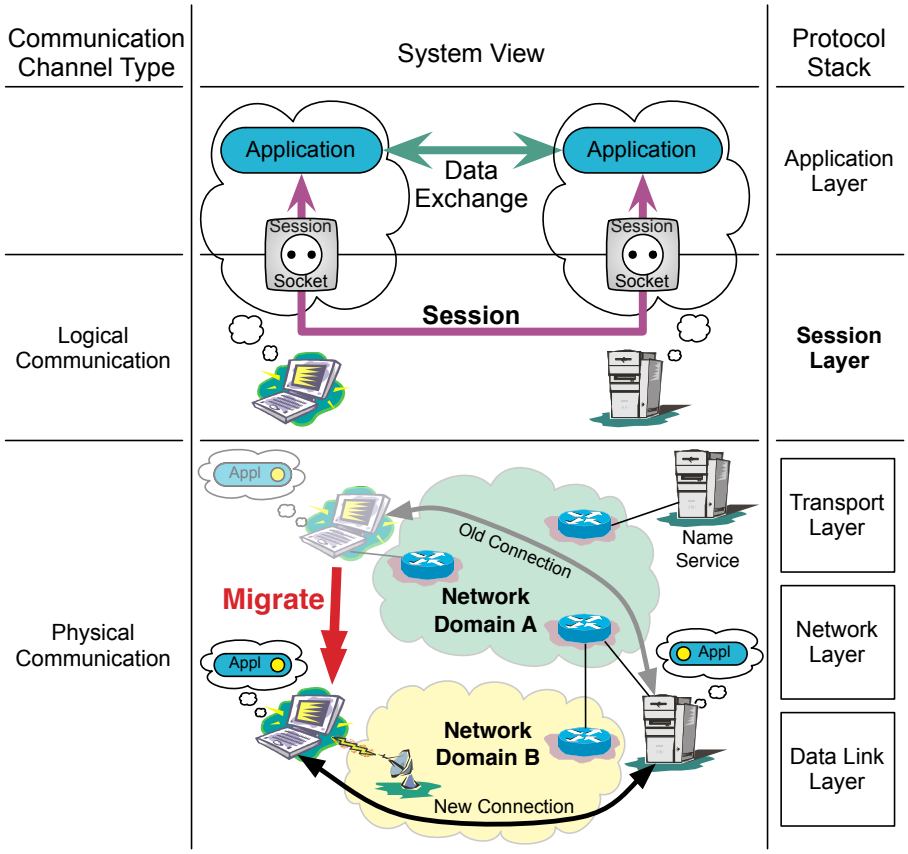


Figure 3.3: A Session Layer Mobility Solution resides in the session layer in the protocol stack. Session functionality is offered to the application by means of a session socket. The Session Layer Mobility Solution uses the services of the lower transport, network and data link layer protocols to establish physical communication channels.

The actual SeLMS is located in the session layer of the protocol stack. Its main responsibility is the realization of sessions: logical communication channels between two session sockets. The SeLMS maintains all the static properties of the session (session identifier, endpoints and communication type) and is responsible for managing the session's state, which represents the relation with the session's physical communication channel (see Section 3.3).

The SeLMS uses the services of the lower protocol stack layers. The transport protocol connections (TPCs) realized by the transport, network and data link layer protocols are used to realize a session's physical communication channels. The bottom row of Figure 3.3 depicts this. During normal communication, i.e. when the device is not switching to another network attachment point, the protocols on the lower protocol stack layers realize the session's communication type. The SeLMS monitors the communication and only intervenes to ensure the communication type in case the TPC aborts. Concretely, if an application requested a reliable data stream, a combination of transport, network and data link layer protocols will be used that realize that communication type. If the TPC aborts, the SeLMS will try to establish a new TPC and resynchronize the data stream to ensure byte stream consistency.

It is essential to track network endpoint movements in an unpredictable dynamic network. An endpoint that moves to another location can contact the peer endpoint from its new location and resume communication. If both endpoints moved, they must be able to locate each other, because they are possibly no longer reachable at their old address. Therefore, a name service will be used to track the movement network endpoints. If the address or address scheme of an endpoint changed, that endpoint must update this information on a name service. In Figure 3.3, the name service is depicted in the System View column, in the Physical Communication row because the name service is a supporting service to establish and re-establish physical communication channels. The type of name service system that must be used is not determined beforehand and can be chosen according to application preference. It must be noted, however, that the name service is preferably a fast converging system that quickly reflects changes in the network situation.

3.5 Session management tasks in dynamic networks

To realize the proposed architecture, a number of tasks must be taken care of. These tasks are the following:

- Session support detection
- Transport and network protocol independent session identification

- Protocol and address hiding
- Session state management
- Session negotiation protocol
- Transport protocol management
- Maintaining communication channel semantics
- Offering application feedback

These tasks are discussed in greater detail in the following sections.

3.5.1 Session support detection

A SeLMS must be able to detect whether the peer communication device is equipped with the SeLMS too. It is unrealistic to think that every protocol stack will contain the SeLMS, especially because existing protocol stacks will not be replaced overnight or at all. If the peer system is not SeLMS enabled, it will not be able to engage in the session protocol. Consequently, applications will not be able to communicate at all.

If a SeLMS cannot detect a peer SeLMS, the protocol stack must fall back on using the legacy communication system because legacy communication is better than no communication at all.

3.5.2 Transport and network protocol independent session identification

A Session Layer Mobility Solution must implement a session identification mechanism that is independent of the identification of the session's associated transport protocol connection. Sessions cannot rely on the identification scheme used by the TPCs because the TPC identification parameters may not persist the entire lifetime of a session. The TPC may change its identification parameters provided that the protocol supports this, or it may be replaced by a new TPC. Regardless of these changes, it must always be possible to identify the session. Note that transport layer protocols often use a network layer address as part of their identification scheme. Transport layer protocols therefore cope badly with network layer address changes (explained in Section 2.1.1.1). SeLMS should not depend on lower layer identification schemes for reasons of protocol and address changes.

Similar to protocols in the network layer and the transport layer, which use network layer address and transport protocol ports respectively, solutions that exist in the session layer must implement their own identification scheme. This will allow the peer SeLMS to identify the session and also the session socket that the SeLMS is handling.

3.5.3 Protocol and address hiding

The necessary precautions must be taken to hide the used addresses from the application because they are subject to change and do not directly affect the application's business logic. Such changes are technical consequences of mobility and do not directly affect the application's business logic. The application should hence not be bothered with address changes. Instead these changes should be handled by the protocol stack, where they originate.

Protocol changes usually boil down to address changes, but of another type (e.g. transition from IPv4 to IPv6), so hiding protocol changes from the application is similar to hiding address changes. However, it is possible that different protocols, offering a different solution to the same problem, use other semantics. For example, RSVP and MPLS are protocols that both offer quality of service (QoS) guarantees, but where the former allows to reserve specific resources, the latter puts network traffic in a number predefined classes. In case protocol changes involve protocols that use different semantics, additional measures must be taken to keep such changes hidden. If an endpoint is moved from a network that uses RSVP to a network that only supports MPLS, QoS-aware applications must be able to use both protocols.

Keeping semantic differences of different solutions hidden is a very domain specific problem and will not be handled in this work. However, to show that this is an acknowledged problem, we give an example of a solution that addresses semantic differences in the QoS domain. The generic QoS (GQoS) framework [Hua01] developed by Microsoft is part of the Winsock 2 specification and enables applications to specify their quality of service requirements in a generic way (in terms of bandwidth, delay, delay variation, service type, ...) independently of a particular QoS protocol. These service requirement are realized by means of the available QoS technologies in the access network. This decoupling of QoS policy and realization by means of particular QoS technologies simplifies changing these technologies when moving in a heterogeneous QoS-enabled network. The GQoS framework does not support runtime protocol changes, because it was not intended to be deployed in a dynamic network. However, it offers the required application abstractions to cope with different QoS protocols. The SeLMS must only provide the additional behavior to cope with runtime changes.

3.5.4 Session state management

Depending on the network status, a session is in a particular state. A SeLMS must maintain the state for every session that it is responsible for. Figure 3.4 shows the states of a session in function of the network status: `Closed`, `Connecting`, `Active`, `Suspended` or `Reconnecting`. Every state needs a different type of monitoring. Depending on the monitoring outcome, actions may have to be taken that change the session state.

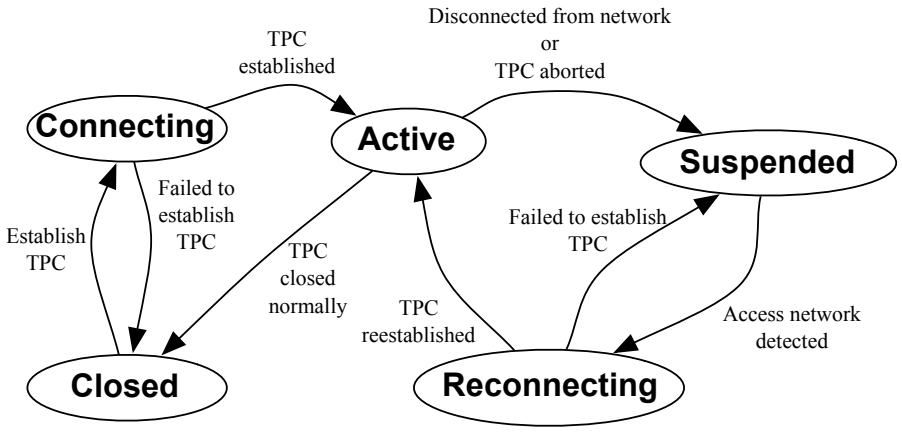


Figure 3.4: Session state management

When the a session is created, it is in the `Closed` state. This means that the session does not yet have an associated transport protocol connection (TPC). When the SeLMS tries to establish this TPC, the session is moved to the `Connecting` state. If establishing the TPC succeeds, the session becomes `Active`. If not, the session is moved back to the `Closed` state.

In the `Active` state, the SeLMS must continuously monitor for changes in the network status. If the network access connection on a particular network interface is dropped, every session with a transport protocol connection using the affected interface must be moved to the `Suspended` state. A sudden abortion of the TPC indicates that the peer network point became disconnected and the session that holds that connection must be `Suspended`.

When disconnected, the SeLMS must detect network reconnection or find an alternative way to get reconnected. When a new network access connection becomes available, the SeLMS must try to resume all suspended sessions. While the SeLMS is attempting to establish a new TPC, the session is in the `Reconnecting` state. If the TPC is successfully established, the session can resume communication and is placed again in the `Active` state. If the reconnection attempt fails using the newly available network interface, the session is put in the `Suspended` state.

When the session is no longer needed, the SeLMS can close the TPC. When the TPC is closed, the session is moved to `Closed` state and the SeLMS can destroy the session.

3.5.5 Session negotiation protocol

Before data can be exchanged, both communicating endpoints must agree on the establishment of a session. Setting up a session must be done adhering to a protocol. This can be compared with connection-oriented transport layer protocols that must first set up a connection. Next to the establishment of sessions, a session protocol must also define the protocol for the resumption of sessions that have become suspended due to dynamic network events. Optionally, also the suspension of a session can be defined in a protocol. In some cases network disconnection can be anticipated by one of the peers. Communicating this to the peer party can simplify session suspension. Finally, the session protocol must specify how to terminate a session.

The session establishment protocol must encompass the creation of the session's endpoints and the exchange of a session identifier. Session endpoints are created when applications create the session socket. The session identifier is exchanged when two applications establish a session using their socket. This identifier is used by both peers to distinguish the session's network traffic. This traffic consists of application data and session protocol messages. The session identifier is also used locally to identify the session for managing the session's state, and sending feedback to the application owning the session.

If the session is suspended anticipatedly, the suspending party must notify the peer. The session suspension protocol requires agreeing on the suspension of a session with a particular identifier. When both parties have agreed on suspension, the associated TPC can be torn down normally, leaving both parties in a stable state. In case of unexpected suspension, the TPC is aborted when disconnection has been detected or suspended when the transport protocol supports this. Usually only the peer that detected disconnection from the network can immediately abort or suspend the TPC. The session on the disconnecting party can then also immediately be suspended. Its peer will only notice that the disconnecting party is no longer there if its associated TPC aborts or suspends because of timeouts.

The session resumption protocol encompasses setting up a new TPC and informing the peer party that this TPC must be associated to a particular session. To realize this association, the session identifier must be exchanged. Once the TPC is associated with the correct session, both endpoints may have to synchronize before resuming the data exchange so correct communication semantics can be ensured (see Section 3.5.7).

The session termination protocol is invoked when the communicating peers stop communicating. Both parties must consent to terminate the session. When both parties have agreed, the TPC can be torn down normally. The parties can then destroy the session socket or reuse it to establish a new session.

The session establishment, suspension, resumption and termination protocol actions must all happen in a secure way. A third party must not be able to interfere in the protocol. For example, a third party must not be able to suspend

a session or resume a session from another location.

3.5.6 Transport protocol management

During the lifetime of a session, data is exchanged between SeLMSs and the applications that created the session. The exchanged data encompasses session management data and application data. Both types of data are exchanged using normal transport protocol connections (TPCs).

During session establishment, a TPC must be created to negotiate the session with the peer. When session establishment has completed, data can be exchanged using the same or another TPC. When suspending, the TPC is suspended (if supported by the transport protocol), torn down or aborted, depending on the network event that happened. When resuming communication, the TPC must be resumed (if the transport protocol supports this) or a new TPC must be established to negotiate session resumption with the peer SeLMS.

The SeLMS must be able to work with different types of TPCs. Depending on what kind of service the application requests from the protocol stack, a different transport protocol must be used. A session created on behalf of an application that requests a reliable data stream service will use a connection oriented reliable transport protocol. An application that does not need reliable communication will communicate using a session that employs a connectionless datagram protocol.

Hence, the SeLMS must cooperate with the transport layer to establish TPCs, send and receive data, tear down TPCs, and notice when TPCs are aborted. Because the SeLMS is not an application layer system but is designed to be on a layer in the protocol stack immediately above the transport layer, the use of traditional application layer sockets may not be possible or adequate.

3.5.7 Maintaining communication channel semantics

Reliable communication over longer and repeated periods of disconnection must be guaranteed. When a session is suspended because its TPC was terminated abruptly, some data in transit might be lost. This is a problem if the application requested a reliable data stream service. If the application requested a non-reliable data transmission service, data loss due to mobile endpoint behavior is not considered problematic.

A SeLMS must ensure the service that the application requested from the protocol stack. Some transport protocols support the sudden disappearance and reappearance on another access network of a network endpoint. TPCs that use such transport protocols are more adequate to be used in a dynamic network environment. When such transport protocols are used, no additional measures are needed to guarantee a reliable transport service because the transport protocol already offers the necessary means, provided the same transport protocol will be used after reconnection.

However, most transport protocols can not cope with dynamic network behavior. Most TPCs are aborted when the endpoint moves and all connection state is lost, including the knowledge of what data both network endpoints have already sent and received. When resuming the connection, this knowledge is required when a reliable data stream service was requested. If such transport protocols are used, the SeLMS must maintain that knowledge instead. When a replacement TPC is established, both SeLMSs can then synchronize before resuming data transfer and hence ensure the correct data stream service.

3.5.8 Offering application feedback

The application may wish to be informed in case network events affect the application's business logic. Such events can be bandwidth changes, or QoS changes in general, and network disconnection. Application feedback is not obligatory because not every network event affects all applications. For example, bandwidth drops or session suspension are not a problem for file transfer applications. Other applications may benefit from a system that informs the application of bandwidth changes. A Voice over IP application may want to know what type of network is used to communicate, in order to adapt voice compression and bandwidth reservation with respect to the capabilities of the currently used access network.

A SeLMS should offer the *optional* possibility to inform the application of relevant network events. Usually, such network events trigger a state change in the state chart of Figure 3.4. For example, an application may wish to be informed if network disconnection occurs. If network disconnection occurs, the session's state will change from `Active` to `Suspended`.

3.6 Evaluation

This section describes how the proposed architecture addresses the challenges for dynamic networks proposed in Section 2.1. The session layer nature of the solution and the session management tasks described in Section 3.5 both contribute to the realization of the challenges.

First, the proposed architecture deals with **address changes** in the following way. The Protocol and address hiding task (Section 3.5.3) ensures that the application does not have to deal with the technical consequences of address changes. These changes are handled by the SeLMS. A session uses transport protocol connections (TPCs) to realize communication. If the session's TPC breaks because the network layer address of the mobile device changed, the session can be easily resumed. When possible (Session state management task, Section 3.5.4), a new TPC is established (Transport protocol management task, Section 3.5.6) and the session is resumed by exchanging the session's identifier using the Session negotiation protocol task (Section 3.5.5). Resuming a session from a different network

address is easy because a session is identified independently of its associated TPC. This is realized by the Transport and network protocol independent session identification task (Section 3.5.2). **Virtual circuit continuity** and **byte stream consistency** are guaranteed by the Communication semantics maintenance task (Section 3.5.7).

Protocol changes and protocol diversity/proliferation are realized by the Protocol hiding task (Section 3.5.3). Applications do not explicitly choose a set of protocols to communicate. Instead, they choose a communication type. The SeLMS is responsible for selecting the protocols that realize that communication prototype. A new protocol that offers the same communication type can easily be added to the protocol stack and used by the session layer without affecting the application.

General applicability is obtained by the nature of the solution. A SeLMS exists in the session layer of the protocol stack, separated from network and transport protocols. The SeLMS does not assume the availability of particular transport or network layer protocols. It realizes its own Session negotiation protocol (Section 3.5.5) independently of the available transport or network protocols.

Secondly, **application involvement** is realized by the Application feedback task (Section 3.5.8). **Disconnected operation** is possible by suspending the session (Session state management task, Section 3.5.4) and optionally notifying the application (Application feedback task, Section 3.5.8).

Thirdly, **security** is obtained by a secure Session negotiation protocol (Section 3.5.5).

Fourthly, SeLMS are **open network-friendly**. SeLMS enable the dynamic addition and replacement of protocols in the protocol stack because they does not depend on particular transport or network layer protocols (Transport protocol management task, Section 3.5.6). SeLMSs are realized in the session layer and should therefore be part of the operating system's protocol stack, making them easily deployable without side effects. SeLMS are end-to-end solutions which do not need specialized network infrastructure.

3.7 The Connection Abstraction System and Address Management System

The following chapters describe a Session Layer Mobility Solution that adheres to the proposed architecture. The solution consists of two large parts, depicted in Figure 3.5: a Connection Abstraction System and an Address Management System. The application can interact with these systems by using the session socket.

The Connection Abstraction System (CAS) is the actual session layer which resides in the protocol stack. The CAS realizes all tasks from Section 3.5 except

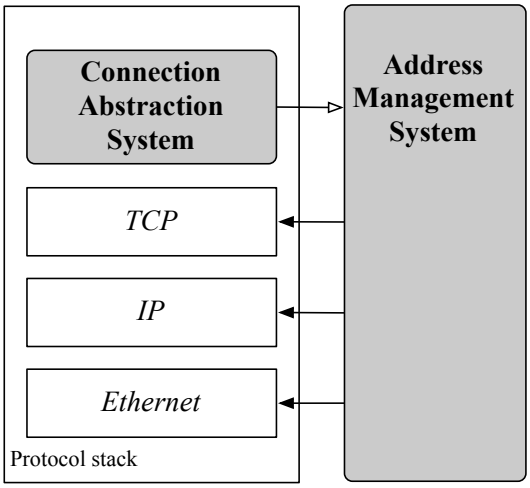


Figure 3.5: the CAS and AMS with respect to the system's protocol stack

the Session support detection task and the Protocol and address hiding task. The Session support detection task is currently not realized. The Protocol and address hiding task is realized by the Address Management System (AMS) is an optional plug-in responsible for protocol and address independence. It facilitates address and protocol independence for the application layer and therefore allows protocol changes to happen during the lifetime of a session. The CAS can work without the AMS if protocol changes are not required. When the systems are both deployed in the protocol stack, the address and protocol challenge, the aware application and the security challenge are addressed.

To facilitate the dynamic addition, upgrading and removal of protocols (Section 2.1.4) we rely on the DiPS+/CuPS framework [Mat99, Mic03, JMMV02, MJD+05]. The DiPS+ framework offers supporting services like among others protocol (re-)composition, hot plugging of network protocols and protocol downloading from a protocol component repository. These are exactly the supporting services that offer the flexibility that is required for a dynamic network environment.

Chapter 4

The Connection Abstraction System

The Connection Abstraction System (CAS) [MVJ05, MMV04, MMV04, MJV03] is a session management system that adheres to the developed architecture. It offers a session concept to the application that reflects the current network status in a dynamic network environment. The CAS is realized independently of the features of a particular transport or network protocol, making it applicable for any protocol stack instance.

The CAS is designed as a protocol stack layer, located above the transport layer. It implements a session protocol and uses the services of the lower layer transport protocols to send session management and application data to the peer. It communicates with the upper application layer by means of a session socket.

This chapter describes the design of the CAS in the protocol stack. Section 4.1 gives a detailed definition of a CAS session. Section 4.2 shows how the design of the CAS follows the design of the OSI reference model. The CAS header that is used by the protocol to exchange control data is described in Section 4.3. Section 4.4 describes the CAS protocols for session establishment, sessions suspension, session resumption and session termination. Section 4.5 specifies how the CAS interoperates with the transport layer. Section 4.6 describes how the CAS can provide network status feedback to the application. Finally, security considerations are discussed in Section 4.7.

4.1 CAS session definition

The CAS offers session services to the application. An application can establish a session with a remote application and send and receive data during the lifetime

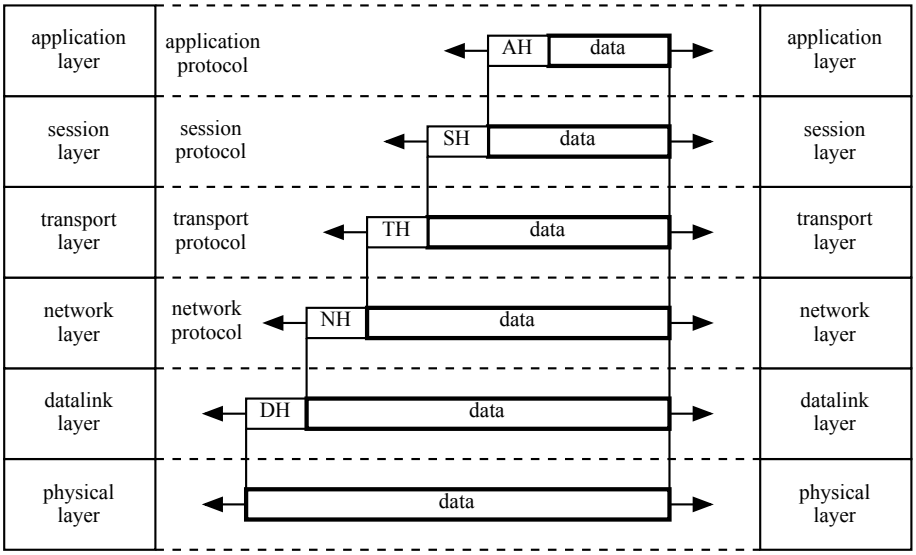


Figure 4.1: The CAS is realized in the session layer of the OSI reference model [Zim80]

of a session similar to the service offered by transport layer protocols. The added value of a session is that communication can be suspended and resumed during its lifetime. Session suspension and resumption can be requested by the application or can occur when the network access is interrupted. The former type of suspension is called *anticipated* suspension, the latter is called *unanticipated* suspension. In the case of unanticipated suspension, the application can be informed so it can adapt its behavior.

4.2 Designing a session layer in the protocol stack

To add session layer functionality, the CAS is realized in the OSI reference model's session layer [Zim80] (see Figure 4.1), located on top of the transport layer and right below the application layer. Note that the presentation layer is omitted from the Figure. Most popular stacks, such as the TCP/IP protocol suite, currently do not contain a session and a presentation layer because the functionality offered by these layers has not been needed up till now. A presentation layer is still not required; the CAS offers its session layer services immediately to the application layer. It offers support for disconnected operation and provides network status feedback to the application if required. The CAS uses the services of the lower

```
public class SessionSocket {

    public SessionSocket(int type)
        throws SessionSocketException;

    public void listen(InetAddress source, int sport)
        throws SessionSocketException;
    public SessionSocket accept()
        throws SessionSocketException;
    public void bind(InetAddress source, int sport)
        throws SessionSocketException;
    public void connect(InetAddress destination,
                        int dport)
        throws SessionSocketException;

    public void send (DataPacket dp)
        throws SessionSocketException;
    public DataPacket receive()
        throws SessionSocketException
    public OutputStream getOutputStream()
        throws SessionSocketException;
    public InputStream getInputStream()
        throws SessionSocketException;

    public void suspend() throws SessionSocketException;
    public void resume() throws SessionSocketException;
}
```

Listing 4.1: A Java Session Socket API.

layer: it uses transport layer protocols to exchange both control and application data with the peer.

Control and application data are exchanged between peers by means of packets. The packets consist of a session header (SH in the figure) that contains the session control data, followed by a payload that contains the data transmitted by the higher application layer. The CAS hence does not establish a separate control and data channel.

The following sections discuss the interaction of the CAS with the application layer and transport layer. The CAS header is described in Section 4.3.

4.2.1 Interaction of CAS with the application layer

Applications that wish to use CAS sessions must use a session socket. At a particular moment, a session socket represents one CAS session. An example socket API in the Java programming language is shown in Listing 4.1. The supported method calls on that socket are an extension of the calls that can be made on a socket of a connection-oriented transport protocol, like TCP. The `listen()`, `accept()`, `bind()` and `connect()` calls are used to configure the socket and establish new sessions. The `send()`, `receive()`, `getOutputStream()` and `getInputStream()` are used to send and receive data. The main difference with connection-oriented transport protocol sockets is in the last two calls: the `suspend()` and `resume()` calls allow an application to suspend and resume a session respectively.

The CAS differentiates between listen sessions and client sessions. Listen sessions are typically created by servers. They only have an administrative purpose and can not be used to transmit data. They represent an access point in the network that clients must contact if they wish to use the server's services. Listen sessions are created by calling the `listen()` method on the socket. The server then waits for new incoming sessions by calling `accept()`. This method returns only when there is an incoming session request.

A client that wishes to contact the service creates a client connection by creating a new session socket and issuing a `connect()` call. The call returns if the session has been successfully established. On the server side, the `accept()` call returns a new socket, which represents a client session that the server can use to exchange data and control the suspension status of the session.

4.2.2 The relation between the CAS and the transport layer

CAS sessions use transport protocols to exchange application data and session management information. Transport protocols typically offer stream based and datagram transport services to the application. The application can choose the preferred service. The CAS offers identical communication services. The `send()` and `receive()` calls in Listing 4.1 must be used if the application uses a datagram service. If the application uses a data stream service, the `getInputStream()` and `getOutputStream()` methods respectively return Java input and output streams to the application. The application can choose the type of the session socket at socket creation time by passing a type to the socket constructor. Note that the application can only determine the service type and not the protocols that will realize that service. This is important to facilitate protocol changes.

CAS client sessions are always associated with at most one transport layer connection[1]. Client sessions never use multiple transport protocol connections

[1]Transport protocol connections denote communication channels that are realized both with connection-oriented protocols (like TCP) and datagram transport protocols (like UDP). Even though datagram protocols do not require the actual establishment of a connection, they are still

simultaneously to exchange application and session control data. Application data and session control data are multiplexed on the same connection. If the session is in a suspended state, it is not associated with a transport protocol connection.

Client sessions are identified with universally unique identifiers (UUID) [LS98]. It is not recommended to use identifiers that transport, network or possibly data link layer protocols use to identify network endpoints, because they can change for mobile nodes. UUIDs are used instead of a simple numbering scheme to prevent that hosts resume a suspended session to the wrong host which accidently uses the same number to identify another session. For example, suppose a node A is communicating with a node B. At a certain point in time, node B suspends all sessions and moves to another network. Shortly thereafter, node A moves away from its network too. Subsequently, node C enters the network and assumes A's old network layer address. When B reconnects to the network, it tries to resume its suspended sessions to A's old network layer address. If C happens to have a session with the same session number, the session will be erroneously resumed. UUIDs prevent this kind of situations.

Listen sessions are identified with the traditional transport, network and data-link identifiers. On a TCP/IP stack, these identifiers are a transport protocol (TCP or UDP) port and an IP address. When a client wishes to establish a new session, it must know the identifiers of the listen session. Currently, the identifiers of a server are typically obtained by contacting a name service. It must be noted that if a server becomes mobile, its listen socket is suddenly reachable on another network layer address. If clients wish to establish new connections they must obtain the server's new network layer address. In dynamic networks, the mobile server will have to update the network layer address with every change.

4.3 The CAS header format

The session header that is used by the CAS is shown in figure 4.2. This header consists of a standard header and one option header which is only needed if a session is resumed.

The first field in the header is the `Marker` field which is used to locate the header in a data stream. If this marker also occurs in the application data, the marker is escaped by repeating the marker twice. The `Length` field indicates the length of the CAS segment. The length of a CAS segment is determined as the sum of the header length in number of bytes and the amount of higher layer data bytes appended to the header. Note that the value of the `Marker` field must be a value that cannot occur in the `Length` field, otherwise it will be interpreted as escaped application data. Because the minimum value in the header field is the length of the CAS header, a marker value that is smaller than the header length

often used to exchange data between only two peers.

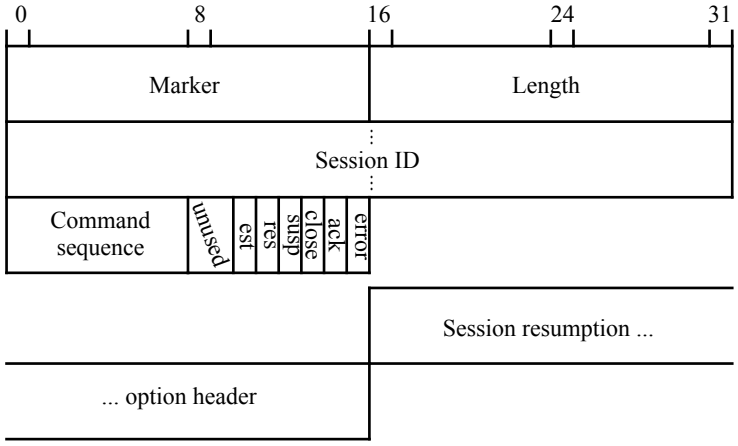


Figure 4.2: The CAS header format

can be used. The reason for having both a `Length` and `Marker` field are mainly performance reasons. The `Length` field allows for more performant processing because it avoids to scan the data stream for the marker field. However, if a data stream is interrupted and the peer starts resending data from an arbitrary point in the stream, the CAS must be able to search for the next header.

The next field in the CAS header is the `Session ID`. This field contains the session UUID and is used by the CAS to identify for what session the header and accompanying application data are intended. The session ID is followed by the `Command sequence` field. This field is used to give a sequence number to each session management request, in case they arrive out of order. For example, if a suspension request is immediately followed by a resumption request, but the resumption request arrives before the suspension request, the session will be erroneously suspended. If session management requests are sequentially numbered, the suspension request will be ignored because it will have an older sequence number than the resumption request.

The `Command sequence` field is followed by the flags that represent session control statements. The flags are realized as bits in a single byte field of the session header. There are only six flags needed, so two bytes remain unused. The `est` flag is used when a new session is to be established. In that case the `Session ID` field contains the identifier of a new session. The `res` flag is used to resume a session. The `susp` flag is used to notify the peer CAS of an anticipated suspension request. The `close` flag is used to terminate a session. The `ack` flag is used to acknowledge the four requests and must be set both with the request flag

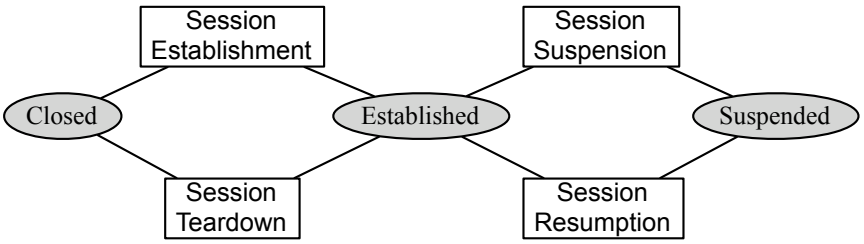


Figure 4.3: Simplified transition diagram of the CAS protocol. The square boxes represent the four CAS protocol actions. The grey ovals represent the states that the detailed state transition diagrams of the connected protocol actions have in common.

it is acknowledging. The `error` flag is used to indicate an error, for instance, a session resumption request failure. The error flag must be activated together with the other bits used in the erroneous request.

As indicated by its name, the session resumption option header is only used when a session resumption request is issued. This header field is only present when the `res` flag is set. The field contains the amount of bytes that have been successfully received by the CAS that sends this option header and is meant for synchronization purposes during session resumption. More details about session resumption can be found in Section 4.4.3.

4.4 Description of the session protocol

Two CAS-enabled systems adhere to a protocol when exchanging session control messages. This CAS protocol is steered by a state transition diagram that maintains the current state of a particular session. A session's state may change if particular network related events happen or session control messages are received from the peer. A state change is typically accompanied by the execution of a number of actions, such as sending control data to the peer CAS system or interacting with the higher application layer.

The CAS protocol consists of four protocol actions: session establishment, session suspension, session resumption and session teardown. The following four sections describe each protocol action in greater detail, with attention for the protocol action's state diagram, the state transitions and the exchanged session control messages. The four state transition diagrams in those sections can be combined into one large state transition diagram. Figure 4.3 shows what states the state transition diagrams of the different protocol actions have in common. The square boxes represent the CAS protocol actions, the grey ovals are states.

A line between a protocol action and a state indicates that the state is present in the protocol action's state transition diagram. The states are the join points when combining the four partial state transition diagrams into the complete protocol state transition diagram.

4.4.1 Session establishment

Session establishment is broken down into two parts: establishing a communication channel and exchanging session parameters. To obtain a communication channel the CAS uses the services of the transport layer. If the transport connection has been successfully established, it can be used to exchange the session parameters. The main goal of this exchange is to agree on a unique session identifier that can be used for future suspension and resumption requests of that particular session. Because the CAS uses UUIDs as session identifiers, the agreement can be reduced to a simple exchange of session identifiers.

Because the CAS is designed as a layer in the protocol stack, transport connection management does not happen by means of application layer sockets. A transport connection is not established by creating a transport protocol socket and calling its `connect()` operation. Data is not send by calling the write operation on a socket. When the CAS wants to send data using a particular transport connection, it encapsulates this data in a packet and gives it to the lower protocol stack layer for further processing. The details are explained in Section 4.5).

The following sections discuss the possible states of a session during session establishment and the control messages that are exchanged to negotiate the session identifier.

4.4.1.1 Transition diagram for session establishment

The transition diagram for session establishment is shown in more detail in Figure 4.4. The figure is split up in two parts. These two parts show the states that apply to the two communicating parties. We will refer to the client as the party that actively initiates the session. The server is the party that passively initiates the session. The left part of the figure depicts the states of the client when establishing a session, the right part shows the states of the server side.

Before a session can be established, the server must first create a listening session. A listening session indicates that the server is willing to accept new sessions. This is similar to traditional protocol stacks where a server must first create a listen socket to be able to accept new transport connections. When a server creates a listen session, the CAS prepares the lower protocol stack layers to accept new transport connections that will be used to exchange data for new client sessions. The lower layer transport protocol is instructed to start listening on a given transport protocol port. When the CAS sends the request to the transport layer, the session is moved to the `ListenRequesting` state. The session stays in

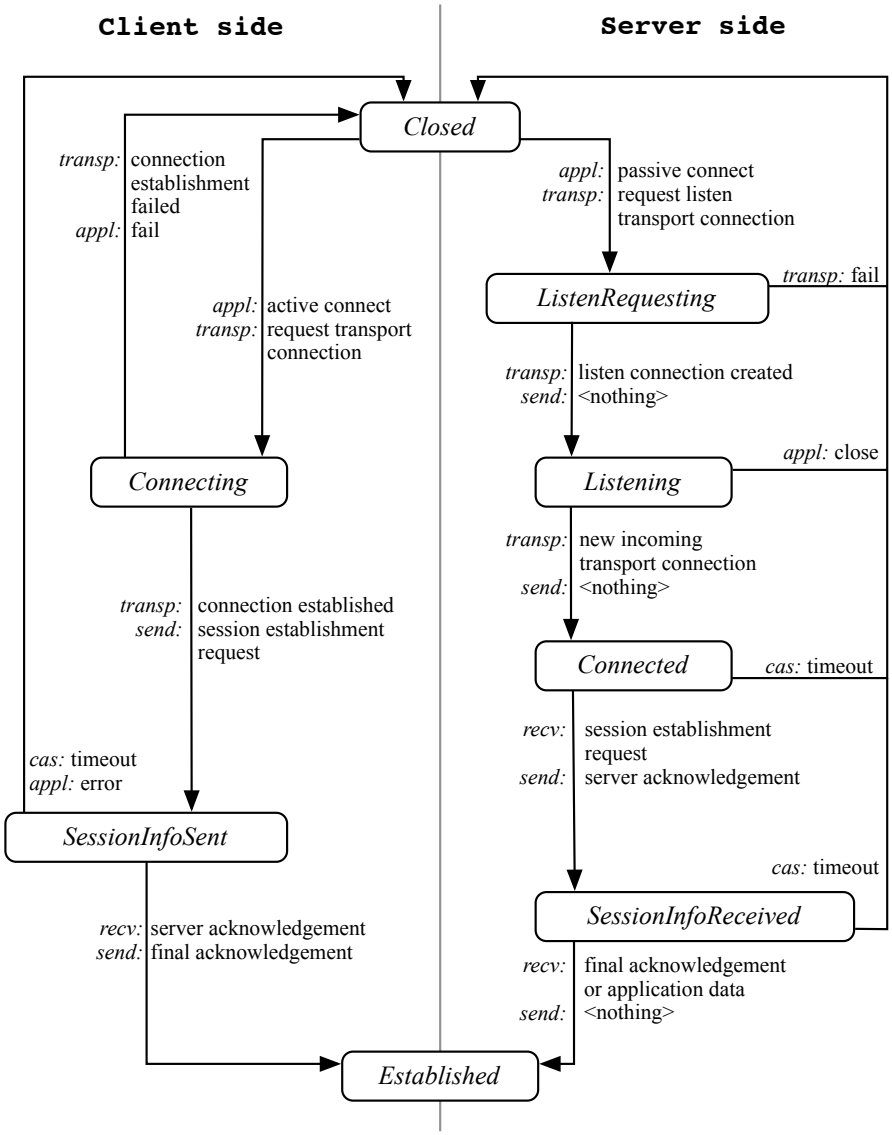


Figure 4.4: Session establishment transition diagram

this state until the transport layer confirms that the request has been executed successfully. If the request is successful, the listening session is moved to the `Listening` state. From that moment, the CAS is ready to receive new sessions. If the request fails, for example because the port is already being used, the listening session moves back to the `Closed` state, and the application is informed.

When a client wishes to communicate with a server, it instructs the CAS to create a new session. The client CAS uses the lower layer transport protocols to create a transport connection to the network coordinates on which the service can be reached. In the case of TCP/IP, these coordinates are represented by an IP address and a transport protocol port. While transport connection establishment proceedings, the client CAS stays in the `Connecting` state. If a transport connection cannot be established, the client CAS moves back to the `Closed` state and the application is notified of the failure. If connection establishment succeeds, the server side CAS moves from the `Listening` to the `Connected` state where it waits for the client CAS to start session negotiation.

After the transport connection has been established both parties must agree on a session identifier. The use of Universally Unique Identifiers (UUIDs) simplifies this agreement process because no clashes can occur. The client can simply create a UUID and then send it to the server.

Exchanging a UUID is done by means of a three way handshake. The UUID is exchanged using the session header that is described in Section 4.3, and is sent using the newly established transport connection. Because it is possible that the used transport protocols do not offer reliable transport, a three way handshake is necessary to ensure that both parties are in possession of the session key before data transfer starts. The remainder of the states depicted in Figure 4.4 are the states that reflect the three way handshake protocol used to exchange the UUID.

When the client CAS receives the notification from the transport layer that the transport connection has been successfully established, it calculates a UUID, sends it to the server side CAS and moves to the `SessionInfoSent` state. The client CAS waits in that state until the server side CAS acknowledges the reception of the UUID. If the client CAS does not receive this acknowledgement within a certain time, it resends the UUID. If repetitive resends do not result in an acknowledgement, the client will assume the server CAS is no longer reachable, e.g. it may have moved away from its access point, close the session and inform the application that an error has occurred. The server side CAS is waiting in the `Connected` state when it receives the UUID. Upon reception, the server CAS immediately sends an acknowledgement (ACK) back to the client CAS and moves to the `SessionInfoReceived` state. If the client CAS receives the acknowledgement, it completes the three way handshake protocol by sending a final ACK to the server side. The client moves to the `Established` state and the application can start sending data to the server. The server side CAS is also moved to the `Established` state if it receives the final ACK from the client CAS. In case that

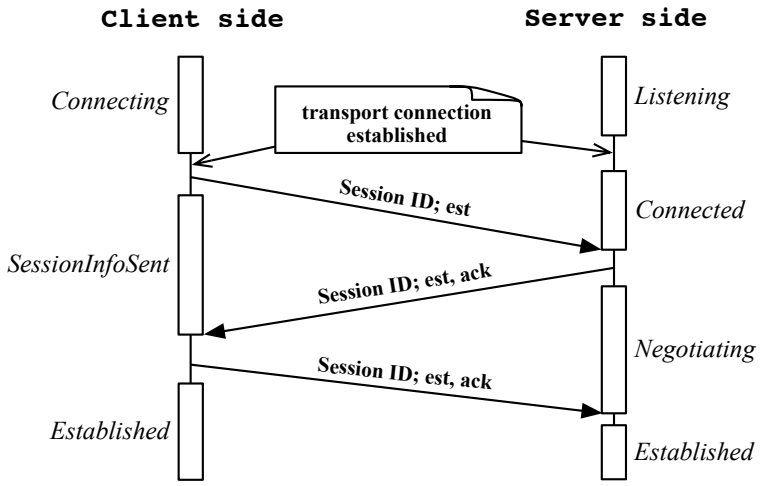


Figure 4.5: Session establishment protocol

ACK gets lost, the server also moves to the `Established` when it receives application data in the `SessionInfoReceived` state. If the server CAS does receive anything within a predefined time from the client CAS, the server will assume that the acknowledgement got lost. The server CAS will therefore resend its acknowledgement until the client CAS answers or until it decides to give up after resending a predefined number of times.

4.4.1.2 Protocol message exchange for session establishment

During session establishment, the client and server CAS system exchange protocol messages by means of the session header. The messages sent during a normal session establishment protocol run are depicted in Figure 4.5 as bold arrows pointing from the client side to the server side and vice versa. The arrows also show what information is contained in the header.

During a normal three way handshake, three messages are exchanged between both parties. The first message is sent from the client to the server. If a new session is being set up, the message must have the `est`-flag set (see Section 4.3) and must contain a new session ID. If the server receives an existing session ID, it must send a session header back with the same session ID and both the `est`-and `err`-flag set. If the server receives a session header that contains a session ID that has not been negotiated before and the `est`-flag is not set, it must send a session header back with that session ID and the `err`-flag set. After sending the

error packet the transport connection must be shut down, and the corresponding session must be destroyed.

If the server received a correct establishment header, the second message in the protocol is a reply from the server to the client. This reply header contains the session ID that was received by the client and has both the `est`- and `ack`-flags set. If a client receives a session header in reply that deviates from the expected header, an error header is sent back. This error header is a copy of the received header with the `err`-flag set. After sending the error header, the transport connection is closed.

The third message is the last reply in the three way handshake, sent from the client to the server. This message is the same message that was received from the server: it contains the session ID and has the `est` and `ack`-flag set. This header completes the three way handshake protocol.

4.4.2 Session suspension

In a dynamic network environment, a node may decide to stop communicating for a while, for example, to save power or may be disconnected from the network because it moves too far away from the wireless access point it is using. In both cases the node's CAS sessions are suspended. In the former case, suspending communication is a decision that can be controlled by the node or the user. This anticipated suspension (see Section 4.1) is negotiated between the two communicating parties. The latter case, is something out of control of the node or and is therefore classified under unanticipated suspension.

This section describes how a session becomes suspended, both in the anticipated and the unanticipated case. First, the transition diagram used to guide a session's suspension process is explained. Secondly, the protocol messages that two CAS systems exchange during suspension are studied in greater detail. Thirdly, the additional measures taken by the CAS if a session uses reliable transport protocols are discussed.

4.4.2.1 Transition diagram for session suspension

Figure 4.6 shows the transition diagram that is used by the CAS during suspension. The full arrows depict normal state transitions that occur during suspension negotiation. The dashed arrows show transitions during an unanticipated suspension. Dashed arrows not only drawn from the `Established` state to the `Suspended` state, but also from any other session suspension state because unanticipated suspension can also occur during suspension negotiation.

We refer again to the client as the party that is the cause of the suspension of the session. In the case of unanticipated suspension, the client is the mobile node that moves away from the access point. In the case of anticipated suspension the client is the party that initiates the suspension request. The server is the party

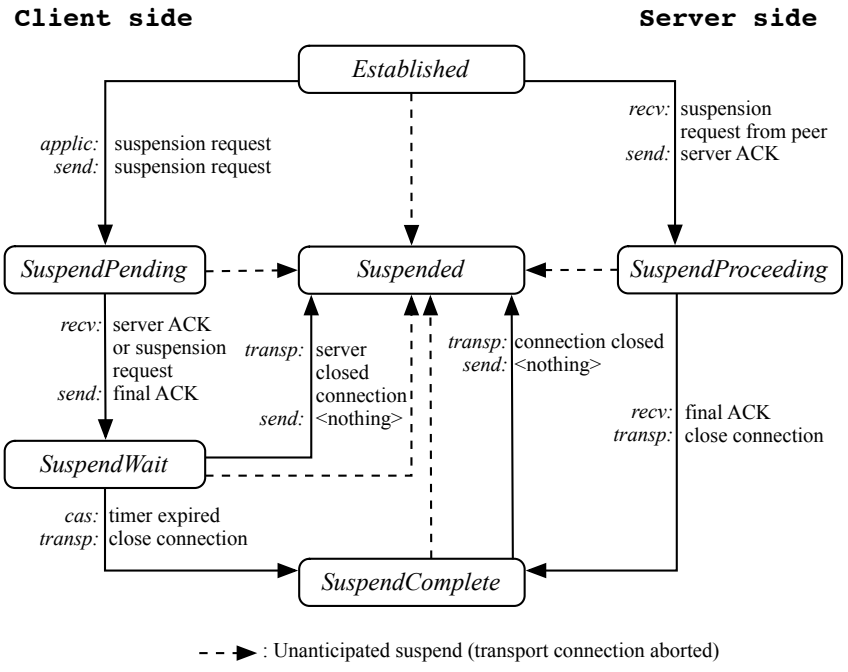


Figure 4.6: Session suspension transition diagram

that receives a suspension request. First, the transitions for anticipated suspension are discussed. Secondly, the measures that the CAS takes in case of unanticipated suspension are described.

Suspending a session in an anticipated way is also realized as a three-way handshake protocol. The used transport protocol may not realize a reliable communication channel. During normal communication, both the client and the server are in the `Established` state. If a client application or the operating system decides to suspend a session, the client CAS sends all remaining data to the server, followed by a suspension request for that session. The client session is subsequently moved to the `SuspendPending` state. The server side CAS is still in the `Established` state when it receives the client's suspension request. If the server CAS receives a suspension request, it acknowledges this request to the client CAS (this is called the server acknowledgement) and moves to the `SuspendProceeding` state. The server CAS notifies the server application if desired, and blocks all server application requests to send or receive data. If the client CAS does not receive an acknowledgement from the server within a certain amount of time, it resends the

suspension request. If after repetitive resends the client still has not received an acknowledgement, the CAS suspends the session unanticipatedly. If the client received the server acknowledgement, it sends the final acknowledgement to the peer and moves to the `SuspendWait` state. If the server CAS does not receive the final acknowledgement, it resends the server acknowledgement. If the server CAS did not receive the final acknowledgement after a number of resends, the server CAS suspends unanticipatedly.

The `SuspendWait` is similar to TCP's `TIME_WAIT` state. A TCP client waits in this state until it can be fairly sure that all packets belonging to the TCP connection and still in the network have left the network. This allows for the connection's TCP ports to be safely reused. Additionally, it keeps the TCP connection alive in case the last acknowledgement must be resent. In the CAS, the `SuspendWait` state is only necessary in case the final acknowledgement is lost in the network since UUIDs will never be reused. A new UUID will be generated when a new session is requested. The client CAS will still be active in the `SuspendWait` state when it receives a duplicate server acknowledgement and can subsequently resend the final ACK. Resending the final ACK would not be possible if the client CAS immediately terminated the transport connection and suspended the session after receiving a server acknowledgement.

The time the CAS session should stay in the `SuspendWait` state depends on the session's round trip time. This is the minimum amount of time the server side must wait before it can start to assume that the server acknowledgement or final acknowledgement was lost. To account for unforeseen delays in the network the server should wait longer than the estimated round trip time. The client must wait a similar amount of time (round trip time plus an additional time margin) before it can move from the `SuspendWait` to the `SuspendComplete` state. If the client does not receive a duplicate server within that time period, it can assume that the server received the final acknowledgement and has moved from the `SuspendProceeding` to the `SuspendComplete state`.

After the waiting period, the client CAS moves to the `SuspendComplete` state. In this state and instructs the transport layer to close the transport connection. If the transport connection confirms that the transport connection is closed, the client session moves to the `Suspended` state. The server CAS moves from the `SuspendProceeding` state to the `SuspendComplete` if it receives the final ACK from the client. The server CAS then also closes the transport connection, and moves to the `Suspended` state when the connection has been terminated.

If connection-oriented transport protocols are used, it is possible to optimize the suspension scenario. The client can be moved from the `SuspendWait` state immediately to the `Suspended` state if the server closes the transport connection (see Figure 4.6). If the server CAS received the final acknowledgement, it can actively close the transport connection. Due to the nature of connection-oriented transport protocols, the client CAS will be informed that the connection is closed. For the

client CAS, this is an indication that the server CAS received the final acknowledgement. Consequently, the CAS can be moved immediately to a `Suspended` state. If the connection is not closed by the server, the client CAS waits for a certain period in the `SuspendWait` state and then closes the connection itself.

In the unlikely event that both the client and the server send a suspend request, they will both be in the `SuspendPending` state. If a suspension request arrives when a peer is in that state, it also sends an acknowledgement and moves to the `SuspendWait` state. Hence, both peers acknowledge each other's request, and wait for a possible duplicate suspension request, which can occur when the acknowledgement was lost. After waiting, both peers close the connection and go to the `Suspend` state.

The CAS executes an unanticipated suspension if the network access connection is lost during communication, or if the transport connection is aborted, or if timeouts happen during the session protocol's suspend negotiations. The CAS consequently moves the session (or sessions) immediately in the `Suspended` state and aborts the transport connections if necessary. If the used transport protocols offer a reliable data stream, the CAS takes additional measures to ensure the data stream reliability between consecutive transport connections (see Section 4.4.2.3). A special case of unanticipated suspension happens if the server CAS receives a session resumption request for a session that is in the `Established` state. In that case the CAS immediately suspends that session unanticipatedly and subsequently starts the session resumption procedure.

4.4.2.2 Protocol message exchange for session suspension

Figure 4.7 visualizes the relation between client and server states and the protocol messages (CAS headers) that are exchanged during a normal session suspension, i.e. a session suspension that is not interrupted by network or transport protocol errors. The figure must be read from top to bottom and depicts how the states change over time as reaction to protocol messages that are shown as arrows.

Both parties are in the `Established` state when the client initiates session suspension. The client sends a session header with the session identifier and the `susp` flag set after transmitting all remaining buffered application data. [2]. This session header is sent using the same transport connection as for exchanging application data. Subsequently, the client session is moved to the `SuspendPending` where it waits for an acknowledgement.

If the server CAS receives the suspension request, it sends the server acknowledgement header back to the client and moves to the `SuspendProceeding` state. This reply header contains the session ID and has both the `susp` and `ack` flag set. If the client receives this header it responds with the final acknowledgement header, which also contains the session ID and has the `susp` and `ack` flag set. The

[2]The suspend flag can be set on the last data packet that is sent by the client party.

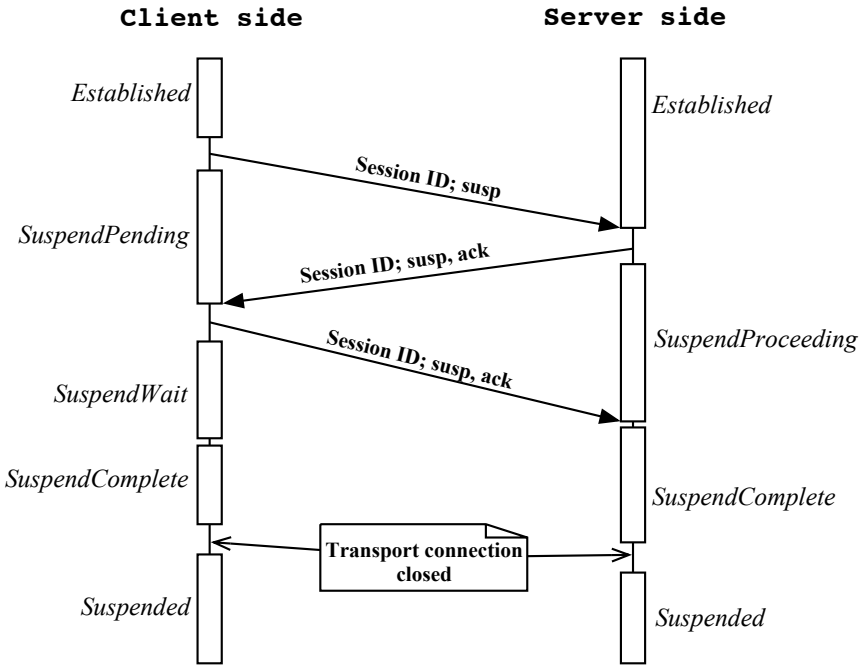


Figure 4.7: Session suspension protocol

client moves to the `SuspendWait` state where it waits for possible duplicate server acknowledgements. After waiting for an amount of time, the client moves to the `SuspendComplete` state and instructs the transport layer to close the transport connection, after which it moves to the `Suspended` state. The server moves from the `SuspendProceeding` state to the `SuspendComplete` state as soon as it received the final acknowledgement. It also closes the transport connection and moves to the `Suspended` state.

If the client or server receives a header that does not apply to the correct session, a copy of the header is sent back and, additionally, the `err` flag is set. If a client or server receives an error packet, it remains in the same state and resends the last transmitted session request or acknowledgement. In case the client or the server is in the `Established` state, the faulty header is simply discarded.

4.4.2.3 Using reliable transport protocols

Applications can request a reliable data stream service from the system's protocol stack. In that case, the goal of the CAS is to ensure this reliability during

the entire lifetime of the session, also when session suspensions occur. Transport protocols that offer reliable communication services in current protocol stack implementations typically only offer reliability in the lifetime of a single transport connection. The CAS on the other hand offers a session to the application that can span multiple transport connections. A session's transport connection is closed normally in case of anticipated suspension. A session is suspended unanticipatedly if the transport connection is aborted. When resuming the session a new transport connection is established and communication can continue (See Section 4.4.3). If the CAS can suspend anticipatedly, it has complete control over what data has been transmitted successfully. In case of unanticipated suspension, there are no guarantees what data was successfully sent and received by the peer party because reliable transport protocols internally buffer transport connection state. The CAS must therefore take additional measures to ensure reliability, which are explained in the following paragraphs. It must also be noted, if unreliable transport services are requested by the application, the CAS must not provide these additional measures. In that case anticipated and unanticipated suspension are the same, apart from the difficulties to detect unanticipated suspension on the server side.

There are two possible solutions to ensure reliability when handling unanticipated suspension. The first solution is *double buffering*: all data that is sent is also buffered in the CAS until it can be assured that the peer CAS has received it. The amount of data that must be buffered is the sum of the amount of data that can be buffered in the local transport protocol before it is sent and the amount of data that can be buffered in the receiving transport protocol before it is read by the peer CAS. In the case of TCP the amount of data to be buffered is the sum of the size of the local send window and the size of the remote receive window. Next to the buffers, both CASs register how many bytes they have sent and received. This is necessary to synchronize after an unanticipated suspend. The synchronization process will be explained in detail in Section 4.4.3.

The second solution realizes reliability by allowing the CAS to extract and insert transport connection state from and in the transport layer protocol. If a transport connection is aborted it is not immediately cleaned up. Instead the CAS extracts the content of the send and receive buffers of the aborted connection from the transport protocol. The CAS then suspends the session and the connection can be cleaned up by the transport layer protocol. When the session is resumed, the send and receive buffers are restored during connection establishment by importing the previously exported connection state. The transport protocol can then resume communication with the same buffer state as at the time of suspension.

The choice of the optimal solution is steered by the trade off between general applicability and performance. Buffering data twice is not very performant, but is independent of a particular transport protocol and therefore generally applicable. Extracting and inserting state is transport protocol dependent, and may not be trivial or even be possible. On the other hand it is more performant because data

is not buffered twice in the protocol stack.

Finally, it must be noted that ensuring reliability is also required during anticipated suspension negotiation because network access can also be interrupted during suspension negotiation. In the case reliable transport connections are used, the client can be sure all data has been sent and received upon arrival of the server acknowledgment. The server can only be sure that all data was sent and received if it received the final acknowledgement. If transport connections are aborted before the client and the server have respectively received the server acknowledgement or the final acknowledgement, the respective CASs execute the unanticipated suspension scenario.

Note that in the case of reliable transport protocols, the sending of the final acknowledgement could be omitted. The client could close the connection immediately after it received the server acknowledgement. As soon as the server notices that the client closed the connection, it can assume that the client side received the server acknowledgement. However, if the connection is aborted instead of closed, the server can not be sure that the client received the server acknowledgement. If the client acknowledges that it received the server acknowledgement by sending a final acknowledgement, the server can be sure that session suspension was complete. After receiving the final acknowledgement, an aborted connection or a connection that is closed normally does not make any difference anymore because the server received an explicit confirmation from the client that suspension negotiation is complete. Hence, the final acknowledgement is sent even if reliable transport connections are used.

4.4.3 Session resumption

This section describes how sessions are resumed after suspension. Resuming a session is independent of the suspension type ((un)anticipated). Resuming a session is similar to session establishment but requires additional bookkeeping. First, the peer communication party is located using a name service and a new transport connection is set up. If that is successful, both CAS layers must negotiate what session is to be resumed. Finally, if the application requested a reliable data stream service, the two CAS systems synchronize to prevent data loss.

The following two sections respectively discuss the states of a session during session resumption and the exchanged protocol control messages to learn the ID of the session that is to be resumed and to synchronize the data stream.

4.4.3.1 Transition diagram for session resumption

For the description of the resumption protocol, a client and server role are again considered. The client is the CAS that decides to resume the session, and actively tries to establish a new connection to the server. The server is the party that passively opens a transport connection and receives a resumption request. In the

case of anticipated suspension, the sessions explicitly suspended explicitly by a user or the system will not be resumed unless explicitly requested by the user or system. The party that suspended the session must also resume the session and is therefore the client in the session resumption protocol. In the case of unanticipated suspension, the CAS acting as client in the session suspension protocol is responsible for resuming a session as well. During session suspension, the client is the party that noticed the network disconnection because it actively moved away from its access point. Because this party noticed network disconnection it will notice reconnection as well. Consequently, the party that reconnects to the network is the obvious party to initiate session resumption and is therefore considered to be the client of the session resumption process.

In the case of transport protocol timeout it is more complex to identify the client and server in the suspension and resumption scenario. Suspension that is detected because of transport protocol timeout, may be caused by the peer party that has disconnected from the network. The peer party consequently is the client. However, it is also possible that there is a problem in the network, for example, because a router went down. In that case both CAS systems will time out. Consequently, both parties will assume the role of server in the session resumption protocol and wait for the peer to resume the session. To avoid that a session will remain suspended indefinitely in case of a network failure, a CAS that assumed the server role must periodically try to actively resume a session, i.e. assume the role of client to avoid eternal suspension.

The state transition diagram for session resumption is shown in Figure 4.8. On the left, the states that the client is in during session resumption are shown. The right side shows the states the server CAS can be in. As mentioned earlier, a transport connection must first be established between the communicating parties. The client CAS tries to establish a new transport connection to the server, which is handled similarly as in the session establishment scenario. The client CAS will first instruct the transport layer to establish a transport connection. After that, the CAS moves to the `Reconnecting` state. If establishing the transport connection fails, the session is moved back into the `Suspended` state. If the transport connection is successfully established, the client CAS immediately sends a resumption request for the session in question. The session is subsequently moved to the `Reconnected` state where it waits to receive a response from the server.

When the server receives an incoming transport connection, it is not clear if the incoming connection belongs to a new session or a existing session. Therefore, the server always assumes the establishment of a new session, and the session establishment protocol is followed until the server is in the `Connected` state (see Figure 4.4). If the server CAS receives a session establishment request in the `Connected` state, the remainder of the establishment scenario is executed. However, if the server CAS receives a session resumption request, it verifies the existence of the received session ID. If the session does not exist, an error header is sent back

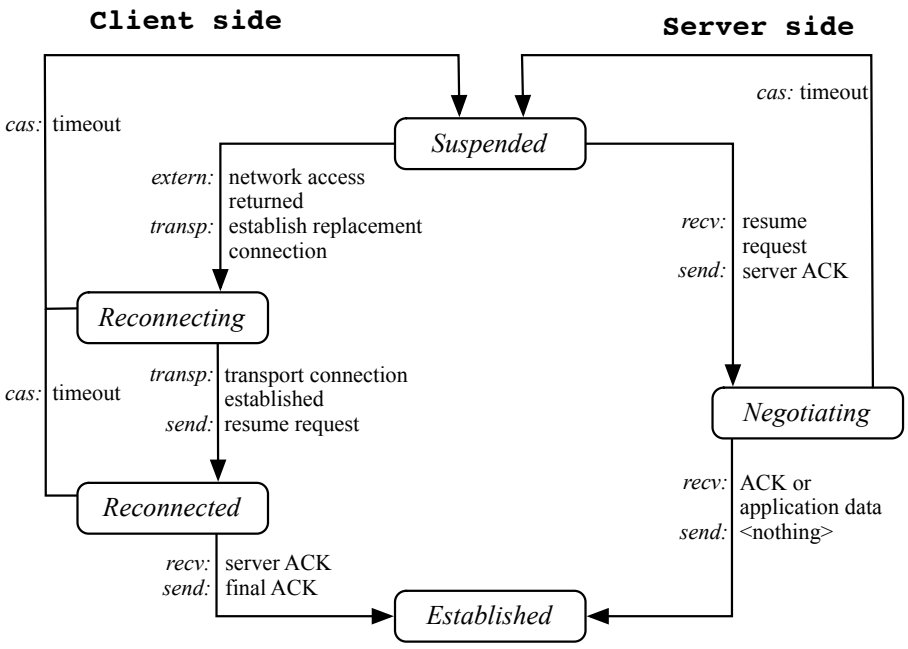


Figure 4.8: Session resumption transition diagram

to the peer because it is trying to resume a non-existing session. If the session exists, the session establishment scenario is abandoned and the transport connection is associated to the existing session where it is used to complete the session resumption protocol. The server CAS sends a server acknowledgement for the resumption request and moves the session to the `Negotiating` state. Note that it is possible that the server receives a resumption request for a session that is in the `Established` state. This can happen if the server did not detect the client disconnection. In that case the server immediately aborts the old transport connection, suspends the session unanticipatedly, acknowledges the resumption request on the new transport connection and moves to the `Negotiating` state.

If the client receives the server acknowledgement, the client CAS sends the final acknowledgement to the server CAS and moves to the `Established` state. If it does not receive the server acknowledgement within a certain timeout period, the client CAS resends the resumption request using the transport connection. If a number of resends still do not result in the reception of a server acknowledgement, the session is moved back to the `Suspended` state. If the server CAS receives the final acknowledgement or application data, it is also moved to the `Established` state. If the server CAS does not receive a final acknowledgement in time, it resends the server acknowledgement. If after a number of resends the client CAS has not responded, the server CAS moves the session back to the `Suspended` state.

4.4.3.2 Protocol message exchange for session resumption

Figure 4.9 displays the protocol messages that are exchanged during a session resumption scenario. The figure shows the state transitions as a function of time. Time passes from top to bottom. The left side shows the client states, the right side the server states and arrows going from left to right and vice versa show the exchanged protocol messages. The figure shows that the client first tries to establish a transport connection because network access returns. As soon as the transport connection is established, the client CAS sends a session header with the ID of the session that must be resumed and the `res` flag set and moves the session to the `Reconnected` state. If the server CAS receives this request, it checks the ID. If the server recognizes the ID, the corresponding session is moved to the `Negotiating` state after sending a server acknowledgement header back. This header also contains the session ID and has the `res` and `ack` flags set. If the client CAS receives this header, it sends the final acknowledgement back to the server. This acknowledgement header also contains the ID of the session that must be resumed and has the `res` and `ack` flags set. The client is moved to the `Established` state. If the server receives the final acknowledgement or if it receives application data, it also moves to the `Established` state.

In case of errors, an error header, which is a copy of the received header and has the `err` flag set, is sent back. If a resumption request is issued to a server with no knowledge about the received session ID, the error packet is sent back,

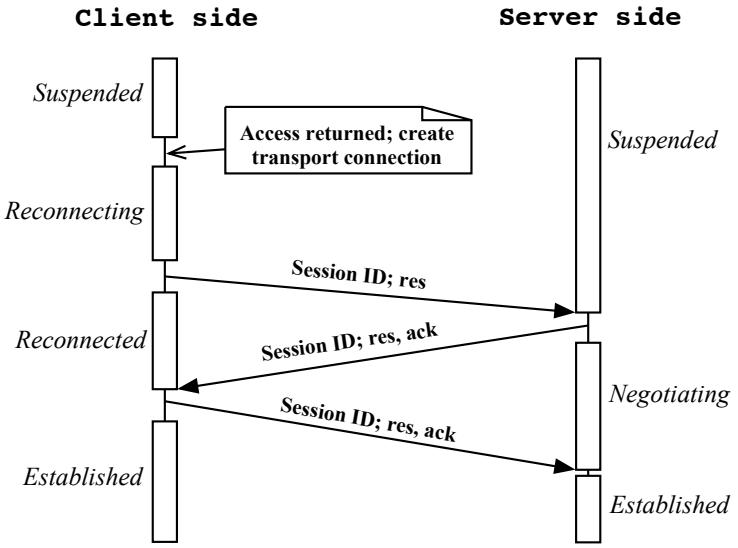


Figure 4.9: Session resumption protocol

the transport connection is terminated and state that was reserved for the session is cleaned up by the server CAS. The client CAS does the same upon reception of the error and informs the application. If the session exists on the server side, but a packet is received that does not comply to the protocol specification, an error is sent back, the transport connection is torn down and the session is moved back to the `Suspended` state.

4.4.3.3 Using reliable transport protocols

If the application requested reliable data transport services, the two CASs must first resynchronize to ensure that no data is lost. Resynchronization is performed according to the selected solution (double buffering or transport protocol state handling, see Section 4.4.2.3).

In the case of double buffering, a "received bytes" option header is sent along with the session resumption header (see Section 4.3). This option header contains the amount of bytes that have been successfully received by the CAS sending the header. Because the CAS buffers an amount of data that is equal to at least the sum of the size of the local transport protocol's send buffer and the size of the peer's receive buffer, the data that was lost in the transport protocol during connection abortion will still be in the CAS's buffer. If the CAS receives the amount of bytes successfully received by the peer CAS, it can deduce what

data must be resent with the new transport connection. During the resumption protocol, the client CAS sends the "received bytes" option header with the initial resumption request. The server CAS sends the option header along with the server acknowledgement. The client CAS can start sending data as soon as the server acknowledgement is received. The server CAS can start sending data as soon as the final acknowledgement is received.

If transport protocol state importing and exporting is used, the content of transport connection's send and receive buffers, together with the required protocol state must be reinstated during session resumption. This reinstatement process is broken down into three steps: the first step prepares the connection for session resumption, the second part is the actual CAS resumption protocol and the third part consists of reinstating the send buffers. These three steps are discussed in detail in the next paragraph.

The first step, preparing the connection for session resumption, consists of emptying the receive buffers of the connection. After the old connection was aborted the receive buffers possibly still contained application data. To prevent that new application data will arrive in the connection's buffer before the old data and accompanying state (for example sequence numbers and acknowledgement state) is reinstated into the new connection, the receive buffer state should be reinstated first. The CAS then must empty the receive buffers to prepare the connection for receiving the CAS protocol headers of the resumption protocol. Ideally this is done immediately after the connection resources are reserved by the transport protocol and before the connection is established. This is only possible for the client. Since the server is not aware that the session is being resumed until the session resumption request was received. To prevent that new data arrives before the old data is imported back into the connection, the server should import the receive buffer state before the server acknowledgement is sent. The client will not send new data after it received the server acknowledgement.

In the second step, the resumption protocol is executed normally. Note that the send buffers are not yet reinstated because the transport connection must first send the CAS protocol headers needed for session resumption.

After the session resumption is completed, the send buffers can be reinstated. On the client side, buffers can be reinstated after the server acknowledgement has been received. On the server side this happens when the final acknowledgement or the first application application data was received (when the final acknowledgement was lost).

4.4.4 Session termination

When the data exchange between two communicating peers is finished, a session must be closed. The protocol for closing a CAS session resembles the TCP closing protocol. The main difference with TCP is that during session closing, the session can still be suspended.

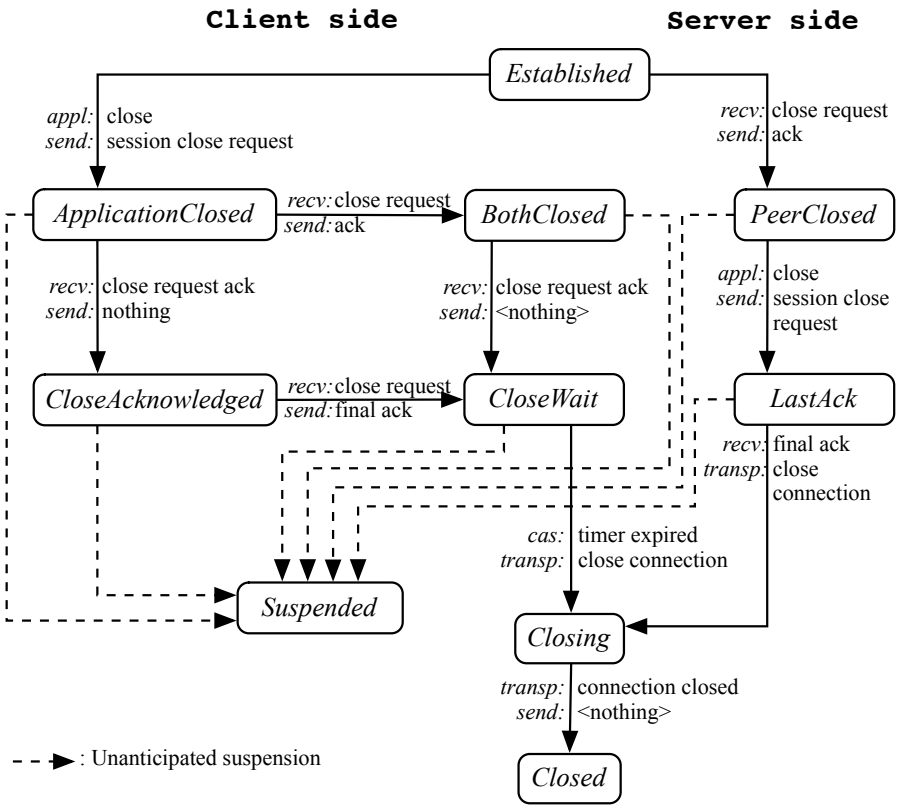


Figure 4.10: Session closing transition diagram

4.4.4.1 Transition diagram for session termination

Figure 4.10 shows the transition diagram used by the CAS to close a session. The peer that first closes the session is referred to as the client in the protocol, the other peer is the server. The left hand side of the figure shows the states for the client, the right hand side shows the states for the server. Both client and server start in the `Established` state. A session is closed when the application closes the session socket. The client CAS then sends a session header in which it announces to the peer that it wants to close the session. The client then moves to the `ApplicationClosed` state where it waits until it receives an acknowledgement from the server.

When the server receives the closing request of the client it sends an acknowl-

edgement to the client and moves to the `PeerClosed` state. If the client receives this acknowledgement, it moves to the `CloseAcknowledged` state, where it waits for the peer to also close the session. If the client does not receive the acknowledgement, it resends the close request. The session is suspended unanticipatedly when an acknowledgement is still not received after repetitive resends.

When the server is ready to close the session, it sends a session close request to the client and moves to the `LastAck` state where it waits for the client's acknowledgement. If the acknowledgement is not received, the request is resent. If the acknowledgement is still not received after repetitive resends, the session is suspended. If the client receives the session close request from the server, it sends the acknowledgement and moves to the `CloseWait` state. This state is similar to the `SuspendWait` state used during session suspension (see Figure 4.6). The client remains in this state for an amount of time in order to allow the server to send duplicate close requests in case of a lost acknowledgement. The discussion on how long the client must stay in this state is similar to the discussion that the client must wait in the `SuspendWait` state (see Section 4.4.2.1). This wait time should be longer than the session's estimated round trip time.

As soon as the server receives the acknowledgement of the client, the server closes the connection and moves to the `Closing` state. The client moves to the `Closing` state when it has not received a duplicate session close request from the server in the determined time period. When the transport layer has closed the connection, the client and server move to the `Closed` state. If connection-oriented transport protocols are used, the client may notice that the connection was closed by the server while it is still in the `CloseWait` state. In that case the client can stop waiting, close the connection immediately and move to the `Closing` state.

The CAS also supports the exceptional event that both peers close the session simultaneously. Since they both send the first close request, both peers act as client. If a peer receives a close request while being in the `ApplicationClosed` state before it received the acknowledgement to its own close request, it acknowledges that request immediately and moves to the `BothClosed` state. Both peers will move to the `CloseWait` state as soon as they receive the acknowledgement for the close request they sent. They wait there for duplicate close requests in case the acknowledgement they sent was lost and the peer resends the close request.

During session termination, a session can still be suspended unanticipatedly. For a session to be successfully closed, the session termination protocol must be completed, i.e. both peers must be in the `Closing` state. In every other case, one of the parties may still be transmitting data, or may be waiting for an acknowledgement from its peer to confirm that it received a close request. When access returns after suspension, the peers execute the session resumption protocol. Each peer that had the session socket closed by the application then immediately repeats the session termination protocol.

4.4.4.2 Protocol message exchange for session termination

Figure 4.11 shows a session termination protocol run with the CAS message headers that are exchanged. The client states are presented on the left and the server states are on the right. The top-down direction shows the evolution in time. Both peers start in the Established state. The client first sends the session close request by means of a header that has the `close` flag set and moves to the `ApplicationClosed` state. As soon as the server receives this message, it moves to the `PeerClosed` state and acknowledges the request by replying with a CAS header that has the `close` and `ack` flag set. The client moves to the `CloseAcknowledged` state when it receives the acknowledgement header.

When the application on the server has finished transmitting data and also closes the server socket, the server sends a close request. For this action the same header is used: it only carries the `close` flag. The server then moves to the `LastAck` state where it waits for the acknowledgment from the client. The client acknowledges this request by responding with a header that has the `close` and `ack` flag set. The client subsequently moves to the `CloseWait` state, where it waits for duplicate close requests to arrive. When the server receives the acknowledgement, it instructs the transport layer to close the connection. The client also closes the connection after it waited in the `CloseWait` state. When connection is closed, both client and server move to the `Closed` state.

4.5 Transport connection management

CAS interaction with the transport layer consists of two parts. First, the CAS uses the transport layer to establish communication channels with the peer CAS. This interaction is similar to the connection establishment procedure applications follow when using traditional protocol stack implementations. However, if the CAS is realized as a layer in the operating system's protocol stack, the use of sockets will be difficult because they are realized in the application layer. Inside the operating system, the functionality of the lower layers is typically used by issuing direct function calls (system calls) in that layer instead of using intermediate sockets. Nevertheless, the interaction between the CAS and the transport layer will resemble the interaction between an application and a transport protocol socket.

Secondly, if the transport protocol supports exporting and importing connection state, the CAS can use this to extract the state from an aborted transport connection, and inject that state back to a replacement connection. This process has been explained in detail in Section 4.4.2.3 and Section 4.4.3.3. In short, the CAS extracts the complete transport connection state, i.e. send and receive buffers, when the connection aborts. Injecting the state back into the replacement connection happens in three parts. First of all, the receive buffer state is reinstated and

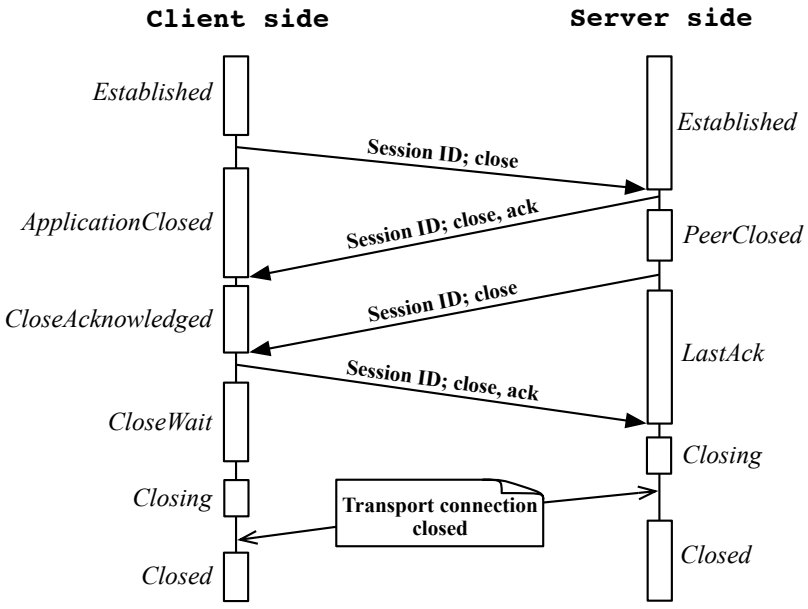


Figure 4.11: Session termination protocol

```
public interface SendBufferState;

public interface ReceiveBufferState;

public class BufferState{
    SendBufferState sbs;
    ReceiveBufferState rbs;
}

public interface CASConnection
  extends ReliableConnection {
    public BufferState export();
    public void import(SendBufferState sbs);
    public void import(ReceiveBufferState rbs);
}
```

Listing 4.2: Example API for connection state exporting and importing.

read by the CAS. Secondly, the session resumption protocol is executed. Thirdly, the send buffer state is restored. The CAS must hence be able to differentiate between send and receive buffer state.

The `CASConnection` Java interface in Listing 4.2 is an example interface the CAS can use and that transport protocols supporting connection state export and import should implement. The `export()` method returns both send and receiver buffer state. However, there are two `import()` methods, one for importing send buffer state, one for importing receive buffer state. The implementation of these methods is transport protocol dependent and is not the responsibility of the CAS. It should also be noted that the `SendBuffer` and `ReceiveBuffer` are intended to be black box data containers for the CAS. They contain state that is only relevant for the transport protocol. For example, for TCP these objects would contain the contents of the buffers, information about the send and receive window sizes and sequence numbers. The CAS has no need for this information and must not be able to change this information.

4.6 Network status feedback for the application

The CAS offers optional feedback to the application if s desired. The feedback system is realized as an Observer pattern [GHJV95]. The application can subscribe to the feedback system if it wishes to receive network status feedback. If not, it should not subscribe and will consequently not receive feedback. If a change in network status occurs, the CAS *actively* informs the application. The application

```
public class FeedbackSessionSocket extends SessionSocket {
    public void subscribe(FeedbackReceiver fr, int type);
    public void unsubscribe(int type);
}

public interface FeedbackReceiver{
        public void statusChanged(CASStatusEvent e);
}

public interface CASStatusEvent;

public class CASNetworkAccessLostEvent
  implements CASStatusEvent {
    public static int TYPE = 1;
}

public class CASNetworkAccessReturnedEvent
   implements CASStatusEvent {
    public static int TYPE = 2;
}

public class CASHandoverOccurredEvent
  implements CASStatusEvent {
    public static int TYPE = 3;
}
```

Listing 4.3: CAS feedback API.

hence must not poll on the CAS socket to be informed about network changes.

Because feedback is optional, legacy applications can also use the CAS with minor modifications. Since legacy sockets have to be modified to use CAS instead of transport protocols and application feedback can be ignored. This way legacy applications will still benefit from suspend/resume behavior without the need for application intervention.

To be able to receive feedback, the application must provide an entry point that the CAS will call when feedback is available. An example of a Java interface for CAS feedback is shown in Listing 4.3. Applications that wish to receive application feedback must use the `FeedbackSessionSocket`. This socket offers the same functionality as the `SessionSocket` (see Listing 4.1), but also includes methods to subscribe and unsubscribe for network status events. Event subscription should be done in a fine grained way, i.e. the application should have to subscribe

to every event type it wishes to be notified of. This has two benefits. First, the CAS must never send network status events to applications that are not interested and, secondly, the application is never bothered with network events it is not interested in. In the Java example, if an application wishes to subscribe to a particular event it must call the `subscribe()` method on the socket. The application must pass two parameters. The first parameter is an object that implements the `FeedbackReceiver` interface. The second parameter is the type of event the application is subscribing to. Unsubscribing for a particular event only requires passing the event type.

To notify an application of a particular network event, the CAS will call the `statusChanged()` method on the `FeedbackReceiver` object that was provided by the application. The argument of that method contains the relevant network event, which can contain more information about the network event that occurred. The application can consult the event type and then adapt its behavior accordingly. The example shows the three events that are currently supported. The `CASNetworkAccessLostEvent` is used when network access was lost. When network access returned, the CAS will send a `CASNetworkAccessReturnedEvent` to the interested application. The `CASHandoverOccurredEvent` is used mainly on the server side when the client did an immediate handover, i.e. a session resumption call was received for a session that was still in the `Established` state.

4.7 Security measures

This section handles a number security issues that are relevant to the CAS. Section 4.7.1 handles security measures that prevent abuse of the CAS protocol. Section 4.7.2 handles two network security solutions that require extra attention when used in combination with the CAS: the use of the CAS in combination with network address translators (NATs) and the use of CAS together with IPSec.

4.7.1 Protocol security

Incorporating the CAS in the protocol stack introduces a number of new vulnerabilities to the protocol stack. Before these vulnerabilities are highlighted, Section 4.7.1.1 first introduces the notion of *attack-equivalence*. Section 4.7.1.2 explains vulnerabilities that are introduced when using the CAS protocol. These vulnerabilities are session hijacking and denial of service attacks, which are then addressed in Section 4.7.1.3 and 4.7.1.4 respectively.

4.7.1.1 Attack-equivalence

The notion of attack-equivalence was introduced by Snoeren in [Sno03]. The attack-equivalent security measures for a particular system feature allow attacks

on the system to be reduced to attacks on the same system that does not contain that feature. In other words, security measures should only address *new* vulnerabilities introduced by the new system feature. If the newly introduced feature suffers from a security problem that was already present in the system before the feature was added, the security measures for the new feature must not address that specific security problem. For example, CAS's connection migration feature does not have to encrypt application data to keep information confidential. The interception of data that was sent in the clear was also a problem with protocol stacks before the connection migration feature was added or it was addressed elsewhere by another protocol or application that was explicitly designed to encrypt and decrypt network data.

It must be noted that the attack-equivalent security concept allows a certain degree of sloppiness. If the system is vulnerable to a particular attack, the developer of a new feature for that system in principle must not address this vulnerability in the new feature, even if this vulnerability also affects the feature. For example, in the case of TCP Migrate (See Section 7.1.2.2), Snoeren does not address man-in-the-middle attacks. The man-in-the-middle attack is a known attack that compromises the Diffie-Hellman authentication scheme, which is used by TCP Migrate. However, Snoeren does not address the consequences of this attack on TCP Migrate because it is also possible to conduct a man-in-the-middle attack on a normal TCP connection. If TCP Migrate would be applied on a protocol that is not vulnerable to man-in-the-middle attacks, it would introduce that vulnerability in the system.

4.7.1.2 CAS vulnerabilities

This section highlights security vulnerabilities of the CAS. The main vulnerability of the CAS protocol is session hijacking. Also denial of service attacks against CAS enabled protocol stacks are possible.

4.7.1.2.1 CAS session hijacking. Because the CAS allows *resuming communication* from an arbitrary location, it is possible for a malicious third party to *hijack a session*. The malicious party must only needs to obtain the ID of the session it wants to hijack. For example, suppose that Eve has obtained the session identifier of a session that Alice is conducting with Bob by means of eavesdropping techniques. Eve can subsequently establish a new transport connection to Alice (or Bob) and issue a resumption request using that transport connection. The CAS of Alice assumes that an immediate handover has occurred, closes the old transport connection to Bob and continues the session by using the connection established by Eve.

While it is fairly easy to resume a session from a different location, it is more difficult to *suspend* a session from a different location. To be able to send a suspension request, one must first have an established session and an accompanying

active transport connection. Before a malicious party can send a suspension request, it must first obtain control over the transport connection or be able to intercept datagram packets. In an attack-equivalent security model, forging suspension requests is not considered a problem, because the underlying transport layer must already be vulnerable to connection hijacking.

4.7.1.2.2 Denial of service. The CAS is also susceptible to three types of denial of service attacks: excessive session establishment requests, excessive session resumption requests and suspended sessions that are never resumed. These three types of attacks are explained in greater detail.

First of all, an attacker can flood a server with session establishment requests. The attacker can send a request and then never reply with the final acknowledgement. The server will resend the server acknowledgement until the client responds or until the session times out. For every new incoming session request, the CAS must reserve some system memory to maintain the CAS protocol state. Consequently, excessive amounts of session establishment requests can quickly deplete the server's resources resulting in the impossibility to accept new sessions.

Secondly, next to an abundance of session establishment requests, the attacker can also flood the server with session resumption requests. These requests can be valid or invalid. In the case of valid resumption requests, the CAS will each time assume that the session is being resumed, possibly from another location, in case of DDOS attacks or in case IP spoofing is used. The CAS will reply to these session resumption requests with an acknowledgement. In case of invalid requests, the CAS only parses the incoming header but then discards the invalid request. Nevertheless, parsing an excessive amount of invalid resumption requests requires a lot of processing power.

Note that the sensitivity of the CAS to such denial of service attacks is closely related to the sensitivity of transport protocols to denial of service attacks. For example, the TCP protocol is vulnerable to TCP SYN flooding attacks [CER96] which results in the inability of the TCP to accept new connections on the attacked listen socket. This indirectly results in a CAS denial of service. However, not all transport protocols suffer from denial of service attacks. Attack-equivalence is therefore not applicable in this case and the CAS should take precautions to prevent flooding attacks.

Thirdly, an attacker could also establish a session, suspend it (anticipatedly or unanticipatedly) and then never resume it. Because one of the main goals of the CAS is to support longer periods of disconnection, the CAS will maintain a suspended session for a long time. In comparison, a session from a client that does not complete the session establishment protocol will only temporarily consume the host's resources because the CAS will clean it up if it does not receive the final acknowledgement in time.

4.7.1.3 Authentication of CAS protocol requests

Session hijacking can be prevented by ensuring that resumption requests are issued by the same peer party that initiated the session. This can be realized using cryptographic authentication protocols that employ a shared secret between the communicating partners. Because it is impossible to know every communicating partner beforehand, a key exchange or key establishment solution will be required. Once the communicating partners have agreed on a shared secret, the server can use it to authenticate future CAS protocol requests, for example, by sending a challenge to the client. If the client responds correctly to this challenge, the request can proceed.

It is not the goal of this work to provide a sound authentication protocol because of two reasons. First, there already exist a number of key establishment and authentication protocols [DH76, Gam85, DvOW92, MTI86] that can be categorized by security properties like entity authentication, data origin authentication, key confirmation, etc. [MvOV97]. Depending on the required security properties, a different security protocol should be used, or a new one should be developed. Secondly, applying a security protocol in a particular computing environment is difficult, mainly because it is hard to prove that the protocol and the used cryptographic techniques really hold the correct properties to obtain the required security goals. Developing a new protocol is then an even harder task. The development and proof of an adequate authentication protocol is therefore subject to future work. However, since security will always be an important issue, it should not be ignored while designing a communication protocol. Therefore, the remainder of this section illustrates how key establishment and challenge/response protocols can be incorporated in the CAS protocol.

4.7.1.3.1 Key establishment in CAS. Key establishment in the CAS is realized by means of an extension header that is only used during session establishment. Hence, while the client and server exchange the session identifier, they are also establishing a session key that will be used to authenticate future session suspension and resumption requests. This extension header is similar to the header that is used to synchronize both CASs during session resumption (see Section 4.3). The length of the extension header depends on the used key establishment protocol and the key length used in that protocol.

By means of illustration, we show how a key can be established by means of the Diffie-Hellman key agreement protocol [DH76]: Diffie-Hellman requires the exchange of no more than two messages to realize key agreement. In the CAS this can be realized as follows (see Figure 4.12). The first Diffie-Hellman protocol message is sent along with the session establishment request sent by the client. The client hence always initiates the Diffie-Hellman exchange. The server sends the second protocol message in the server acknowledgement. At that moment, both parties have a shared secret without knowing what secret key (x and y) the

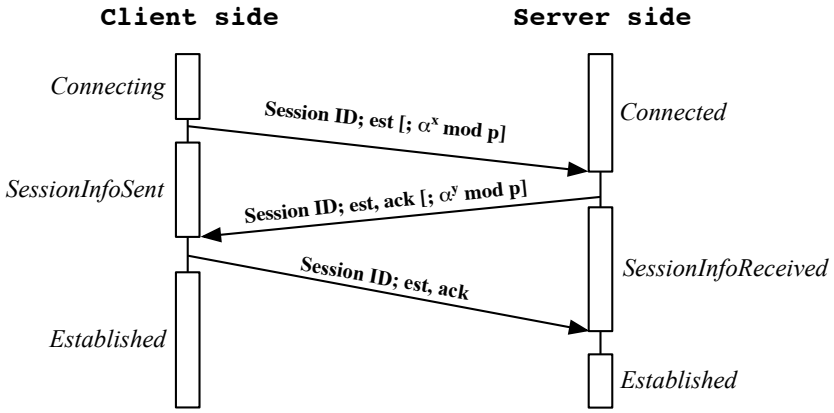


Figure 4.12: Session establishment with Diffie-Hellman key agreement.

other party uses. Obviously other key agreement protocols can be realized during CAS session establishment, as long as they fit in the three way handshake protocol.

4.7.1.3.2 Challenge/response in CAS When the server receives a request to resume (or suspend) a session, it must be able to check that the validity of the request. If the server receives a session resumption request, it sends a challenge to the client. Such a challenge must ensure among others that the requester is the initial establisher of the session, that the request is fresh (has not been stored for an arbitrary amount of time), that the request is not a replay, etc. As an example, a challenge could be a randomly created nonce that is encrypted using the key that was agreed upon during session establishment. The server sends this challenge in an option header together with the CAS header that acknowledges the session resumption request (see Figure 4.13). The client must decrypt and encrypt this challenge with its own key and send the result back as a response to the challenge. If the response is not valid, the server moves the session back to the suspended state, sends an error header to the client and closes the transport connection.

4.7.1.4 Preventing denial-of-service

Preventing denial-of service is not a trivial task because from the CAS's point of view, suspending for longer periods of time and receiving large numbers of session establishment, suspend and resume requests can be perceived as normal behavior. Snoeren also made this observation when dealing with denial-of-service attacks in his Migrate [Sno03]. Snoeren proposes to prevent denial-of-service attacks by

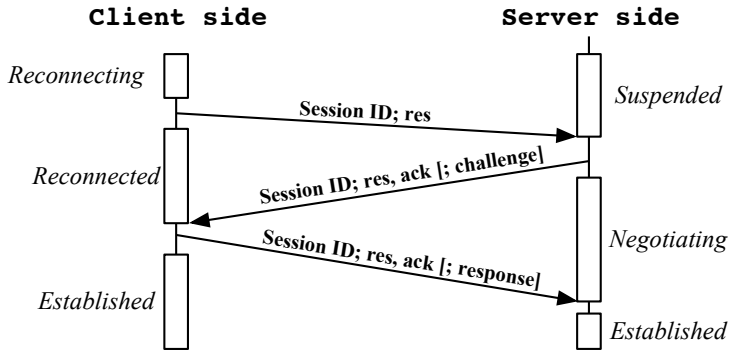


Figure 4.13: Verifying the validity of a session resumption request by sending a challenge to the client. The client must reply with a valid response.

designing policy control mechanisms that detect and prevent actions that may exhaust the system's resources.

To avoid that suspended sessions consume all the system's resources, sessions that are no longer considered valid should be cleaned up, i.e. the resources occupied by that session should be freed. Invalid sessions are sessions that have been suspended for a too long time. Also the number of sessions that can be suspended should be limited to prevent resource depletion. The amount of time the CAS should wait before discarding a session and the maximum amount of sessions that may be suspended depends greatly on the application type and the execution environment. For example, consider a video server that allows to stream the latest news broadcast from a television station. On this server there exist sessions to stream video, but also sessions that were established by the system operator to configure and monitor the system. The cleanup policy of a suspended management session and a suspended video streaming service will be different. A management session should not qualify for cleanup, even if it is suspended longer than all the suspended video streaming sessions. Therefore, the application should determine what session is valid or invalid.

To be able to prevent denial-of-service, the session socket is extended with the methods shown in Listing 4.4. These methods allow to tune the CAS's cleanup policy with a number of parameters. For every session, the application can set a timeout on a client session's socket by calling `setMaxSuspendedTime(long seconds)`. If a session is suspended for a longer time it will be considered for cleanup. The maximum amount of client sessions that can be in a suspended state can only be defined in the context of a listening socket. The server application can call `setMaxSuspendedSessions(int amount)` to indicate how many client sessions

```
public class SessionSocket {
    setMaxSuspendedTime(long seconds)
      throws SessionSocketException;
    setMaxSuspendedSessions(int amount)
      throws SessionSocketException;
    setRequestRate(int amount);
}
```

Listing 4.4: CAS socket methods to avoid denial of service attacks

that were created using the listening socket are allowed to be in a suspended state. If this amount is exceeded, client sessions with the smallest timeout value left (set with `setMaxSuspendedTime(long seconds)`) will be considered first for cleanup. Note that sessions will only be *considered* for cleanup. The suspended sessions may still be retained if the system's resources are not exhausted, even if the maximum suspension time has expired or if the number of suspended sessions exceeds the configured amount.

To avoid having to handle excessive amounts of session establishment, suspension and resumption requests, the application can call `setRequestRate(int amount)` on a session socket. In case of a listening socket, this limits the number of session establishment requests the socket can receive in a minute. If called on a client socket, this limits the amount of suspension requests that the client session will handle in a minute. If the amount of CAS requests are exceeded, the CAS simply drops the request.

It should be clear that offering these methods alone does not prevent denial-of-service attacks. Reasonable parameter values must be chosen when a CAS enabled application is deployed. These parameters will depend on the environment in which the application is deployed, i.e. the resources on the machine on which it is executing, the application's importance with respect to other applications running on the machine, etc. The parameters should hence not be hard coded in the application.

4.7.2 Network security

To be able to use the CAS in contemporary networks, some additional measures are required. First, if the CAS is used in combination with NATs and firewalls, some extra configuration is required on those NATs and firewalls. A number of configuration alternatives are explained in Section 4.7.2.1. Section 4.7.2.2 shortly discusses the use of the CAS in a network where hosts are IPSec enabled.

4.7.2.1 NATs and firewalls

In contemporary networks, a lot of firewalls are deployed to protect local networks against unwanted network traffic. There is a risk that the behavior of CAS enabled computers interferes with the policies on firewalls that are on the path between the communicating endpoints. In this discussion, we will differentiate between network address translators (NAT), connection tracking firewalls and stateless firewalls. Contrary to stateless firewalls, connection tracking firewalls and NATs are stateful firewall technologies that maintain information about the filtered network traffic and can adapt their filtering rules dynamically. During the following discussion, we will use also the following terminology. A NAT host or firewalled host is a host that is protected by a NAT or firewall respectively. We refer to a CAS client as a host that initiates a session, causes a session to suspend, or tries to resume a session. The CAS server is the host communicating with the CAS client. Both a CAS server and a CAS client can be a firewalled or NAT host. In a client-server application, we will just refer to the communicating parties as application client and application server. Note that in such a case the application server is not necessarily the CAS server. If the application server tries to resume a CAS session, it will be referred to as the CAS client. For reasons of clarity, we will also assume we're operating in a TCP/IP environment.

Firewalls do not interfere with CAS behavior when establishing a session or suspending a session anticipatedly. Before a session can be established, one must be able to establish a transport connection or send a datagram. This is no different from traditional networking where CAS is absent from the protocol stacks. If a CAS session has been successfully established between two communicating hosts, i.e. they are not blocked from each other by the firewall, they can also suspend a session.

Only session resumption requires extra attention in a network with firewalls because the firewall situation may change after a host moved and is trying to resume its sessions. In the following discussion, we will always assume that the CAS client moves and that the CAS server remains in the same location. The case where both CAS hosts move is discussed at the end of this section. Before a CAS client can resume a session, it must first establish a new transport connection to the CAS server from its new location. In case the CAS server is also an application server, it will be reachable on a well known address. Session establishment will be successful if the firewall policy allows establishing a new connection from the CAS client's new location to the server.

In case the CAS server is an application client, firewalls need extra configuration because they normally do not allow new connections to client machines. We consider three possibilities configuration solutions. A first possible solution is to allow connections to ephemeral ports. Clients use random ephemeral transport protocols ports when communicating with a server which listens on a fixed, well-known port. The CAS can then send the session resumption request to the

ephemeral port the client was using before session suspension. The downside of this solution is that every client host in the protected network is again exposed to the outside network. Also, connections from the public network to a NAT host are normally not possible. A second solution is using port forwarding. Every CAS instance, including the CAS running on an application client, must be listening on well known ports, one for each possible transport protocol. This allows the firewall to be configured with a port forwarding rule for each transport protocol used on a firewall protected CAS server. Consequently, session resumption requests can penetrate the firewall. The downside of this solution is scalability: port forwarding must be done for each CAS server and each transport protocol used by those CAS servers. A third solution is using a connection tracking firewall or NAT. The connection tracking functionality can be extended to track CAS sessions as well and dynamically adapt the ruleset to allow new transport connections to application clients that expect session resumption requests. Connection tracking firewalls and NATs already do this for application protocols that use multiple transport connections such as FTP. Such firewalls monitor the FTP control channel to learn what new data connections will be established to hosts in the protected network. The firewall then temporarily allows the negotiated data connections. The downside of this solution is that every firewall must be equipped with CAS session tracking functionality.

The remainder of this section explains how the connection tracking functionality of a NAT must be extended to support session resumption requests. Connection tracking firewalls are a simplified case of a NAT because they contain the same connection tracking functionality as NATs but without address translation. The extension hence also applies for normal connection tracking firewalls. We again assume that the NAT host is a CAS server and does not move in the network. The NAT host is also an application client, and is therefore not reachable by means of NAT port forwarding.

In normal circumstances, it is not possible to establish a new connection to NAT hosts because they are assigned a private network layer address that is not accessible from the public network. A NAT maintains an address translation table. For every transport connection from a NAT client to a public host, this table contains a mapping between the private network layer address and transport protocol port of the NAT host and the public network layer address and transport protocol port that the NAT uses when sending the packet on the public Internet. Packets that arrive from the public network are matched against this table, have their destination network layer address and transport protocol port changed accordingly and are then forwarded to the correct NAT host. If a mapping is not found in the table, the packet is dropped. If a CAS client wishes to resume a session, it should hence direct its packets to the NAT's public IP address with a destination port that will resolve to a valid private address/port combination.

Because the NAT host is a CAS server that does not move, the peer host

can send packets to the NAT's public IP address and the transport protocol port that was used before migrating. There are two requirements however. First, the NAT must accept packets from a different source address that are destined to an address and port tuple for which there already exists an entry in the address mapping table. NATs must hence relax their matching policy to allow packets from an unknown source address as well. For example, TCP packets destined to an existing port and addressing tuple should also be allowed when the SYN flag is set. Additionally, the NAT should also extend connection tracking to incorporate the session ID. If the NAT sees a packet that carries an existing session ID, the NAT can allow it, regardless of the used source network layer addresses and and transport layer addresses. It must be noted though that incorporating the session ID in the connection tracking algorithm is only possible for datagram transport protocols. If connection-oriented transport protocols are used, a number of connection establishment packets must first be exchanged before the first CAS headers can be exchanged.

Secondly, the NAT must maintain address mappings for transport connections that no longer exist but belonged to CAS sessions that are now suspended. A NAT normally discards an address mapping when a connection is closed normally (anticipated suspension) or when a connection is aborted (results in unanticipated suspension). The NAT should maintain the mapping so the CAS client can again send the resumption request to an address and port tuple that can be resolved in the table. This mapping should be maintained as long as the CAS session is considered valid. A suspended CAS session is valid as long as both the CAS client and server have not discarded the suspended session. Because a NAT cannot determine if a CAS client or server discarded a suspended session, it is not clear how long the NAT should maintain the address mapping. A similar problem exists in current NAT implementations for idle TCP connections. There is no way for a NAT to learn when an endpoint of an idle TCP connection disappeared, for example, because its host rebooted.

If the client is located behind a firewall instead of a NAT, the same requirements apply. The main differences with a NAT is that a connection tracking firewall monitors the same transport connections, but does not change the addresses. A connection firewall should hence also accept packets from a different source address and with destination an address and port tuple that exists in the connection tracking table. The firewall should also maintain mappings for closed connections in a similar way as a NAT.

Up till now we have assumed that the CAS server (which is an application client) did not move. If both CAS client and server move, the CAS client must first learn the new address on which the CAS server can be reached before it can attempt resumption. If the CAS server moves to another location which is again behind a NAT or firewall, the CAS client won't be able to reconnect unless the CAS server's firewall or NAT policy allows that. If the firewall allows connections

to ephemeral ports, the CAS client can communicate with the CAS server on its new address using the previously used port. In case of port forwarding, new port forwarding rules must be configured on the new firewall each time a CAS server enters the firewalled network and he old port forwarding rules must be removed on the old firewall. This requires close cooperation between the CAS host and the firewall. Similar cooperation is required in case a connection tracking firewall is used. The CAS server's new firewall must be able to adapt the connection tracking table to the new situation. It must be noted that a CAS server will also try to reconnect after it moved. Because the CAS client is an application server, it should be possible for the CAS server to connect to the CAS client and resume the session, unless the CAS client's firewall policy does not reconnections and/or session resumptions from the CAS server's new location.

4.7.2.2 Using IPsec

The CAS supports the addition of authentication protocols (Section 4.7.1.3), so endpoints can authenticate themselves when resuming a session from a different location. However, if endpoints can authenticate themselves properly using alternative security solutions that exist outside of the session layer, the CAS does not have to take care of authentication. IPsec [KA98] is such a security solution that is realized in the network layer. IPsec is used in VPN solutions and is a mandatory part of the IPv6 protocol. The remainder of this section gives a short overview of IPsec and discusses how IPsec can be used for authentication in a dynamic network with a mobility solution like CAS.

IPsec can be used to provide authentication and payload encryption. IPsec supports two main modes of operation: tunnel mode and transport mode. Tunnel mode is used to create secure IP channels between gateways or between gateways and end hosts. Transport mode is typically used to secure network traffic between two endpoints. IPsec typically operates on a per-packet basis, although the RFC also mentions socket based IPsec implementations. Every packet that is sent through the IPsec layer is checked against a number of *selectors*, comparable to the rulesets used by a non-stateful firewall. These selectors are implemented in two databases: the security policy database (SPD) and security association database (SAD). If a packet matches the configured selectors, the corresponding security association (SA) is applied. Such a SA defines whether the packet must be authenticated or encrypted and also contains the required cryptographic keys.

IPSec has been used in mobile environments where it is mostly used in conjunction with Mobile IP to secure traffic from and to mobile hosts [Bin01, Cis06]. IPSec uses the home address of the mobile node as part of the selector to consult the SPD and SAD. Since the home address of a mobile node never changes, the same SA will be applied regardless of the mobile node's current location.

If IPsec's end-to-end (transport mode) authentication scheme has the required security properties for the application domain in which CAS enabled hosts will

be deployed, it is not necessary to add an authentication protocol to the CAS protocol. IPSec can be used instead. The main difference with applying IPSec in conjunction with Mobile IP is that the CAS does not use the concept of a home address. IP address changes are thus not hidden from IPSec. Consequently, IPSec selectors used to consult the SPD and SAD cannot rely on the mobile node's IP address. Instead, IPSec must be configured to use selectors that are resilient to changing network addresses. The IPsec RFC mentions the use of user IDs as selectors instead of addresses, which is an example of a selector that would not change as a consequence of mobile behavior.

Chapter 5

The Address Management System

5.1 Introduction

Currently, applications are always programmed with a particular protocol stack in mind. For example, a web browser is typically designed to run on an IPv4 network and uses TCP to retrieve data from a web server. Moreover, applications must explicitly specify the network layer and transport layer protocols they will use to communicate. For example, to create a socket using the BSD socket interface [Ste90], the application must pass 3 parameters. The first parameter is the address family, which boils down to the network layer protocol one wants to use. The second parameter is the socket type, such as a datagram or stream socket and the third parameter is the transport layer protocol needed to implement the socket type. To establish a connection with a peer communication partner, the application must use network layer addresses and transport layer addresses (referred to as ports in the TCP/IP protocol suite) that adhere to the socket's protocol specifications.

This tight coupling between application and protocol stack obstructs application deployment in a heterogeneous network environment, such as dynamic networks. Applications that are programmed to use a fixed protocol stack cannot adapt to the environment they are deployed in. Even if an application is programmed to support a number of protocol stacks, it will never support all probable protocol stacks nor will it be able to adapt to use new protocols.

The main goal of the address management system (AMS) [MMMV01, MJV03] realized in this dissertation is to decouple the application from a particular protocol stack. More specifically, the AMS allows the application to be developed independently of a particular protocol stack and allows the application to run in

different network environments without adaptations. Section 5.2 gives a high level overview of the AMS solution. In Section 5.3, the AMS is studied from a more technical point of view. Finally, Section 5.4 describes how the AMS and CAS complement each other in a mobile, heterogeneous network.

5.2 High level overview of the AMS

The AMS removes the tight coupling between the application and the protocol stack by eliminating the explicit knowledge of the addresses and protocols that are required to communicate with the peer host. With the address management system, applications no longer use explicit addresses and must never instruct the protocol stack what protocols to use. The address management system takes over this functionality instead.

The AMS's main abstraction is the *generic address*. A generic address is a container that holds all information needed to communicate with a particular peer application, i.e. the protocols and addresses by which the peer application can be reached. This peer application can be a traditional service like a web server, but also a peer-to-peer application running on a client terminal. A generic address is a black box for the application. An application has no access to the address and protocol information in the generic address. This information is only intended for the AMS. The application uses a generic address only for service identification purposes.

If an application wishes to communicate with a service, it must first obtain the service's generic address. The AMS does not specify how an application must obtain a generic address. For example, a name service can be used to obtain the generic address for a service. When an application possesses the generic address, it creates a socket and connects to the service using the generic address. At that moment, the address management system *reduces* the generic address: it interprets the protocol and address information in the container and matches it with the protocols used by the client host to communicate at that moment. When a generic address is reduced, the AMS instructs to protocol stack to create a transport protocol connection to the service using the selected protocols.

The communication protocols that an application must use and the addresses and ports on which a service can be contacted must no longer be chosen beforehand. All possible ways to communicate with a service are encapsulated in a generic address. The AMS interprets the information in a generic address dynamically, and adapts the used protocols to the environment in which the client is currently running.

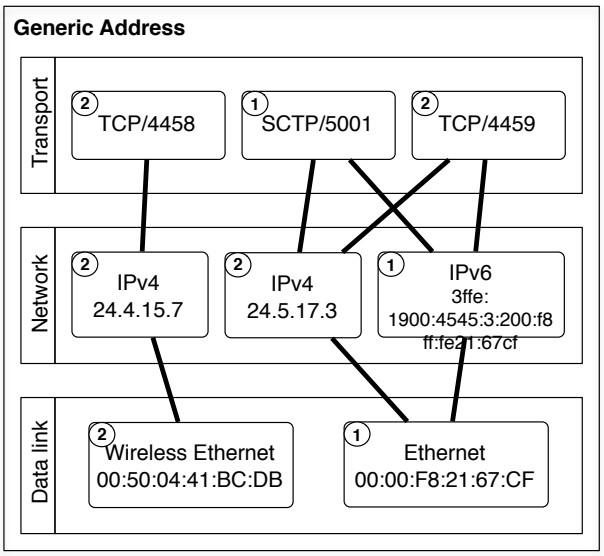


Figure 5.1: An example of a generic address

5.3 Technical realization of the AMS

5.3.1 Generic addresses

A *generic address* represents a network service. A generic address encapsulates all possible protocol and addressing information that can be used to contact a network service. Figure 5.1 shows an example generic address. The generic address holds address information per protocol stack layer: data link layer, network layer and transport layer. The protocol addresses are depicted as rounded rectangles. For the transport, network and data-link layer, these protocol addresses are respectively transport protocol ports, network layer addresses such as IP addresses and MAC addresses. In every layer, protocol addresses can belong to different protocols. In the example, there are TCP and SCTP protocol addresses in the transport layer. On the network layer both IPv4 and IPv6 are represented. In the data link layer the different protocols represent the network medium types on the service's host machine that can be used to consult the service. In the figure, these are ethernet and wireless ethernet.

Data link layer addresses are usually not required to contact a service, because the network layer hides such details from higher layers. Nevertheless, the data link layer addresses are still included in case the network layer is not required or available for a particular service.

A generic address also dictates what *protocol address combinations* (PAC) can be used to contact the service. A service is typically contacted using a combination of one transport layer address, a network layer address and, implicitly, a MAC address. A PAC consist of exactly one protocol address from each layer. In Figure 5.1 these protocol address combinations are depicted by thick lines connecting protocol addresses on different OSI layers. The lines that connect a network layer protocol address with a data link layer protocol address have a special meaning. They indicate that the interface with that particular MAC address is configured with the connected network layer protocol address.

Some specific protocols can be favored. This is indicated by protocol address preferences per layers. In the figure these preferences are presented by numbers in the top left corner of a protocol address. The lower the number, the more the service provider prefers the clients contacting the service using that protocol address. These preferences are merely an indication for the application that wants to connect to the service. It can ignore these preferences, but choosing other protocol addresses may result in deteriorated service handling.

The example service in Figure 5.1 is available on a multi-homed host. It can be reached on a wired and a wireless ethernet interface. The host is reachable on two IPv4 addresses and one IPv6 address. The service can be contacted using TCP on ports 4458 and 4459 or SCTP on port 5001. The protocol combinations for this generic address are as follows. Using the wireless ethernet interface, the service can be reached on IP address 24.4.15.7. The wired ethernet interface is configured with both an IPv4 and an IPv6 address. The service is preferably accessed on the wired interface, as indicated by the number one in the circle in the top left corner of the data link layer address. The wireless ethernet interface has preference two. If the service is contacted on the wireless ethernet interface, the selection of the IPv4 address and the TCP port is automatic because the interface is only configured with one address (24.4.15.7) and on that interface the service is only available on TCP port 4458. If the service is contacted using the wired ethernet interface, the client is encouraged to contact the service via IPv6, because the IPv6 address has priority one. If this is not possible, for instance because the client does not have a protocol stack that supports IPv6 or because it is currently located in an IPv4 network, the client can also contact the service using IPv4. The choice of IPv4 or IPv6 does not affect the choice of transport protocol port. Both SCTP (port 5001) and TCP (port 4459) can be used to contact the service independent of the selected network layer address. Having a priority number one, SCTP is preferred.

From the point of view of the application a generic address represents a service. This allows generic addresses to be used in a different way. Generic addresses allow a service to be implemented by a pool of servers. This is realized by adding the network layer addresses of all the servers in the pool in the service's generic address. When the client's selection process chooses a network layer address, it selects a different host rather than a particular network interface on the same host. In this

perspective, the preferences create a hierarchy in the server pool. For example, the protocol combination with the highest preference may represent the server with the most processing power in the pool.

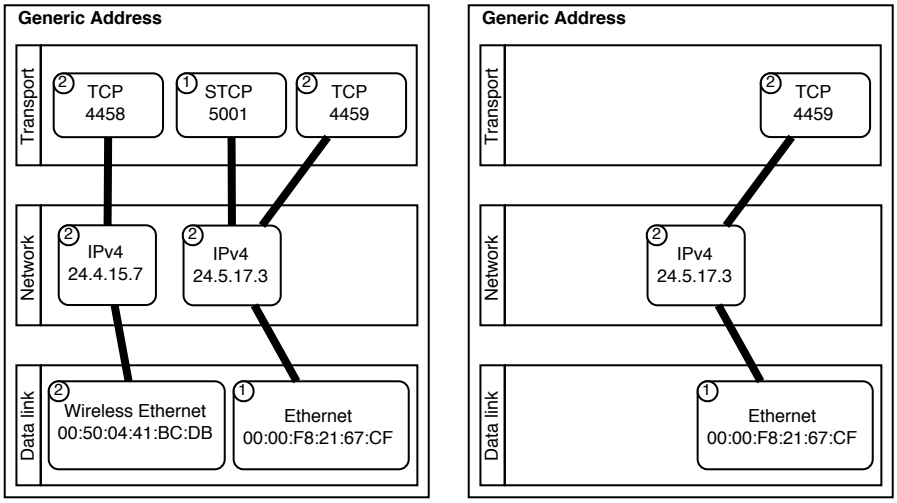
5.3.2 Reducing generic addresses

Before an actual transport connection can be established, a unique selection must be made from all the addresses and protocols available in a generic address. This selection process is called *reduction* and is broken down in protocol reduction and address reduction. The selection process happens entirely on the host establishing the connection. No third party is contacted that may influence the selection process. The selection process takes the host's network capabilities and access network into account.

Protocol reduction is selecting the protocol combinations in the generic address that can be used to communicate with the service represented by the generic address. The resulting protocol combinations can be deduced as follows. Transport protocols that are adequate for communication can be found by matching the transport protocols in the client's protocol stack and the protocols represented in the generic address. On the network layer, adequate protocols can be obtained by taking the intersection between the network layer protocols used by the connecting host and the network layer protocols represented in the generic address. Note that data link layer protocols are normally not involved in the protocol reduction process, because the data link layer address is not determinative to establish a transport protocol connection. The network layer makes abstraction of the used data link layer address.

After protocol reduction, only the protocols that the client can use to communicate with the service will be left in the generic address. It is important that there is at least one PAC left. Otherwise the client will not be able to consult the service. For example, after protocol reduction, the address from Figure 5.1 may look like the address in Figure 5.2a. In this case, no protocol has been eliminated from the transport layer. This means that the client is capable to use both TCP and SCTP. On the network layer, the IPv6 protocol has been eliminated. A possible cause is that the client is not IPv6 capable or that it is currently operating in an IPv4 network.

The next step after protocol reduction is *address reduction*, which will select one protocol combination from the remaining PACs. During address reduction, the connecting host outweighs its own preferences against the preferences in the generic address. The host may adhere to the priorities in the generic address to select certain transport or network layer protocols. Other hosts may prefer to use a particular communication medium regardless of the used protocols. They may also choose to completely ignore the generic address's priority scheme. After address reduction one PAC will be left, which will be used to establish a transport connection.



a) After protocol reduction

b) After address reduction

Figure 5.2: The generic address from Figure 5.1 after a) protocol reduction and b) address reduction.

The example address from Figure 5.1 may look like the address in Figure 5.2b after protocol and address reduction. The connecting host chose to use the preferred interface in the generic address, which is wired ethernet interface (priority 1). The choice of the IPv4 address follows automatically. The connecting host chose to ignore the transport protocol preference in the address though. Although SCTP was still in the generic address after protocol reduction, which means that the host was capable to handle SCTP, it selected TCP as the transport protocol to use. Possible reasons could be that the performance connecting host's SCTP implementation is not good, or simply a matter of user preference where the user is reluctant to use new protocols.

It must be noted that the selection process does not take into account the network status of the remote host, or hosts in case of a server pool. Such information could help the selection process because it could indicate that the remote host is no longer reachable on a remote interface, or it could give an indication of the load in the server pool. Such information could be represented by the priority schemes in the generic address, but would require a network environment where generic addresses can be updated dynamically to reflect the state changes of the service.

5.4 The CAS and AMS synergy

The CAS and AMS together result in a system where applications can run on a terminal that moves between heterogeneous networks. We shortly discuss the contributions of both systems to the application's dynamic network execution environment.

The AMS facilitates protocol and address independence. Applications must no longer be developed with a limited set of communication protocols, which only allows them to be deployed in network environments using these protocols. Instead applications use generic addresses, which make the application completely protocol agnostic. Consequently, applications can be deployed without adaptations in different network environments that are realized with different communication protocols. However, the AMS does not allow hosts to move between different access networks. The AMS alone is therefore not useful in a dynamic networks characterized by mobile terminals.

The CAS allows hosts to move between different access networks. It facilitates the migration of sessions, suspension of sessions in case of long periods of disconnection and session resumption when network connection returns. However, the CAS does not support protocol changes when migrating or resuming a session. The CAS alone is therefore not useful in a dynamic network characterized by protocol heterogeneity.

When combined, the CAS and AMS form a mobility solution that is adequate for a dynamic network. The CAS takes care of the migration of sessions and depends on the AMS to handle protocol changes that can occur when moving between two networks that use different communication protocols.

Chapter 6

Realization and Evaluation

This chapter evaluates the design of the CAS and the AMS, evaluates an implementation of both systems and describes how both systems have been evaluated in industrial projects. First, Section 3.5 verifies if the CAS and AMS design is compliant with the proposed architecture. Secondly, the implementation of the CAS in the DiPS+ framework is discussed. Section 6.2 is a short introduction to the DiPS+ protocol stack framework. Sections 6.3 and 6.4 describe the design of the CAS and AMS in DiPS+ respectively. Section 6.5 discusses performance issues of the CAS/AMS approach and the DiPS+ implementation. Thirdly, Section 6.6 shortly describes how both the CAS and AMS have been successfully applied in two industry projects.

6.1 Architecture compliance

The solution proposed in Chapters 4 and 5 is an end-to-end approach to dynamic networks (see Section 3.4) that only requires a specialized name service if the AMS is used. If the AMS is not used, existing dynamic name services can be used to locate endpoints. The name service must be dynamic so the coordinates of an endpoint can be easily updated after a move to another location.

The remainder of this section discusses if and how the CAS/AMS approach realizes the session management tasks that were outlined in Section 3.5. For each task, the goals are shortly outlined in *italic*.

Session support detection

A SeLMS must be able to detect whether the peer communication system supports the SeLMS extensions. It is not likely that all protocol stacks will suddenly be equipped with a compatible SeLMS.

Currently, there is no unique way to detect if a remote system also supports CAS/AMS extensions. Session support detection solutions should be completely transparent, also for systems that do not support the extensions. This is obviously a goal that is hard, if not impossible, to achieve.

Existing solutions to dynamic networks often exploit the particular usage of a protocol (see Chapter 7) to detect if the remote system also supports the dynamic network extensions. The CAS does not use such an approach because these detection techniques are transport protocol specific and therefore not generally applicable. Other detection approaches involve a third party, like contacting a network service on the remote system or consulting a directory service. In case the daemon is contacted successfully or the directory lookup result indicates that the remote system supports the extension, communication can continue with the protocol stack enhancements in place. Otherwise, the system must resort to traditional network communication.

The easiest way to implement detection is by using a third party. Because there exist numerous solutions to interact with a third party, we choose not to specify session support detection as part of the CAS. The performance and adequacy of a particular approach will vary between different deployment environments. The CAS therefore does not impose any limitations on the used detection technique.

Transport/network protocol independent session identification

A session must be identified independently of the transport protocol connection (TPC) identification mechanisms used by the lower transport and network layer. This is necessary because TPC may break and be replaced by other TPCs during the lifetime of a session.

CAS sessions are identified using universally unique identifiers (UUID). UUIDs are independent of the identifiers of the TPC used to realize the session's communication. The UUID of a session remains fixed, even if its associated TPC breaks and is replaced by a new TPC.

Protocol and address hiding

For an application to function properly in a dynamic network environment, the session layer solution should hide the used communication protocols and the addresses that these protocols use to communicate from the application. This allows applications to run in different network environments and move to different environments, regardless of the used protocols.

If the CAS is used in conjunction with the AMS, the network protocols and addresses used to communicate are hidden from the application. The AMS also hides protocol alternatives to communicate with a service by making a protocol selection on behalf of the application.

The AMS only hides which particular protocols are used, not the type of service offered by these protocols. A transport layer protocol realizes a particular network service, such as reliable data transport. The way in which an application interacts with the protocol stack depends on the expected service. When a CAS/AMS socket is created, the application must only specify what type of service it wishes to use: a reliable data stream service or an unreliable datagram service. The application hence must not choose beforehand a protocol that realizes that service. In the case of traditional, non-CAS/AMS sockets, the application must also specify what protocols must be used.

Consequently, when a CAS socket is used, the application is not aware of what protocols are used to realize the communication service. The protocols can therefore change without burdening the application with the technical consequences.

Session state management

A SeLMS must maintain the state of every session. A session can be closed, connecting, active, suspended or reconnecting. The SeLMS's behavior for a session will depend on the state of that session.

The CAS maintains a state transition diagram for every session. This state transition diagram reflects the current network situation of a CAS session. The state of a session is directly related to the condition of the related transport protocol connection (TPC). If the host is connected to the network and the session's associated transport connection is established; the session is active. If the host disconnects from the network or if the TPC breaks due to timeouts or other network errors, the session becomes suspended. If the host reconnects to the network, the session will be in the reconnecting state where it tries to establish a new TPC.

CAS state management also reflects the status of the transport connection when the session is moving between closed, active and suspended and reconnecting states. This is the main reason why the CAS state transition diagram contains 23 states instead of the five states shown in Figure 3.4. These additional states in the CAS's transition diagram are mainly needed if the CAS is used in combination with unreliable transport protocols. In this case, a number of additional checks are required to verify whether a session was successfully established, suspended anticipatedly or terminated. The extra states reflect these additional verifications.

Session negotiation protocol

To coordinate session management between two communicating endpoints, a session negotiation protocol is required. This protocol must support the establishment, suspension and resumption of sessions.

The CAS implements a session negotiation protocol which is driven by the CAS state machine. The protocol handles the negotiation of session establishment, i.e.

negotiating a session identifier, anticipated session suspension and session resumption. All protocol negotiations are realized as a three way handshake in case unreliable transport protocols are used.

Protocol negotiation messages are exchanged by means of a session header attached to each packet transmitted on the network. Session protocol messages and application data are hence exchanged using the same transport protocol connection (TPC). Other mobility approaches that are deployed in dynamic network environments often use separate TPCs for session control and data transfer.

Transport protocol management

An SeLMS uses TPCs to exchange both application data and session management. The SeLMS must hence be able to establish and tear down TPCs, send and receive data using that TPC and detect that TPCs are aborted.

The CAS interacts with the transport layer using direct function calls (or system calls) because it is implemented as a protocol stack layer. These function calls may differ between protocol stack implementations, but typically have the same semantics. A CAS hence does not use the traditional socket API that are used by the application layer.

Maintaining communication channel semantics

A SeLMS must be able to offer the requested communication service, such as reliable communication, to the application. If reliable communication is realized by means of a connection-oriented transport protocol that is not equipped to cope with mobile endpoint behavior, TPCs are aborted and data may be lost. In that case, the SeLMS must take the appropriate measures to ensure reliable communication.

The CAS offers two mechanisms to maintain reliable communication (byte stream consistency): double buffering or TPC state exporting and importing. The first solution is more generic and will work with any transport protocols, the second solution is more performant, but requires support from the transport protocol implementation. The CAS proposes a programming interface that must be implemented by protocols allowing state export and import.

These mechanisms are only applied if the application needs reliable communication. If there is no need, transport layer protocols that do not offer reliable communication will be used. In that case, the CAS will not compensate data loss.

Offering application feedback

A SeLMS should provide optional network status feedback to the application. This allows an application to adapt when network events occur that affect the

application's business logic. Example events are bandwidth drops or network disconnection.

The CAS offers optional network status feedback to the application. Applications that wish to receive feedback can subscribe to the CAS's application feedback service. Feedback is realized as an event driven mechanism, i.e. the application must not poll on the session socket to learn about network status changes. The CAS actively notifies the application if a status change occurs. Applications receive feedback when session state changes occur: if the session becomes suspended, if the session resumes, or if an immediate network handover occurs. Legacy applications remain unaffected. They will typically not subscribe to receive network status feedback and will hence receive none.

6.2 DiPS+ overview

We have implemented the CAS and AMS in the DiPS+ protocol stack framework. This section gives an overview of the DiPS+ framework. The following sections describe the realization of the CAS and AMS in DiPS+

The DiPS (for **Di**strinet **P**rotocol **S**tack) framework [Mat99][DRG] is a software component framework specifically designed for the development of protocol stack software. It allows the programmer to create a protocol in a layered protocol stack, while protocol functionality and non-functional concerns like concurrency or queueing policies can be managed separately. A protocol can be created by developing and composing a number of functional components, like header constructors, header parsers, routing components, send and receive buffers. These components are *black box components* and, depending on their generality, can be reused in different protocol implementations. Non-functional concerns are realized by adding separate components to the composition that implement concurrency functionality (e.g. thread pools and semaphores) or queues (e.g. leaky buckets) etc.

DiPS has evolved to DiPS+ [Mic03] which improves the framework with better software engineering support and adds runtime protocol stack monitoring. Software engineering improvements are mainly in the domain of component testing [MWJV02a, MWJV02b] and a more uniform model of the DiPS core concepts. Protocol stack monitoring offers the possibility to observe protocol stack behavior at runtime and adapt the protocol stack (e.g. alter thread allocation) to improve protocol stack performance [MDJV04, MDJ+02]. Additionally, an extension called CuPS [JMMV02] allows the component composition of a DiPS+ protocol stack to be changed at runtime. This boils down to the ability to remove or replace protocol functionality or even entire protocols while the protocol stack is operational.

The following sections describe the core building blocks of the DiPS+ framework. Only those concepts necessary to understand the design of the CAS and AMS prototype in DiPS+ are described. For a more detailed description of DiPS+

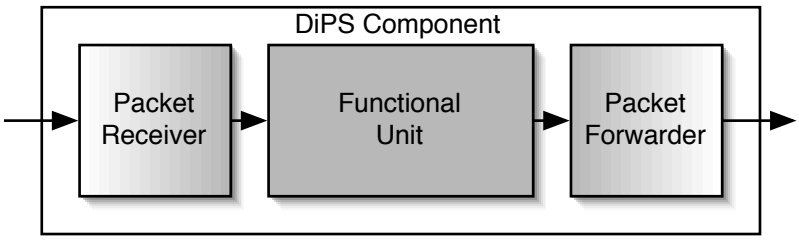


Figure 6.1: DiPS+ component anatomy

we refer to [Mic03]. Sections 6.2.1 to 6.2.6 describe respectively DiPS+ packets, components, connectors, layers, layer resources and the framework communication mechanisms used in the framework. Section 6.2.7 discusses why DiPS+ addresses dynamic network challenges on protocol stack flexibility (Section 2.1.4).

6.2.1 DIPS+ packets

The core data structure of the DiPS+ framework is the packet. A packet reflects the common perception of a network packet in a packet switched network. It contains application data (payload) and a number of protocol headers that are added when the packet is being processed by the protocols in the protocol stack. The same packet data structure is used for packets that must be transmitted and for packets that were received on a network interface.

6.2.2 DiPS+ components

Where a DiPS+ packet is merely a data container, a component is the most basic DiPS+ framework construct that can be used to process a packet. A component's internal anatomy is threefold, as shown in Figure 6.1. A component possesses exactly one entry point (`PacketReceiver` or PR) and can have multiple exit points (`PacketForwarder`s or PF). The component's `FunctionalUnit` contains the processing logic of the component. The PR and PF are offered by the DiPS+ framework and must normally not be altered or changed by the protocol programmer. The protocol programmer's main task is to implement the component's `FunctionalUnit`. The `FunctionalUnit` can realize a small protocol task, such as calculating a checksum, or it can implement a packet queue, a semaphore, etc.

The PR and PF are modeled explicitly for protocol composition and component monitoring purposes. Composing components is done by connecting the PF of one component to the PR of another component. The PR and PF also represent

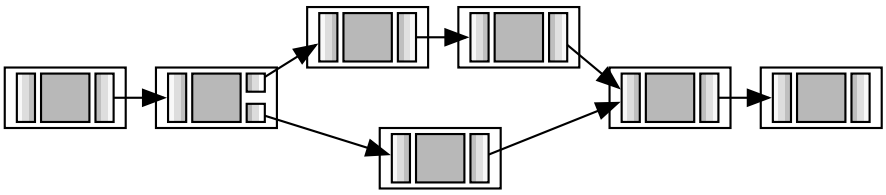


Figure 6.2: DiPS+ components are connected to each other in a pipeline

framework hooks that can be used to monitor what and how many DiPS+ packets go in and come out of a component.

6.2.3 DiPS+ connectors

DiPS+ builds on the *pipes and filters* architectural pattern [SG96, BMR+96]. Several DiPS+ components are composed into a component pipeline when connecting. A PFs can be connected with only one PR of another component in the pipeline. Components with multiple PFs can be connected to multiple components. This is shown in Figure 6.2. By consequence, multiple sub-pipelines are formed. DiPS+ packets are passed down the pipeline. After a component handled the packet, the packet is forwarded to the next component in the pipeline. A component that splits the pipeline in sub-pipelines selects a sub-pipeline based on a property of the packet. For example, depending on the destination IP address of an IP packet, the packet will be moved to the forwarding sub-pipeline or the local delivery sub-pipeline.

It is important to mention that components in the pipeline are not aware of the identity of other components in the pipeline. When components are finished processing a packet, they can only send the packet further down the pipeline; they cannot forward it to a particular component. This inter-independence of DiPS+ components allows single components to be reused easily and facilitates the runtime removal, addition and updating of components.

The only means to exchange information between components is by means of packets. Often the payload and protocol header are not sufficient to pass all information between packets. Therefore, packets also contain *meta information* that is used to exchange protocol implementation specific information between components. For example, a TCP packet's meta information will among others contain a reference to the Transport Control Block (TCB) of the connection it belongs to. A packet that was received from the network will receive a meta tag in the IP layer that denotes whether the packet is meant to be delivered locally or must be forwarded.

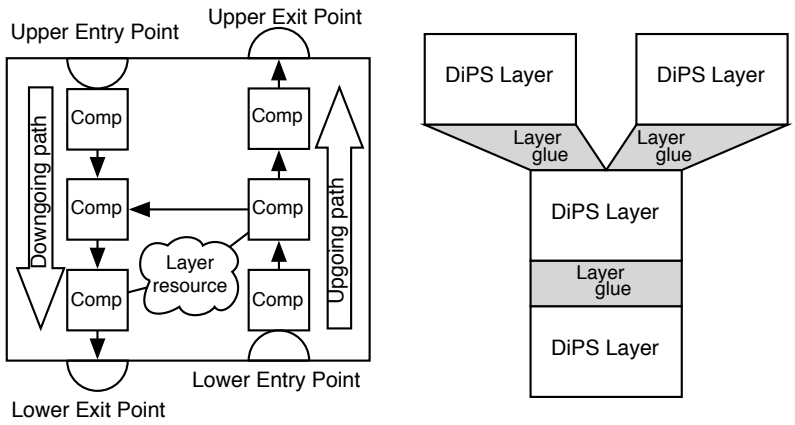


Figure 6.3: DiPS+ Layers. The left hand side of the figure shows that a DiPS+ layer internally possesses two component pipelines: a downgoing path, which goes from the upper entry point to the lower exit point and an upgoing path, which goes from the lower entry point to the upper exit point. This layer also contains one layer resource. The right hand side shows that DiPS+ layers can be stacked on top of each other and exist next to each other on the same OSI level. Layers between levels are interconnected by means of layer glue.

6.2.4 DiPS+ layers

An important property of protocol stacks is the concept of layering. The DiPS+ framework therefore also offers a layer abstraction to the protocol programmer. The DiPS+ layer is a container for exactly one protocol like Ethernet, IP or UDP. DiPS+ layers are stacked on top of each other to form a protocol stack. The internal structure of a DiPS+ layer is shown on the left side of Figure 6.3. A layer consists of two component pipelines: a downgoing path and an upgoing path. The downgoing path is meant for packets that need to be transmitted, the upgoing path is the pipeline that is used for packets that were received on a network interface. A layer also explicitly defines where DiPS+ packets enter (entry points) and leave (exit points) the layer. Every layer has two entry points and two exit points. The layer receives packets from a higher layer on the *upper entry point*. Packets arriving on the upper entry point travel down along the downgoing path and end up at the *lower exit point*, where they are handed off to a lower DiPS+ layer. Similarly, the layer receives packets from a layer lower in the stack on the lower entry point. These packets travel along the upgoing path to the upper exit point where they are forwarded to a layer higher in the protocol stack.

DiPS+ layers can coexist on the same OSI level or can be stacked on top of each

other, as shown in the right part of Figure 6.3. A DiPS+ layer exchanges packets with one or more layers above and below itself, but never with layers that coexist on the same OSI level. Before packets are exchanged between layers, the entry and exit points from the layers must be connected with each other using `Layerglue`. Such `Layerglue` connects the upper layer's lower exit point with the lower layer's upper entry point and the lower layer's upper exit point with upper layer's lower entry point. If more than one layer is to be connected to a particular layer, e.g. TCP and UDP are both located above the IP layer, this layer is configured with the necessary information so packets can be forwarded to the correct layer after processing. In protocol terminology, the layer is configured with the encapsulation types of the connected protocols. For example, if IPv4 receives a packet with encapsulation type 17, the packet is a UDP packet, if it receives a packet with encapsulation type 6, the packet is a TCP packet. The other way around, a UDP packet will be given to the IPv6 layer if its destination network layer address is an IPv6 or to the IPv4 layer if the address is an IPv4 address. In that case the encapsulation type is the address type.

6.2.5 DiPS+ layer resources

A DiPS+ layer can be configured with a number of layer resources. A layer resource contains state that components in a layer need to perform their task. Examples of layer resources are routing tables, ARP caches, hash tables that hold TCBs of open transport protocol connections, etc. Layer resources can be shared by multiple components inside a layer. The left hand side of Figure 6.3 shows a DiPS+ layer that is configured with a layer resource that is used by two components.

6.2.6 DiPS+ framework communication mechanisms

Normal DiPS+ framework operation is mainly forwarding network packets to the next component. This pure pipes and filters approach is usually not enough to realize a network protocol. Therefore two additional component communication mechanisms are available in DiPS+.

First, the meta information that is attached to a DiPS+ packet must be perceived as a *blackboard communication system* [BMR+96]. A component that processes a packet can attach other information than application data and protocol headers and leave it on the packet for other components to use. Components that use this information do not know what component provided that information. Blackboard communication is hence also an anonymous communication model and combines perfectly with the anonymous nature of DIPS+'s pipes and filters style.

Secondly, DiPS+ also offers event communication between components. The component pipeline's main purpose is handling data packets that must be transmitted or are received. Sometimes component functionality is also triggered by

external entities, such as timers or application layer sockets. For such purposes, the DiPS+ framework also includes an event sending mechanism. A component can fire events and can subscribe to receive particular events. If a component fires an event, The DiPS+ event mechanism will deliver it to the components that are interested. All event handling is done by the DiPS+ framework to maintain anonymous component communication. For example, a TCP resend timer will fire an event to the DiPS+ framework when the timer expires. The DiPS+ framework will deliver this event to the TCP resend component, which is subscribed to this timer event. Upon reception, the TCP resend component can resend the necessary network packets.

6.2.7 DiPS+ in dynamic networks

Using a protocol stack framework like DiPS+ can address the open networks challenge (Section 2.1.4). The component approach and the anonymous communication mechanisms used by DiPS+ facilitate easier protocol stack adaptations, which may be necessary in a dynamic network. If a mobile device migrates to a network where other communication protocols are used, it may be required to add new protocols to the stack to be able to communicate on that network. Some other protocols may have to be removed to reduce the protocol stack's memory footprint on the device.

A component composition tool [ŞMBV03, ŞVB02] for automatic composition of component based systems has been successfully applied on the DiPS+ component framework. This composition tools allow to build a DiPS+ protocol stack, i.e. a composition of DiPS+ components, from a set of high level protocol stack requirements. These requirements are determined by the application's communication needs and the network environment where this application will be deployed in. Every time a device is moved to a different network, the composition tool alter the DiPS+ component configuration.

To alter a running protocol stack, CuPS [JMMV02] can be used. CuPS is a DiPS+ extension that allows component replacement on an operational protocol stack. It coordinates the removal, addition and replacement of DiPS+ components at runtime without breaking protocol stack consistency. CuPS monitors DiPS+ components on their packet forwarder and packet receiver hooks (see Section 6.2.2) and replaces components when it is considered safe.

6.3 CAS design in DiPS+

This section discusses the design of the CAS in the DiPS+ framework. Central to the CAS design is the session data structure that is maintained by the CAS. Therefore, before discussing the actual design, this data structure is explained. The design of the CAS in DiPS+ mainly encompasses the design of the CAS

layer in DiPS+, described in Section 6.3.2, and the session state machine which is described in Section 6.3.3. Additionally, Sections 6.3.4 and 6.3.5 respectively describe how the CAS layer interacts with the transport layer to establish transport protocol connections and how the CAS layer provides application feedback.

6.3.1 Maintaining session data

The CAS maintains a data structure that contains all the data belonging to one session. In the remainder of this text, this data structure is also referred to as the *session control block* (SCB), because it can be compared with the transport control block (TCB) that is typically used in TCP implementations to maintain information about a TCP connection. The SCB encompasses session characteristics such as the session type (reliable/unreliable, data stream/datagram), the session identifier, information about the associated transport connection, listen connection and also operational data structures such as send buffers, receive buffers and session state information. All this data is located in a container which is a Java class called `Session`.

A DiPS+ packet that enters the CAS layer must always belong to a particular session. If not, it will be discarded. A CAS packet always contains a reference to the `Session` data structure. This reference is attached to the packet as meta data (see Section 6.2.1) as soon as the packet enters the CAS layer. This way, when a component in the CAS layer must process a packet, it can access all information of the pertaining session though the packet's meta data.

6.3.2 The CAS layer design

Figure 6.4 shows the DiPS+ design of the CAS layer. On the left hand side the downgoing path can be seen, while the upgoing path is depicted on the right. The layer possesses three layer resources: the `Connection Management` layer resource which is used for managing transport protocol connections, the `Session Manager`, which maintains all sessions and the `Session State Machine` which implements the CAS's state transition diagram.

This section shortly discusses the task of each component in the CAS layer. First, the responsibilities of the `Session Manager` and `Connection Management` layer resources are explained. Then the component anatomy of the layer is discussed. Due to its complexity, the `Session State Machine` is explained separately in Section 6.3.3.

The goal of the `Session Manager` is to maintain SCBs and monitor network changes. For every session, the associated transport protocol connection (if any) and associated listening session (if any) are recorded. This is necessary to associate packets received from the network to the correct session. The `Session Manager` is also subscribed to DiPS+ events that indicate changes in network status. If the `Session Manager` receives such events, it may (unanticipatedly)

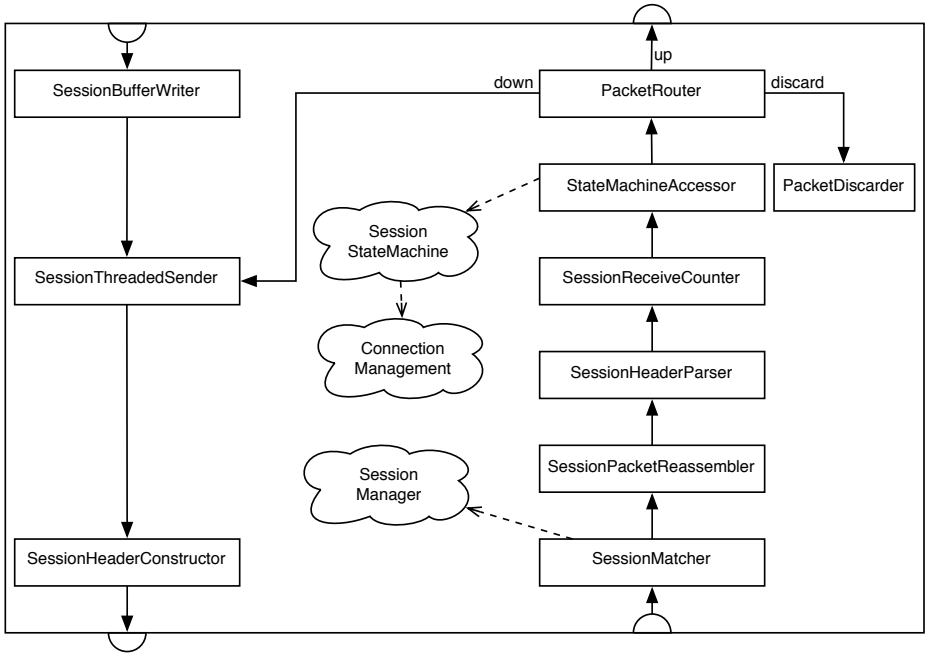


Figure 6.4: DiPS+ design of the CAS

suspend or resume sessions by sending a DiPS+ suspension or resumption event to the `Session State Machine` layer resource for every affected session. The `Connection Establishment` layer resource is responsible for managing transport protocol connections. It is used when transport connections must be established and torn down. It also detects the creation of new incoming connections or the abortion of connections. In the latter case, it sends a DiPS+ event to the `Session State Machine` so it can take the appropriate measures when such events happen.

Data that must be transmitted must first be written on the socket. The socket puts all the received data from the application in a DiPS+ packet and attaches the associated SCB as meta information. The packet is then given to the upper entry point of the CAS layer where it follows the downgoing path of the CAS layer. The downgoing path consists of three DiPS+ components. The first component is the `SessionBufferWriter` which implements the double buffer strategy. All data received from the application is put in a circular buffer if a reliable data transfer service is used. Otherwise, no system resources are wasted on buffering. The amount of data that is buffered depends on the size of the send and receive buffers of the used transport protocol (see Section 6.5.1.4). The task of the `SessionThreadedSender`, the second component in the downgoing path, is of a

non-functional nature. The component maintains one packet queue for every CAS session and allocates a thread to each queue. Packets that arrive in the component after being processed by the `SessionBufferWriter` are added to the queue. When packets are in the queue, the thread for the associated session is activated and then sends the packets further down the protocol stack. The last component in the downgoing path is the `SessionHeaderConstructor` which creates the session header (see Section 4.3) and attaches it to the packet that is being transmitted. Subsequently, the packet leaves the CAS layer through the lower exit point.

The upgoing path is more complex as it consists of seven DiPS+ components. When packets arrive in the session layer at the lower entry point, they are first guided towards the `SessionMatcher`. The task of the `SessionMatcher` is to find the session the received packet belongs to. The meta data of the received packet holds information about the transport protocol connection on which the packet was received. The SCB can hence be retrieved by consulting the `Session Manager`, which keeps a mapping between transport connections and the belonging session. The `SessionMatcher` subsequently adds the session data structure to the packets meta data. The next component in the upgoing path is the `SessionPacketReassembler`. It is possible that transport layer protocols break CAS packets up into multiple transport protocol packets or combine CAS packets into one transport protocol packet. This problem occurs mainly when transport protocols offer data stream services and reorganize segments as part of their protocol operation. An example of such a protocol is TCP. The `SessionPacketReassembler` reorganizes packets that come from the lower layer into CAS packets that start with the CAS header. The reorganized packets are then handed to the `SessionHeaderParser` which creates a CAS Header object and attaches it to the packet's meta information. The `SessionReceiveCounter` subsequently counts the data bytes in the received packet adds it to the total amount of received data. This number is used when the session is resuming after an unanticipated suspension. The packet is subsequently guided through the `StateMachineAccessor`. This component hands the received packet over to the state machine, which is described in Section 6.3.3. The state machine performs a number of protocol checks on the received packet, performs state transitions and produces reply packets if necessary. All packets that come from the `StateMachineAccessor`, the original packet and new packets, are forwarded to the `PacketRouter`. The `PacketRouter` sends packets that contain application data to the socket (the up arrow). Reply packets are sent down the downgoing path (the down arrow) to the `SessionThreadedSender`. The remaining packets, such as session establishment requests that contain no application data, are destroyed by the `PacketDiscarder`.

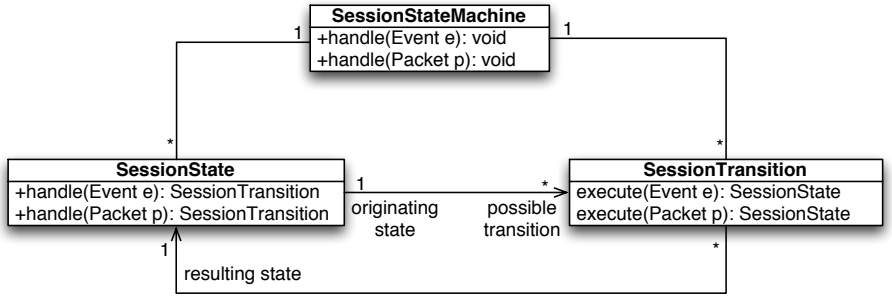


Figure 6.5: UML Diagram of the CAS state machine in DiPS+

6.3.3 CAS state machine design

The CAS adheres to a protocol that is defined by a state transition diagram (see Section 4.4). Every session is in a particular state at any moment in time. Depending on the state of a session, the arrival of a packet or the occurrence of a network event may trigger a particular action and change the session's state.

The CAS state transition diagram is realized as a state machine in the DiPS+ implementation. The UML class diagram of the state machine is depicted in Figure 6.5. The `SessionStateMachine` class is the CAS layer resource shown in Figure 6.4. The packets that are traveling along the upgoing path are handed over to the `SessionStateMachine` by the `StateMachineAccessor`. To realize this, the StateMachineAccessor calls the `handle(Packet p)` method on the layer resource. Next to packets, the state machine also responds to DiPS+ events. The state machine is interested in events that signal a.o. session establishment, session suspension and session resumption requests and timer expirations so session management requests can be resent. To this end, the DiPS+ event mechanism calls the `handle(Event e)` method on the `SessionStateMachine`.

The UML diagram shows that the state machine consists of a number of states and a number of state transitions. Every state is associated with a number of transitions that can be triggered from that state. Transitions are associated with their originating state and also have a resulting state, which is the state the session will be in after that transition is executed.

`SessionState`s and `SessionTransition`s contain all the CAS's protocol logic. Every time the session state machine receives a packet or an event, this packet or event is handed over to the current state of the session. The `handle()` methods of a `SessionState` object perform a number of checks on the received packet or event to decide what transition must be executed in response. The selected transition is transfered back to the `SessionStateMachine` which invokes the received transition by calling the `execute()` method on the transition. This method contains the

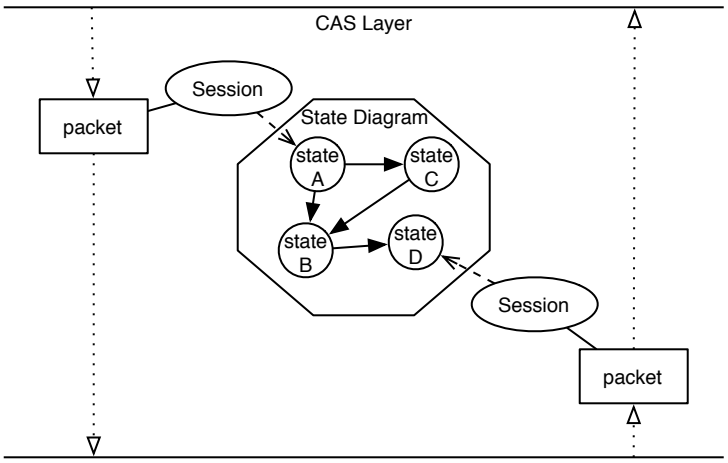


Figure 6.6: DiPS+ design of the CAS state machine

instructions needed to handle session management requests, create reply packets and error messages, etc. As a result, the `SessionTransition` hands back the resulting `SessionState` to the `SessionStateMachine`. This state is the session's new state.

It should be clear that the entire session state machine is a large object structure. Currently, the DiPS+ implementation consists of 15 states and 24 state transitions, some of which are instantiated more than once. To reduce the memory requirements, the DiPS+ implementation only creates one instance of the state machine, instead of creating a new state machine structure for every session. This is shown in Figure 6.6. All session specific data is maintained in the session control block (Section 6.3.1) while the state machine only contains the CAS protocol. Every session data structure maintains a reference to one state object in the state machine. This is shown in the figure by a dashed arrow. This reference always points to the current state of a particular session. When a packet or event is received for a particular session, the `SessionStatemachine` can retrieve the session's current state from the session data structure. The `SessionStateMachine` subsequently calls the `handle()` method on the retrieved state and the `execute()` method on the resulting transition. All session specific data required to execute these methods is retrieved from the session data structure.

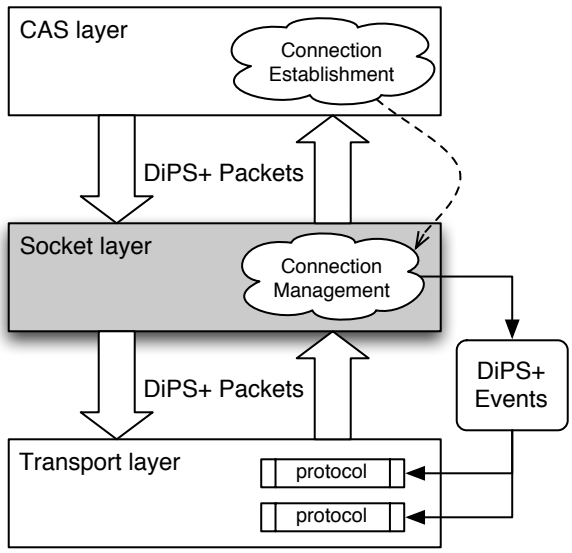


Figure 6.7: Using the socket layer in the DiPS+ framework

6.3.4 Transport layer interaction

To establish and terminate a transport protocol connection (TPC), the CAS layer interacts with the transport layer using a special *socket layer*. Because the CAS is realized as a DiPS+ layer, it can in principle only exchange packets with the lower layer by means of its lower exit point and lower entry point. Establishing and terminating a TPC requires more complex interaction with the transport layer than simple packet exchange. The socket layer offers functionality to the higher protocol stack layers that is similar to the functionality of an application layer socket (ALS).

Figure 6.7 shows how the CAS layer interacts with the socket layer and how the socket layer interacts with transport layer protocols. The socket layer contains a `Connection Management` layer resource that contains a number of methods to establish and terminate sessions. If that layer resource receives a call to establish a transport connection, it contacts the transport protocol to set up the connection. Contacting the transport protocol is done by sending transport protocol specific DiPS+ events, which are also used by the transport protocol's ALSs. The transport protocol must hence not be adapted to be used with the socket layer.

The main difference between using ALSs and the socket layer is that an ALS represents only one transport connection, where the socket layer is responsible for all transport connections it created. To identify a single TPC, a unique token

is generated for every transport protocol connection created by the socket layer. The higher protocol stack layer that requested the TPC obtains a reference to this token if the TPC was established successfully. If a higher layer wishes to send a packet on that TPC, it must attach the token reference to the packet's meta information. The socket layer checks if the attached token refers to a valid TPC. If it does, the packet is allowed to travel further down to the transport layer, otherwise the packet is dropped. If the socket layer receives a packet from the transport layer, it also adds the token reference to the packet's meta information before it is forwarded to the upgoing path, so higher layers can differentiate packets that were received on different connections.

6.3.5 Application interaction

Application interaction is easily realized in DiPS+ by using the DiPS+ event system. An event is generated as part of a state transition's `execute()` method. The transitions that generate application feedback events will typically be the transitions that move the session's state machine to the suspended or the established state.

If an application subscribes to an event on its session socket (see Section 4.6), the socket registers itself with the DiPS+ event system as an event receiver for that particular event. If for that session a state transition is executed that generates the DiPS+ event, the socket will consequently receive the event and forward the event to the application by calling the `statusChanged()` method on the application's `FeedbackReceiver` (shown in Listing 4.3 in Chapter 4). Note that the transition always fires the event, even if the application did not subscribe to receive the event. If the application did not subscribe, the socket has not registered itself with the DiPS+ event system and will not be notified of the event.

6.4 AMS design in DiPS+

The AMS is realized as an optional protocol stack add-on in DiPS+. It can be added to the protocol stack if desired. The next section explains how address reduction is realized in DiPS+ by means of stack address managers and layer address managers. Subsequently, the way the AMS is plugged in DiPS+ is discussed.

6.4.1 Reduction: stack and layer address managers

Address reduction functionality is divided among *layer address managers*, which are all coordinated by the *stack address manager*. Figure 6.8 shows how both manager types relate to the protocol stack. The left hand side of the figure depicts the protocol stack's communication protocols. The right hand side shows the address management infrastructure. The stack address manager (SAM) is the

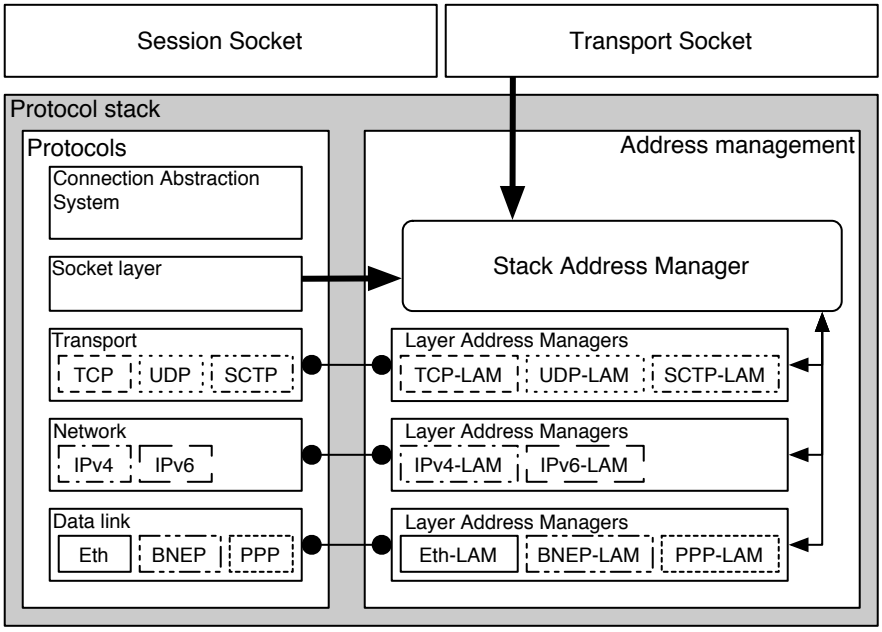


Figure 6.8: Address management in the DiPS+ framework

main responsible for reducing generic addresses (see Sections 5.3.1 and 5.3.2). The socket layer and protocol independent transport protocol sockets must direct their address reduction requests to the SAM, as is indicated on the figure by the thick arrows.

The SAM controls a number of layer address managers (LAM). For every communication protocol in the stack, one LAM is present in the address management subsystem. In the figure, a transport protocol and corresponding LAM are depicted as 2 boxes that have the same line style. LAMs are also divided into OSI levels corresponding to the level the associated protocol belongs to.

If the SAM receives a request to reduce a generic address, it first performs protocol reduction. Protocol reduction is the reason why address managers are ordered per OSI level because it reflects what protocols are available on every OSI level. The address types on every OSI level in the generic address are matched against the available LAMs in the corresponding OSI level. For every OSI level one address is selected for which the protocols (and hence LAMs) are available in the stack.

After protocol reduction, the SAM sends every remaining layer address in the generic address to the corresponding LAM for address reduction. Layer address are normally reduced in a top-down way: the transport layer address is reduced first, then the network layer address. The data-link address is typically not reduced when a network layer protocol is used because the selection of a network layer address implies the selection of a data-link address. The SAM coordinates layer address reduction and can override or affect the reduction decision of a LAM. For example, the policy of an IP LAM can be to select the IP address that corresponds with the fastest interface. The SAM can instead instruct the IP LAM to choose the interface that is cheapest to use because the user does not wish to send business critical information.

6.4.2 Plugging AMS in the DiPS+ stack

When a packet is sent down the pipeline, the socket attaches the required addressing information to the packet's meta information[1]. If the AMS is used, the reduced generic address is attached to every data packet instead of the protocol specific transport protocol port and network layer address. For received packets, the generic address is built gradually as the packet travels up the different OSI levels.

The protocol stack must be altered to use generic addresses instead. Due to the component-oriented nature of the DiPS+ framework, changes to the protocol stack are isolated to the components that access a packet's addressing information. These components must be replaced with components that know how to access the

[1] In the case the CAS is used, the socket attaches session information to the packet. The CAS will then attach the transport and network layer address information that is valid at that point.

generic address data structure. The functionality of the DiPS+ components and the component composition does not change, only the way by which addressing information is retrieved from a packet. Because every component has no explicit dependencies towards other components (See Sections 6.2.2 and 6.2.3), it is easy to replace components with their AMS aware equivalents.

6.5 Implementation and protocol evaluation

This section discusses the consequences of using the proposed session layer in the protocol stack. First, overhead of the solution is discussed in Section 6.5.1. Section 6.5.2 discusses the speed by which disconnection can be detected with a session layer solution. Finally, Section 6.5.3 discusses the behavior of transport protocols during immediate handover.

6.5.1 CAS overhead

A new protocol layer in the protocol stack introduces extra overhead. First, the processing overhead of the DiPS+ implementation is measured in Section 6.5.1.1 by comparing DiPS+ protocol stacks with and without the CAS/AMS solution. In Section 6.5.1.2 the overhead of the three way handshake protocols is evaluated. The overhead of the CAS header is investigated in Section 6.5.1.3. Memory consumption of a session control block is discussed in Section 6.5.1.4. Finally, Section 6.5.1.5 examines the code size of the implementation.

6.5.1.1 Processing overhead

Goal. This section discusses the relative overhead of the CAS and AMS by comparing a CAS/AMS enabled DiPS+ protocol stack with a protocol stack that does not contain the CAS layer and AMS extension. Because CAS behavior is different depending on the type of transport protocols used, processing overhead must be evaluated both for unreliable protocols and reliable protocols. More specifically, the CAS must retain data that can still get lost in transit if a reliable transport protocol is used, where this is not required for unreliable transport protocols. Currently the most popular transport for unreliable and reliable transport protocols are respectively UDP and TCP, which are both implemented in DiPS+. The tests compare a CAS/SocketLayer/**UDP**/IPv4 stack with an **UDP**/IPv4 stack and a CAS/SocketLayer/**TCP**/IPv4 stack with a **TCP**/IPv4 stack.

Test setup. The tested stacks are run on Intel Pentium II 400MHz machines with 256 Mbyte RAM running Debian Linux with kernel version 2.4.25. The used Java virtual Machine is the Sun JVM version 1.4.2_01. The machines are connected with each other using a 100 Mbit ethernet network. Every machine consists of a

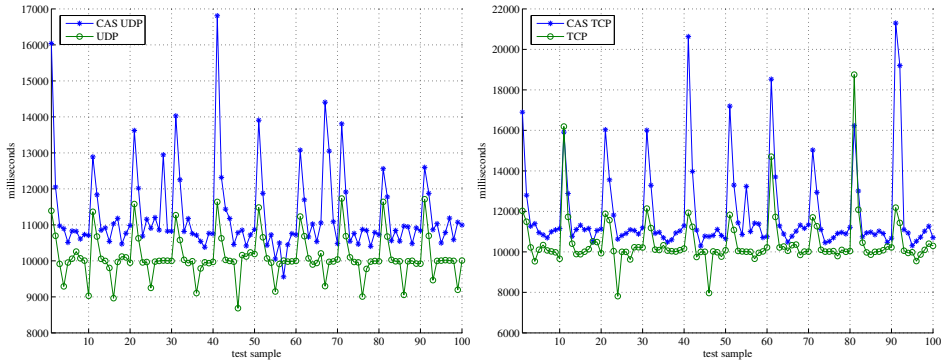


Figure 6.9: Timing results for 10 testruns for UDP (left hand side) and TCP (right hand side). The stars on the blue lines show the timing results for a DiPS+ stack that contains the CAS layer, the circles on the green lines show the results for a stack that does not contain the CAS layer.

number of *ethertap* devices. An application that opens an ethertap device file can read and write ethernet packets directly on the wire. This allows a DiPS+ stack to generate Ethernet frames and output them directly to the physical layer and import packets directly from the physical layer so they can be processed by DiPS+ instead of the operating system's stack.

A test consists of sending a large number of integers back and forth (two-way communication) between two test machines. When a machine receives an integer, it sends an integer back and waits until it receives the next integer. For each test, the time between sending the first integer and receiving the last integer is measured. A large number of integers is required because the granularity of Java's clock is a millisecond. A test must last long enough for Java to be able to measure the test duration. We therefore decided to send 1000 integers back and forth. Session and/or connection establishment is not measured because the intent of this test is to measure processing overhead during normal, two-way communication.

The testresults discussed here are generated by running 10 test runs. A test run consists of building the required DiPS+ stack and repeating ten tests (ten times sending 1000 integers) on that stack. Ten test runs hence yield 100 timings. Running 100 tests on the same stack resulted in unexpected test results. The duration of a test increased linearly after running approximately 30 to 40 normal tests. We suspect increasing garbage collector activity after a larger number of test runs. We therefore chose to only run 10 tests on one protocol stack.

Test results. The timings for the tested stacks are shown in Figure 6.9. The left hand figure shows the test results for both the CAS/SocketLayer/UDP/IPv4

and the UDP/IPv4 stack, the right hand figure shows the results for the CAS/-SocketLayer/TCP/IPv4 and TCP/IPv4 stack. In both figures, the stars on the blue line show the timings of a stack with CAS. The circles on the green line are the timings of the stacks without CAS.

Remarkable about this figure are the peak timing values that mainly appear in the beginning of a test run. Tests samples 1, 11, 21, 31, ... show peak timing values. Only after two tests, the timing values in a test run stabilize. The protocol stack does not function optimally right after it was built and needs a warmup phase. The first two tests in a testrun are considered to be the *warmup phase* of the protocol stack. Therefore, for every testrun the timing values of the first two tests are considered to be noise and are removed. The 80 remaining test samples are depicted in Figure 6.10.

Figure 6.10 shows the results after the warming up tests are removed. Note that there are still extreme high and low timing values in both the TCP and UDP case. The occurrence of the high values is difficult to explain as they occur randomly and do not appear in every test run. The most plausible explanation is the Java garbage collector that is cleaning up unused memory when it deems necessary to do so. The low timings occur more frequently in the UDP case. These can be explained by the use of the ethertap device. The ethertap device is accessible by means of a random access file that is used for both reading and writing. Reading from the tap device happens in a separate Java thread. The speed and frequency of reading from the tap device depends on the Java thread scheduling policy. If the tap device's thread is favored, the test results will be better because the protocol stack receives data from the network faster. DiPS+'s TCP implementation uses more java threads than the UDP implementation which explains why there are more low timing values in the UDP case; the chance that the tap device thread is scheduled in favor of other threads is larger in the UDP case.

The graph at the top in Figure 6.10 depicts the timing results in case UDP is used as a transport protocol. The mean time to run a test with CAS/AMS is 10861 ms (standard deviation 600.44 ms). A test without CAS/AMS support takes on average 9871 ms (standard deviation 333.87 ms). CAS overhead when using UDP as a transport protocol is then about 990 ms or 10.02%. In the case of TCP, depicted at the bottom in Figure 6.10, the mean time to run a test with CAS/AMS is 10963 ms (standard deviation 382.76 ms). Without CAS/AMS it takes 10000 ms (standard deviation 392.96 ms) to complete a test. In case of TCP the overhead amounts to 963 ms or 9.64% overhead.

Note that the standard deviation of the CAS/SocketLayer/UDP/IPv4 timings is substantially larger than the standard deviation of the other tests. This is a consequence of the peak timing results for the UDP case. Because the median of a data set is not very sensible to extreme values, using the median instead of the arithmetic mean will give more correct results. Figure 6.11 shows the median values and gives an indication of the dispersion of the measured time values.

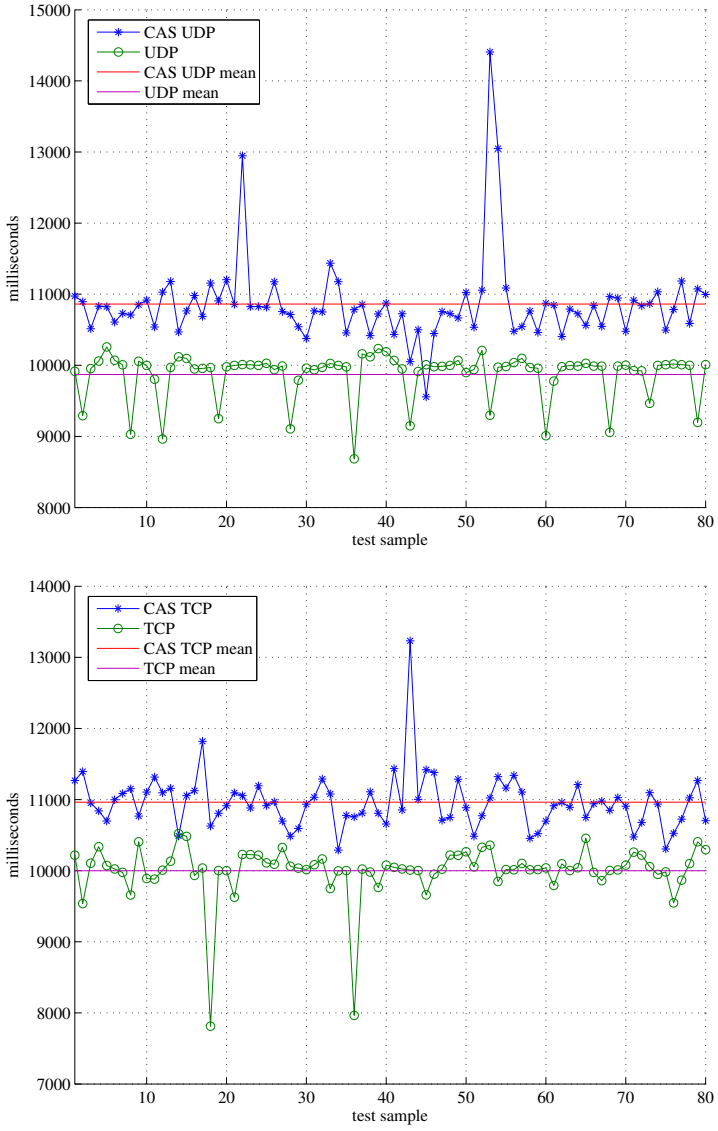


Figure 6.10: Timing results for 10 testruns for UDP (top figure) and TCP (bottom figure). The timing values depicted are the same as in Figure 6.9 without the timing values from the warmup phase. The red line depicts the mean time to run a test with CAS/AMS, the purple line depicts the mean time to run a test without CAS/AMS.

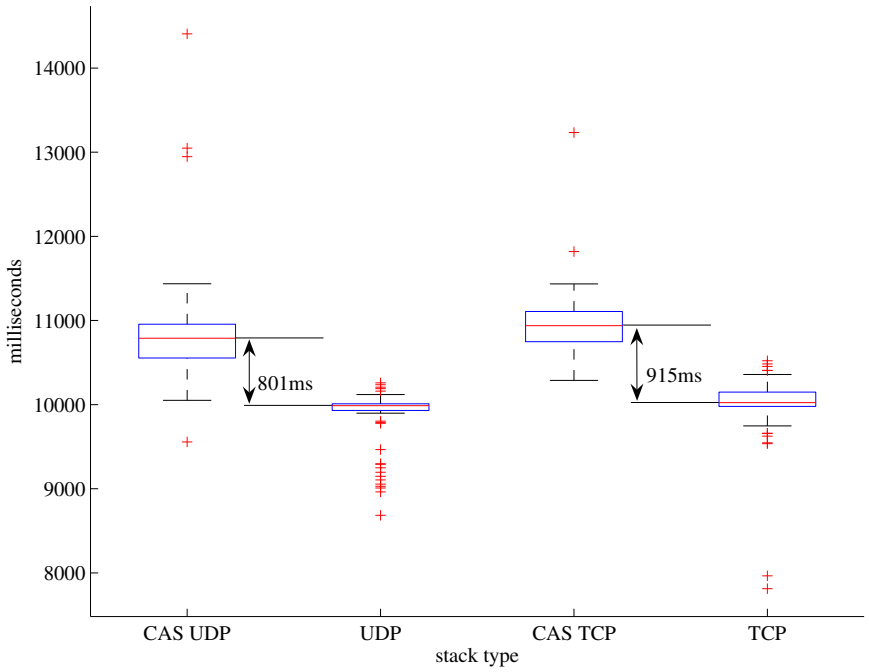


Figure 6.11: Boxplot of the timing results for every protocol stack type. The figure indicates the medians for every stack tested, and also indicates the dispersion of the time values measured by means of upper and lower quartiles and whiskers that extend to maximally 1.5 times the interquartile range. The plus signs indicate values that fall outside the range of the whiskers.

When using the median instead of the arithmetic mean, the relative overhead of CAS/AMS in case of UDP is 8.03%. In case of TCP, the relative overhead is 9.13%.

Figure 6.11 also shows the absolute difference in milliseconds between the medians per transport protocol. For the UDP test, running with CAS/AMS supports results in a performance penalty of 801 ms. For TCP, this penalty is 915 ms. The CAS implementation used for TCP is the same as the implementation used for UDP with buffering functionality turned on. One could therefore say that buffering added 114 milliseconds to the execution time of the CAS. In the tests, the overhead of buffering in the CAS hence amounts to 14.16%.

Conclusion. We have tested our CAS prototype implementation in DiPS+ and have evaluated it using both a connectionless unreliable transport protocol and a

connection-oriented, reliable transport protocol. A protocol stack that uses the CAS/AMS requires less than 10% additional computing resources, even when a reliable transport protocol is used and double buffering is necessary. The overhead of double buffering amounts to 15% of total CAS processing overhead.

We believe that a 10% overhead in a stack with normal operation is an acceptable considering its added value of endpoint mobility, disconnected operation and application feedback. However, we are aware that these numbers are not totally conclusive, but rather give an indication of the consumed resources. More conclusive results require more tests, using other types of traffic like bulk transfer (file transfer) and asymmetric two-way communication (interactive login sessions send small amounts of data and may receive large amounts of data in reply). Also, implementing and testing the CAS in a more widely used protocol stack, like the linux or BSD protocol stack, will yield different results because such protocol stacks are more tuned towards performance than flexibility. Such protocol stacks do not offer the runtime flexibility that DiPS+ offers and that may be needed in dynamic networks. Nevertheless, the overhead should remain small since the CAS would have to be implemented using the same design principles (e.g. single copy principle of data in the protocol stack) and data structures that are tuned towards performance.

6.5.1.2 Protocol overhead

Before an application using the CAS can start communication it must establish a session using a three-way handshake protocol, independent of the type of transport protocol the application uses. Session establishment hence introduces an extra delay before actual communication can start, which may affect an application's response time. This delay's lower bound is the time it takes to transmit the protocol messages plus the protocol's processing time. Exchanging 3 protocol messages ideally adds up to 1.5 times the round trip time, if no protocol messages are lost on an unreliable communication infrastructure. The required processing time depends on machine speed and the cryptographic measures that may be used to secure session negotiation. Similar delays must be taken into account when resuming a session.

Delays for anticipated session suspension and teardown of a session are less important for the application. In both cases the application (temporarily) stops communicating. The time required to suspend or tear down a session does not affect the responsiveness of the application.

This delay points out that establishing a CAS session is only feasible for sessions that have a longer life span. For short transactions, the benefits of the session layer do not always outweigh the amount of protocol overhead. The chance that the connection is lost during a short transaction is small, and when it does occur it is often faster, and not even considered a burden, to repeat the transactions. For example, if network connection is dropped because of a handover while loading a

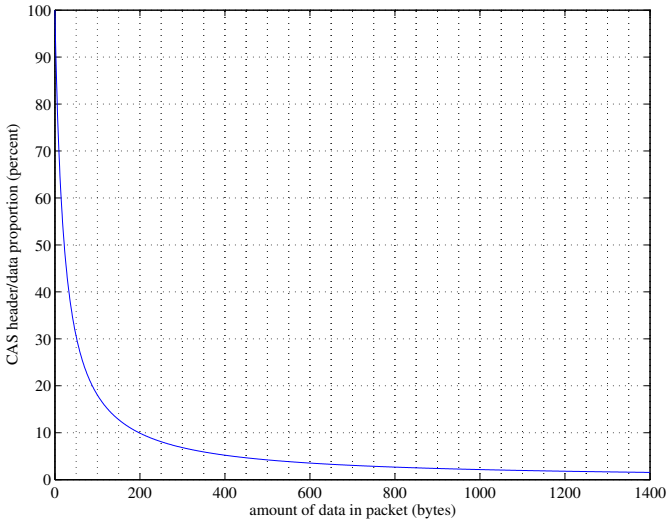


Figure 6.12: Proportional CAS header size with respect to the amount of data in the CAS packer.

webpage, a user typically is not concerned having to reload the page, as long as it does not happen too often. Another example is getting one's mail from a POP3 server. Even executing a money transfer may be interrupted because application layer atomic transaction models are used to provide reliable bank transactions. The use of the CAS is more interesting to use when transferring files, watching movie streams, chat sessions, which typically use a single transport protocol connection which is kept alive for a longer time span. Only then the overhead of session establishment becomes negligible with respect to the benefits the protocol offers in such application environments.

6.5.1.3 Header overhead

The CAS header size is 22 bytes and consumes network bandwidth each time it is sent. The impact of this overhead depends on the amount of data that is sent for every header. Figure 6.12 shows how the amount of relative overhead of CAS header decreases if the amount of data sent per CAS header increases. In the DiPS+ prototype, the application determines how much data is sent per CAS header. Every data buffer that is passed by a call that the application makes on the CAS socket (datagram service) or CAS output stream (stream service) is converted into a single CAS segment.

The amount of data that is given to the CAS protocol stack with each socket call must not be too big. Offering too much data to the protocol stack in one call

has two disadvantages. First, one must try to prevent fragmentation of network packets. Especially in the case of datagram transport protocols, the amount of application data and the protocol headers should fit in a packet whose size is determined by the maximum transmission unit (MTU) of the used network hardware and network route. If a larger amount than the MTU is transmitted, the network layer will have to fragment and reassemble the segment during transmission, which also decreases the chance that the datagram packet will be successfully delivered. In case of data stream transport protocols, the transport protocol is typically responsible for dividing the data stream into segments. CAS segments are concatenated in a stream and re-segmented by the transport protocol. Secondly, sending a large amount of data in one output call affects the response time of the CAS protocol, because it determines the spacing between the headers. An unlikely but illustrative example is an application that presents a chunk of 64kB to the CAS. The CAS creates a segment, consisting of a CAS header followed by 64kB, and is given to the transport layer for further processing. The next CAS header will be sent after those 64kB have been processed. On a device using a slow access connection, transmitting 64kB can take a large amount of time. If the header that follows contains a suspension request, for example, it may take a while to complete the suspension protocol because those 64kB must first be processed. CAS protocol response time hence decreases if applications send large chunks of data in one call. Smaller chunks of data mean a larger bandwidth penalty, because header/data ratio increases, but also mean quicker CAS protocol response time.

6.5.1.4 Memory usage

The CAS maintains a Session Control Block (SCB) for every session, similar to the TCP's Transport Control Block (see Section 6.3.1). The largest part of an SCB consists of the buffered application data that is maintained to ensure reliable transport in case of unanticipated suspension. If unreliable transport protocols are used, these suspend buffers are not allocated. If transport protocol state can be exported and imported (See Section 4.4.2.3), the buffers are only allocated when the session becomes suspended. When the session is resumed, the occupied memory can be freed. In case double buffering is used, the buffers are allocated when the session is being established and remain allocated for the entire lifetime of the session.

The size of the SCB is different for every session because the required suspend buffer size depends on the amount of data that can get lost in the transport protocol when network access is lost. For example, in the case of TCP, the allocated buffer size even varies between different transport connections. To maximize throughput, the window sizes of a TCP connection can be calculated by means of the bandwidth-delay [Ste00]. Because the bandwidth and delay are typically different between different peers, the window sizes will be different for each transport connection. The size of the SCB will be different for each session, and can even

vary during the lifetime of a session, depending on the TPC that is used at that moment.

In the DiPS+ prototype of the CAS, another constraint is added to the suspend buffer size. The buffer size is determined also by the largest CAS packet that can be sent. In the DiPS+ prototype, before data is put in the CAS receive buffer, it must be completely reassembled in the `SessionPacketReassembler` (see Section 6.3.2). The `SessionPacketReassember` discards packets that have not arrived completely when a session suspends. The amount of received data hence always coincides with the end of a CAS packet. Consequently, resending data after resumption always happens at the granularity of a CAS packet. If the size of a CAS packet exceeds the size of the transport protocol buffers, the suspend buffers should be the size of the largest possible CAS packet. An SCB of an active session in DiPS+ is 200kB large if CAS packets would be 64kB. The session suspend buffer is 64kB if the transport protocol buffers are not larger than 64kB. The session reassembly buffer, which is also maintained in the SCB in the DiPS+ implementation is 128kB[2]. The remainder 8kB is occupied by counters, timer values, generic addresses, etc. A suspended SCB would take 72kB because the reassembly buffers are not maintained during suspension.

6.5.1.5 Code size

At the time of writing, the CAS implementation in DiPS+ consists of 6912 lines of code in 118 classes. This amounts to 243kB of compiled Java bytecode. The DiPS+ implementation of TCP and UDP are respectively 371kB and 24kB. The DiPS+ TCP implementation is a fairly straightforward implementation containing the fast retransmit and fast recovery algorithms (Reno TCP), without extra acknowledgment strategies such as SACK[MMFR96].

The DiPS+ CAS implementation is designed in a similar way as the DiPS+ TCP implementation. The same state machine architecture is used, the responsibilities of the DiPS+ upgoing and downgoing paths of both protocols are similar (parsing or creating the header, finding the SCB for a packet, checking the session's state machine,...), both implementations use the same buffering techniques. In a way, the CAS is a simpler protocol than TCP, however. The CAS does not need to implement a sliding window protocol or realize an acknowledgement strategy. This could explain why the CAS implementation is smaller than the TCP implementation, despite the similarities in design. Although it is difficult to compare a DiPS+ protocol implementation with an implementation of that protocol in another protocol stack framework, it should be possible to create a CAS implementation in a protocol stack framework other than DiPS+ that has a smaller memory footprint than the framework's TCP implementation.

[2]The packet reassembly buffers are chosen to be twice the size of a CAS packet, so the packet reassembler can already start processing the next packet before the previous packet is sent further down the pipeline.

A small memory footprint is typically a requirement when realizing a protocol stack for small, embedded devices with limited memory and computing resources. There exist TCP/IP implementations that are only few kilobytes large. For example, lwIP and uIP [Dun03] implement a complete TCP/IP protocol stack for systems with very limited memory. lwIP code size varies from 14kB to 21kB, uIP is only 5kB.

To obtain such small code sizes, protocol implementations on embedded devices do not always implement the entire RFC, or limit the possibilities of the protocol. An implementation with a small memory footprint is often favored over a completely RFC compliant implementation as long as it is still able to communicate which RFC compliant implementations. For example, uIP does not provide a soft error reporting mechanism, does not implement a sliding window protocol and does not contain any congestion control mechanism, but it is still able to communicate with RFC compliant TCP implementations.

To reduce CAS code size, one could also eliminate CAS functionality to reduce its memory footprint. For example, one may choose to remove support for anticipated suspension or remove application feedback functionality if it is not needed.

6.5.2 Detecting disconnection

It is desired that unanticipated disconnection is detected as fast as possible during data exchange. The faster disconnection is detected, the faster the application can be informed and adapt its behavior. Detection time and even the possibility to detect disconnection is typically different for the participating peers in the session. Additionally, disconnection detection time also depends on the used type of transport protocol. During the discussion, it will be assumed that only one party participating in the session is responsible for the disconnection. This party will be referred to as the *client*. The peer communication partner will be referred to as the *server*.

The client normally detects disconnection immediately. If a device actively disconnects from the network, this is usually detected by the hardware. The protocol stack is notified by such an event so it can respond accordingly. For example, if the network cable is detached from the device, this is detected by the network interface card driver and by consequence, the protocol stack removes the routing table entries that are no longer reachable. If the client's protocol stack contains the CAS protocol, the CAS will suspend all sessions that depended on that interface.

The server is usually not immediately aware of the disconnection of the client. Because the server side CAS device is not detached from the network, hardware detection is not possible. Detection on the server side CAS relies mainly on the errors generated by transport layer protocols that are caused by a client CAS that is no longer communicating. These errors, if any, depend on the type of transport

protocol that is used: connection-oriented transport protocols or connectionless protocols. The next two sections discuss server side disconnection when using these two transport protocol types, using TCP and UDP respectively as example protocols.

6.5.2.1 Disconnection detection with connection-oriented protocols

Connection-oriented protocols generate an error in case of an irregular event. A transport protocol monitors a transport connection by means of a number of timers that expire if a particular event does not happen. If such a timer expires, the transport protocol responds with the appropriate action. For example, TCP forces the retransmission of a data segment if it does not receive an acknowledgement for that segment in time. If that particular event does not happen after the protocol tried to resolve the situation multiple times, the protocol assumes that the connection is no longer valid and an error is generated which must be addressed by higher layers in the protocol stack.

The speed by which higher layers, and more importantly the CAS layer, are informed about disconnection depends on the speed by which timers in the transport protocol expire and on the number of times this timer may expire before it is considered an error. A TCP implementation commonly resends a lost segment twelve times before it concludes that the peer has disappeared, sends a reset signal and aborts the connection[Ste00]. This entire process lasts about nine minutes. A compromise can be made between disconnection detection speed and TCP reliability. Disconnection detection can be sped up by reducing the amount of resends and the amount of time between resends. The downside is the reduction in reliability offered by TCP.

Ideally, transport protocol timeout behavior should be tunable to the network situation and the type of application. Nine minutes between the application sending data and the detection of the error by the CAS is a considerable long time for interactive applications, while it is acceptable for download operations running in the background. Depending on the network situation, retransmission behavior can also be altered. For example, TCP perceives packet loss as congestion. In wireless networks this is not necessarily the case and therefore the retransmission policy in TCP should be adapted. Such Improvements for TCP in wireless networks have already been investigated[BSAK95]. It must be noted that such optimizations are still proper to the transport protocol and are not handled by the CAS. Disconnection detection time hence still depends on the timeout policies of the transport protocols.

6.5.2.2 Disconnection detection with connectionless protocols

Contrary to connection-oriented protocols, connectionless protocols do not generate an error in case of communication problems while sending data to a peer. If

data is lost, for example because of congestion, it is the responsibility of higher layer protocols to detect this. These transport protocols are mostly used by applications that do not require full reliability, so applications typically do not detect lost data.

To detect the disappearance of the peer communication partner, mobility solutions must use other mechanisms than transport protocol errors. Often a heartbeat mechanism is used: one side sends a small packet, typically using separate channel realized with an unreliable transport protocol. If the peer side receives this heartbeat packet, it immediately responds. If the sender does not receive an answer for a particular amount of time, it assumes that the peer has disappeared.

A heartbeat mechanism can be implemented in two ways when using the CAS. As with most mobility solutions, it can be implemented externally. For example, a user space daemon can periodically send a heartbeat to every peer the host is communicating with. If the peer stops responding, the daemon sends a signal (an event in case of the DiPS+ prototype) to the CAS. The CAS responds to the signal by suspending all sessions with that peer. A second solution is to make heartbeats part of the protocol. The CAS header can be extended with a heartbeat flag. Heartbeats can be sent as empty CAS packets, i.e. a packet containing only a CAS header, that carries the heartbeat flag, or they can be piggybacked on packets containing application data. For every heart beat, the command sequence is increased. If the peer notices the heartbeat flag, it sends a reply with the heartbeat and the acknowledgement flag set in the header. The advantages of the heartbeat daemon approach are that changes to the heartbeat policy can be realized without having to alter the CAS protocol. Using a heartbeat daemon can also be more efficient than the protocol approach. With the daemon approach, the heartbeats are realized at the granularity of a host. With the protocol approach, heartbeats are implemented at the granularity of a session. The host level granularity can be particularly interesting if there are a lot of sessions established between peers: only one heartbeat must be sent between 2 hosts, independent of the amount of sessions there exist between those hosts. The main advantage of realizing heartbeats in the protocol is the total autonomy of the solution; there are no dependencies on external systems.

6.5.3 Network traffic during immediate handover

If performing an unanticipated but immediate handover, a CAS session on the client side is suspended unanticipatedly and immediately resumed using a new transport connection initiated from the new network. Where the immediate CAS suspension and resumption protocol handling may not be noticed on faster networks, it still has consequences on the transport protocol level. What exactly happens differs from protocol to protocol. This section highlights the consequences of immediate handovers on UDP and TCP traffic.

6.5.3.1 UDP

If UDP is used, a number of packets will be lost during handover. In the DiPS+ implementation of the CAS, the socket blocks immediately on the client side as a consequence of unanticipated suspension. The server side is initially not aware of the handover of the client side and keeps sending datagram packets. These datagram packets may get lost because they are still sent to the client's old address. The client immediately sends a resumption request from its new location. Only when the server receives the resumption request, it also suspends unanticipatedly and executes the session resumption protocol. Because the server side CAS is now suspended, the server side application also becomes blocked and stops sending data. All UDP packets the server sends between the moment that the client performs the handover, and the moment the server receives the resumption request are lost.

6.5.3.2 TCP

In the case of TCP, a similar situation occurs, except for the fact that no data is lost during the handover, but a short reduction in throughput can be monitored after session resumption. Right after the client migrated to the new network, the server side is not immediately aware of the handover. As soon as the replacement connection is established and a resumption request is received from the client, the server socket blocks until session resumption is complete. After completing resumption, the replacement TCP connection will still be transmitting in slow start mode because of TCP's flow control. The TCP connection is adjusting to the congestion situation in the new network.

6.6 Validation in industry projects

This work has been evaluated in 2 industry projects. Both the AMS and CAS have been successfully applied in two projects, the PEPiTA and SCAN project, which were conducted in cooperation with industrial partners. The following sections shortly outline the goals for each project and describe how the CAS and/or the AMS helps fulfilling these goals.

6.6.1 Project PEPiTA

Telecom operators believe that future revenue will come from service provisioning rather than mere network access. Telecom operators are therefore searching to extend their core business towards service provisioning. Their presence in the home network makes them the ideal point of contact for service provisioning. These services include services offered by the access provider as well as third party service providers.

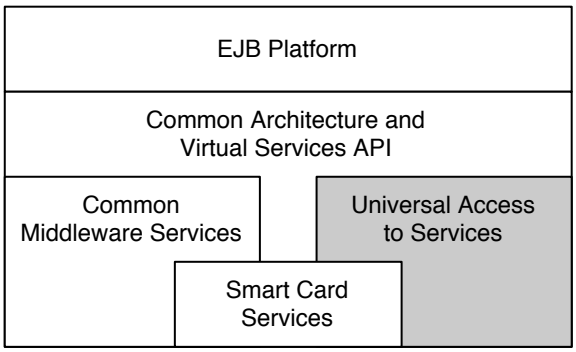


Figure 6.13: The PEPiTA software platform

This shift from access providing to service providing requires additional support on the edge of the network. Where access providers mainly provide transparent bit transport to their end-users, service providers must provide a *software platform* that allows the easy development and deployment of services. The proliferation of new technologies such as new terminal types (PC, PDAs, cellular phones, set top boxes, etc.) and access network types (LANs, home networks, mobile networks, etc.) makes the realization of such a platform a challenging task.

The main goal of PEPiTA[3] [ITE99] is to realize such a software platform which allows service development independent of a particular access network and the used terminal type. It consists of the following subsystems, visualized in Figure 6.13. On the bottom, the PEPiTA platform realizes the core network services. The services include *Common Middleware Services* with typical examples like security (authentication, authorization and data protection), transaction services and user profile management, *Smartcard Services* which offer security services and user profile management integrated in a smartcard on the terminal side, and *Universal Access to Services* which offers the necessary software components for access network independent communication services and terminal independence.

The *Common Architecture and Virtual Services API* hides the particular technologies and implementation details of the lower middleware, smartcard and access services. The used technologies can thus easily be altered or replaced without affecting the application. On top of the common architecture and virtual services API an *Enterprise Java Bean Platform* is provided, which is an enhanced EJB platform that incorporates the additional common middleware services.

[3]PEPiTA has been carried out in the context of the ITEA program with the support of the Flemish Institute for the advancement of scientific-technological research in the industry (IWT PEPiTA #990219).

6.6.1.1 AMS role in PEPiTA

The AMS is included in the protocol stack that is part of the Universal Access to Services part of the PEPiTA software platform. Because the access network on which an application will be deployed is not known beforehand, applications cannot be programmed to use a fixed set of protocols. Instead, protocol stacks are built on application demand in the PEPiTA system using a *stack composition framework* (SCF). The application can instruct the SCF to build a protocol stack given a number of high level protocol stack requirements [ŞMBV03, ŞVB02]. For example, an application can state that it wants to set up a multimedia session, using a transport protocol that does not have to offer reliable data transfer to a client not located on the local area network. Depending on the terminal's access network, the available protocols on the terminal and the service the application wishes to connect to, the resulting stack may for instance consist of a normal UDP/IP stack that also offers the SIP protocol [HSSR99] for negotiating the application's multimedia session.

To communicate with a service, a PEPiTA application uses a generic address as defined by the AMS. The SCF can use the information in the generic address while building a protocol stack. Once the protocol stack is built, the AMS can reduce the service's generic address and a transport connection can be established.

The main advantage of using the AMS in the PEPiTA platform is that applications can be developed independent of the network in which it will be deployed. They must not be adapted for every possible access network. Moreover, having to adapt every applications to all possible network combinations would quickly result in an unmanageable service platform.

6.6.2 Project SCAN

There is a trend where home networks are evolving to a situation where multiple access networks are available. These access networks will be available simultaneously, using varying technologies such as television cable, DSL, wireless (UMTS, GPRS...) or even satellite networks. Currently, the selection of an access network is done by the end user and typically remains fixed for a long time. In the case of mobile computing devices, access networks may change more frequently. At work, a user's laptop may be attached to the network with a cable, at home the user may be connected wirelessly. However, as long as the user does not move, the access network typically does not change.

In service centric access networks, the end user no longer explicitly selects an access network. Instead, the access network is selected when the user starts using a particular network service. In the PEPiTA world, where access providers are adopting the role of service provider, it is a logical consequence that consulting a service may require the need to connect to a different access network. The network infrastructure is hence adapted to the needs of the service, instead of

the traditional way where services are developed with the network capabilities in mind.

The SCAN project[4] [Alc03] pursues the goal of offering a software platform that enables the development of SCAN applications. This includes system support on the client side, application server side and in the access network. The software platform on the client side is responsible for handling the aspects of connecting to multiple access networks and switching to a different network based on the service's requirements. This includes application support but also management of the underlying access technology (cable and DSL modems, etc.). On the server side, a J2EE platform is offered that is adapted to function in a distributed access network environment where downtime must be reduced to a minimum. SCAN support in the access network encompasses protocols and hardware that allow connecting to multiple networks simultaneously.

6.6.2.1 The role of the AMS and CAS in SCAN

The CAS and AMS are included in the SCAN client and server. It will be very likely that the user will be consulting several services simultaneously. If the user starts to use a particular service, it is possible that the user's computer switches to another access network. This kind of dynamic network behavior is problematic for applications because of changing network characteristics, changing network layer addresses, possible long periods of disconnection, etc.

The CAS and AMS offer support in this network environment where such handovers can occur. Without a mobility solution, handovers are fatal for applications that were active before the network handover. The CAS allows existing service sessions to continue on the new access network after a handover was performed, or block when this is not feasible. Applications get feedback from the CAS when a handover is performed so they can decide whether the new access network is still adequate to continue handling the service. If the new access network is not adequate or its service provider policy does not allow the user to consult the service from a different access network, the session can be suspended. The AMS bridges protocol differences that result from the handovers.

[4]The SCAN research project has been carried out in order of Alcatel Bell, supported by the Flemish institute for the advancement of scientific-technological research in the industry (IWT SCAN #010319).

Chapter 7

Related work

A vast amount of research has already been conducted in the field of mobile and wireless computing. An overview of this research is given in Section 7.1. Solutions are categorized according to where in the protocol stack they are applied. Session layer solutions are discussed separately in Section 7.2 because they are more closely related to the solution proposed in this work. Section 7.3 shortly describes the GSM system and discusses the position of GSM telecommunication networks with respect to Session Layer Mobility Solutions in contemporary and future networks.

7.1 Existing solutions to mobile computing

This section discusses non-Session Layer Mobility Solutions. The mobility solutions are organized according to mobility solution type: Section 7.1.1 discusses Network Layer Mobility Solutions, Section 7.1.2 Transport Layer Mobility Solutions, Section 7.1.3 Socket Layer Mobility Solutions and Section 7.1.4 treats Proxy Mobility Solutions. We shortly describe each mobility solution and evaluate them with respect to the challenges proposed in Section 2.1. For each solution category, a general conclusion is formulated.

7.1.1 Network Layer Mobility Solutions

Network layer mobility solutions (NLMSs) are the most popular solutions in mobile environments. Network problems in the mobile computing domain have mainly been perceived as an addressing problem. When an endpoint moves, its network layer address changes. Therefore, the ideal place to handle mobility is where the problem occurs: the network layer.

Nearly all network layer mobility solutions are developed for IP networks because IP is the most ubiquitous network technology. Future communication net-

works, referred to as fourth generation (4G) networks, are claimed to be all-IP networks using a heterogeneous set of access technologies.

NLMSs can be divided in two solution categories: solutions based on IP unicast and solutions based on multicast. Unicast solutions are used most frequently, while multicast solutions can be considered a research effort. Section 7.1.1.1 and Section 7.1.1.2 give a short outline on the principles of both solution types. Section 7.1.1.3 evaluates the solutions in the light of the dynamic network challenges introduced in this work.

7.1.1.1 Unicast based solutions

Network layer mobility solutions are mostly unicast based solutions [Per96, Per02, JPA04, IDJ91, IJ93, TYT91, PB94]. However, all these unicast based solutions are very similar because they can be mapped to the same architecture [BPT96]. For example, Mobile IP [Per96, Per02, JPA04], which is an IETF standard and also the most popular network layer mobility solution, can be mapped on this general architecture. This section therefore does not describe all possible unicast based NLMSs but shortly summarizes this architecture.

Unicast network layer mobility solutions use a *two-tier addressing scheme* when a correspondent node (CN) wishes to exchange data with a mobile node (MN). This two-tier addressing scheme provides a solution in the network layer for the problem of the dual role of the network layer address (see Section 2.1.1.1). When a CN communicates with a mobile node, two network layer addresses are used instead of one. The first address is called the *home address*. This address is used to identify the MN and remains fixed regardless of the MN's location. The second address, the *foreign address*, reflects the current location of the mobile node and is used by unicast solutions as a routing directive. This address changes every time the MN changes attachment point.

Because the application and transport layer use the network layer address to identify endpoints and transport protocol connections respectively, it is important that network layer addresses are always fixed. Transport and application layers will therefore always use the home address of the MN to communicate. The foreign address is only used by the NLMS to deliver data packets to the current location of the MN. The application and transport layer remain oblivious to the existence of the foreign address.

For this two-tier addressing approach to work, all NLMSs typically have 3 infrastructure components in the network. An example infrastructure is depicted in Figure 7.1. The figure shows how a packets from the CN to the MN is sent, using the 3 components. First, an *Address Translation Agent* (ATA) is needed to make the mapping from the home address to the foreign address when a CN sends something to the MN's home address. The ATA usually alters the data packet so the packet is routed to the MN's foreign address instead of its home address. In Figure 7.1, a packet that is sent from the CN to the MN is intercepted by an ATA

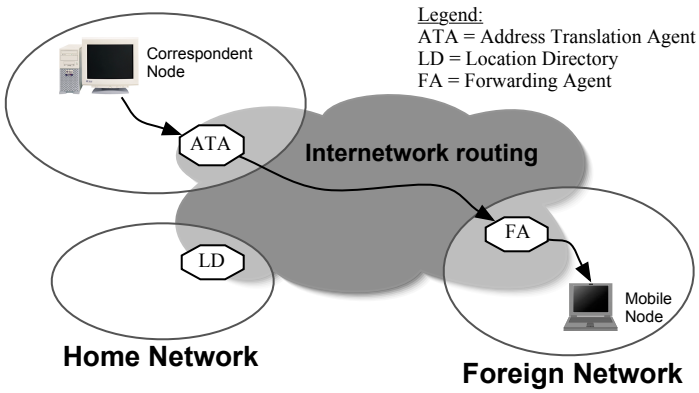


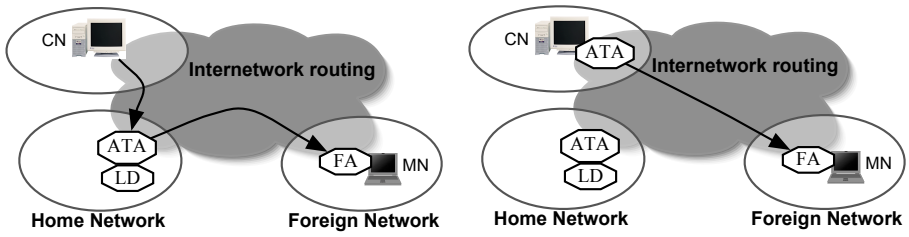
Figure 7.1: General architecture of network layer mobility solutions

that is located in the CN's network. Secondly, a *Forwarding Agent* (FA) is used to make the reverse mapping from the foreign address back to the home address, so the operation of higher protocol stack layers is not affected by address changes. More specifically, the packet is restored in its original form, as it was sent to the MN's home network. The FA finally delivers the packet to the MN. The FA in the figure is located in the network where the MN is currently located. Thirdly, a *Location Directory* (LD) is needed. The location directory maintains the mapping between the MN's home address and the current foreign address. The MN or its current FA update the LD with the new foreign address when the MN changes attachment point.

Theoretically, the MN can send a reply packet back to the CN without the extra components if the CN is not mobile. The source address in the transmitted packet is the home address of the CN. In practice, this does not always work, because edge routers on an intranet often filter outgoing packets that originate from a source address that is not covered by the intranet's subnet range [CB96]. To overcome this problem, the replies from the MN are sent back through the ATA which then forwards it to the CN after it changed the source address on the packet to the MN's home address.

These 3 components can be found in every mobility solution that operates in the network layer. NLMSs differ mainly in the location of the components. Figure 7.2a shows where the 3 components are located for Mobile IP. The ATA, which is called the home agent in the Mobile IP model, and LD are collocated in the MN's home network. The FA can be integrated with the mobile host, as shown in the Figure, or can be a separate machine that is dedicated to provide network access to mobile hosts.

Packets sent from the CN are always first routed through the home network



(a) Mobile IP

(b) Mobile IP with route optimization

Figure 7.2: The general architecture of network layer mobility solutions applied to Mobile IP

before being sent to the MN's current location. The replies from the MN are sent directly to the CN. This routing situation is often referred to as *triangular routing*. A routing optimization added to Mobile IPv4 [PJ01] and Mobile IPv6 [JPA04] addresses this triangular routing problem by allowing a device on the CN's network to be informed of the foreign address of a MN. This device can then perform the ATA functionality itself and send packets directly to the MN's foreign address. This is depicted in Figure 7.2b.

In mobile environments with frequently changing node locations, updating the ATA can introduce considerable overhead and packet loss. *Micromobility protocols* [CGK+02, CGKW02] have been developed to support fast handovers to reduce delay and packet loss. Techniques to limit updates to the ATA include FA hierarchies and paging. *Hierarchies of FAs* avoid that location updates must be sent to the ATA. FAs are typically collocated with an organization's internet gateway. As long as the MN moves within the boundaries of that organization, mobility updates are only sent to that organization's FA. Only when the MN leaves the organization's network, the parent FA in the hierarchy is notified of an update. This approach keeps mobility updates confined to small areas; updates must not traverse the entire network to the MN's ATA. *Paging* allows a MN to travel without the need for continuous notification of its current location. When the FA has packets available for the MN, the MN will be paged. Only when a MN receives a paging request, the MN will update its exact location. This approach works well when in a network with a hierarchy of FAs. *Improved handover techniques* mainly boil down to a proactive handover before the existing connection is lost completely. These three techniques are often used in conjunction with each other. Example micromobility solutions are Cellular IP [Val99], Hawaii [RVS+02] and Hierarchical Mobile IP [GJP04, SCEB05].

7.1.1.2 Multicast based solutions

Research efforts to enable mobile behavior in the network by using IP multicast have also been conducted [MB97]. Multicast based solutions obtain mobility by assigning a class D multicast address to every mobile node (MN). Essential to the multicast mobility model is that a multicast address is a *location independent identifier*. When a correspondent node (CN) wants to send something to the MN, it multicasts a packet to the MN's multicast address. The multicast routers in the address will send this packet to every member of the multicast group, which is in this case only the mobile node. The exact location of the MN must not be known as long as it is registered with a multicast router to the multicast group that is identified with the MN's class D address.

Although this approach is very promising, the authors of this system identified a number of problems that must be solved before this approach can be widely implemented. The major problems are a consequence of the multicast nature of the solution. First, there is a scalability problem in the addressing space. The amount of available class D addresses is limited. Secondly, the solutions requires every router in the network to be multicast enabled, otherwise CNs cannot send packets to the mobile host. Thirdly, a lot of protocols are not designed to work with class D addresses. TCP, ARP and ICMP are protocols that show erroneous behavior if used with multicast addresses.

7.1.1.3 General NLMS evaluation

NLMSs only support **address changes** and do not support **protocol changes**. They are designed to be used with one particular network layer protocol. NLMSs are typically designed as an extension to an existing network layer protocol, and use the protocol's services to realize mobile behavior. NLMSs depend on protocol properties such as the used address scheme. Consequently, it is impossible to change the network protocol on which they depend.

The main benefit of NLMSs is also the major disadvantage: total **transparency**. Without additional support protocol stack layers above the transport layer are completely oblivious to mobile behavior that occurs in the network. Consequently, transport layer protocols and the application layer cannot adapt to changing network conditions, unless additional services are made available to them. Moreover, network layer solutions handle the problem on changing network layer addresses. **Disconnected operation**, the absence of a network layer address, is not supported. NLMSs, both unicast and multicast based, therefore do not address the challenge on application awareness (see Section 2.1.2.1).

Unicast solutions depend heavily on specialized **network infrastructure**: They require a home and foreign agent deployed in every subnet that must support mobility. This introduces triangular routing in the network, which increases communication overhead. However, network infrastructure also improves handover

performance of NLMSs. The use FA hierarchies prevents location updates from having to travel all the way to the ATA. End-to-end solutions cannot benefit from such optimizations if they want to stay network infrastructure independent.

Multicast solutions basically do not need specialized **network infrastructure**, but require a number of essential protocols to be adapted if they are to work with multicast IP addresses. **General applicability** of multicast solutions is therefore a problem.

Security is an important issue when the MN or FA send a location update to the location directory. Authenticating the updates is required, otherwise it is possible that a malicious node sends an erroneous foreign address to the LD, leaving the MN unreachable.

7.1.2 Transport Layer Mobility Solutions

Most transport layer solutions are realized as extensions of TCP. The main motivation to handle mobility in the transport layer is the end-to-end argument: functionality is best provided at the endpoints, as close as possible to the application layer. Moreover, overhead of transport protocol extensions is very limited.

We describe four TLMSs Section 7.1.2.1 describes TCP-R, Section 7.1.2.2 discusses TCP Migrate and Section 7.1.2.3 treats a mobility solution that is used mainly for improved service availability.

7.1.2.1 TCP-R

According to the authors of TCP-R (TCP Redirections) [FYT97], mobility solutions have two responsibilities: they must ensure *continuous operation* and offer support for *compensative operation*. Continuous operation is the ability to keep transport connections running when IP addresses change due to access network changes. Compensative operation is the ability to establish new connections to and from the mobile node. For compensative operation, TCP-R relies on network layer solutions techniques like Mobile IP [Per02] or on Dynamic DNS [TRB97]. TCP-R only offers continuous operation and is self supporting – it does not depend on network layer solutions to realize continuous operation.

When endpoints migrate, TCP-R keeps established TCP connections alive by revising the IP addresses used to identify the connections. The authors refer to this revision process as *redirecting* the TCP connection. Since TCP-R does not depend on NLMSs to realize continuous operation, IP address changes are visible to the transport layer. When the mobile node (MN) detects an IP address change (it is not specified how this is detected), the node's TCP-R sends a redirect message to the correspondent node (CN), indicating that the old address must be replaced with the new address. To ensure the identity of the MN, the CN responds to the redirect message with an authentication request. When the MN successfully

authenticates itself, the IP address is revised, and the TCP connection can resume operation.

To illustrate, TCP-R can be used as follows. Suppose a CN establishes a connection with a MN. It depends on Mobile IP to do this. It creates a TCP connection using the MN's home address. The MN's home agent forwards the packets to the MN's current location. When the TCP connection is successfully established, TCP-R no longer requires Mobile IP's compensative functionality. TCP-R can immediately redirect the connection to replace the MN's home address with its current foreign address. This eliminates the triangular routing that is introduced by a non-optimized Mobile IP implementation. When the mobile node migrates to another network, it repeats the redirection process to revise the TCP connection with the new address.

TCP-R realizes redirection by means of TCP options in the TCP header. TCP-R detects if the peer node supports the redirection functionality by sending a TCP option in the SYN segment. In that case, the peer responds with another option which contains the authentication information that must be used to authenticate future redirection requests. The redirection request and authentication handling is also done using TCP options. To handle the requests correctly, the TCP state diagram is altered by adding three states. Two of these states (`RD_SENT` and `RD_RECEIVED`) handle the sending and receiving of redirect messages. The third state, `RD_WAIT` is used to prevent the TCP connection from aborting when the TCP retransmission timer expires.

Evaluation To handle **address changes**, TCP-R makes the distinction between compensative and continuous operation. All transport layer mobility solutions explicitly require a dynamic name service, which they use to retrieve a host's address. The name service must be dynamic because the endpoint's address changes when it moves. TCP-R does not need a dynamic name service. Instead it relies on Mobile IP for compensative operation: it uses the MN's home address to establish a connection. To avoid triangular routing, TCP-R redirects the connection to the mobile node's foreign address.

To support **virtual circuit continuity**, compensative operation is still required. TCP-R assumes that only one of the two endpoints moves. When both endpoints move simultaneously, it is not possible to send a redirection request to the peer node's address because it changed too. Compensative operation is required to retrieve the new network layer address of the peer node.

TCP-R is limited to TCP and does not support **protocol changes**. TCP-R allows revising the IP address of the mobile node, it does not allow the address type to change, e.g. from IPv4 to IPv6. Reliable communication is ensured by the mechanisms provided by TCP. **Disconnected operation** is supported because the modifications to the TCP protocol, more specifically the `RD_WAIT` state, handle timeouts of the retransmission timer.

General applicability of TCP-R is limited. TCP-R requires a number of intrusive changes to the TCP protocol's state transition diagram. However, TCP-R does not break when used with traditional TCP implementations, because it first detects if the peer TCP implementation supports TCP-R. TCP-R also depends on TCP options, which are used to detect redirection support and to handle the management of connection redirections. These protocol specific features are not available for every transport protocol. The TCP-R solution concept is therefore limited to protocols that require a connection establishment procedure and support the exchange of options during that procedure.

TCP-R uses Mobile IP for compensative operation. Hence TCP-R depends on the **network infrastructure** that is required by Mobile IP.

TCP-R exchanges authentication information when a TCP connection is established. This information is used to **secure** future redirection requests.

The only goal of TCP-R is to ensure continuous operation of transport layer connections. **Application feedback** is not supported. Mobility events occur transparently for the application layer.

7.1.2.2 TCP Migrate

The goal of TCP Migrate [SB00] is to provide end-to-end mobility support. The motivation for an end-to-end solutions is that it is the application's responsibility to specify the need for mobility support. Additionally, changes at the endpoints meet less resistance than changes in the network core.

TCP Migrate uses DNS to locate Mobile Nodes (MN). DNS names are invariant while the IP address may change, especially in a mobile environment. When a MN changes attachment point, it must update the mapping with the DNS server. To avoid stale mapping, name mappings must not be cached.

When a node is located, a TCP connection is established. During connection establishment, a connection token is negotiated. This token identifies the TCP connection. Normally a TCP connection is identified using a 4-tuple, consisting of the IP addresses and TCP ports that both peers are using to communicate. When addresses change, both endpoints can still identify the connection using the negotiated token.

Token negotiation must happen in a secure way in order to avoid hijacking of the connection. If the used network supports IPSec, no additional measures must be taken. In the case the network does not provide security measures, TCP Migrate uses an Elliptic Curve [Kob87] Diffie Hellman key exchange algorithm [DH76] to establish a secret key between the two communicating endpoints. This key is used the connection token is computed.

When a MN changes attachment point, it can resume the connection by sending a resumption request with the negotiated token to the CN. This request is signed using the secret key that was established during connection establishment. If the CN recognizes the token and can validate the signature, the connection can be

resumed. The connection's 4-tuple is adapted with the new IP address and TCP port.

Similar to TCP-R, TCP Migrate is realized using TCP options. During TCP's three way handshake a *Migrate permitted* option is used to negotiate the token. The two approaches differ in the way migration is performed. To migrate a session, the MN sends a new SYN packet with a *Migrate* option that contains the negotiated token and the request signature. The TCP protocol must hence be adapted to accept SYN packets on an established connection. Additionally, TCP Migrate also introduces a `MIGRATE_WAIT` state to the TCP state machine. This state is introduced to handle the rare case that the MN's IP address is reused by another node (let us call it node B) shortly after the MN left its attachment point. If the CN sends a TCP packet to this IP address, the TCP implementation of node B will respond with a reset packet because it is obviously not aware of the TCP connection. Upon reception of the reset packet, the TCP connection on the CN is moved to the `MIGRATE_WAIT` state, where it awaits the SYN packet carrying the migrate option. The TCP connection on the CN remains in this state for a limited period before it is aborted. The authors recommend to keep a TCP connection in this state for a 2MSL (Maximum Segment Lifetime) time period as specified in [ISI81].

Evaluation TCP Migrate supports network layer **address changes**. The IP addresses that are used to identify a connection are allowed to change because TCP Migrate uses a connection token to identify the connection after migration. **Protocol changes** are not supported.

TCP Migrate is not totally **transparent** for the application, because an application must be able to choose whether it needs mobility support or not. An application must hence explicitly state its need for mobility support. Application control is limited though because TCP Migrate does not offer **feedback** to the application when an MN actually changes attachment point.

TCP Migrate does not handle **disconnected operation**. The `MIGRATE_WAIT` state is only meant to handle reset packets on an established connection. When the retransmission timer times out, TCP Migrate aborts the connection, where TCP-R moves to the `RD_WAIT` state until the MN is reattached to the network again and starts negotiating a redirection.

The authors claim that TCP Migrate's migration mechanism is **generally applicable**: it can be applied easily in other protocols. They refer to application layer UDP based protocols such as RTP[SCFJ96], Quicktime [App06] and Real [Rea06]. On that level in the protocol stack, it is possible to provide adequate application feedback.

No additional **network infrastructure** is required. Migrate does not support the simultaneous movement of both TCP endpoints. If the correspondent node never moves, the mobile node will always be able to reconnect to its correspondent

node. The CN will never have to update its address with a dynamic DNS service and the MN will hence never have to look up the address of the CN again.

Endpoint migration is **secured** by means of Diffie-Hellman elliptic curve cryptography. A connection token is computed using a shared secret key, and migration requests are also signed using this key.

7.1.2.3 Endpoint migration for improved service availability

Transport layer mobility solutions have not only been used in networks where endpoints are mobile. Endpoint migration has also proven to improve network service availability. The idea is to improve service availability by means of a pool of servers that replicate the same service and are geographically spread. When a server becomes overloaded or fails, the network endpoint on this server is migrated to another server in the pool. In this service model, long living transport connections are used to handle the service. If the service can be handled using short living connections, connection endpoint migration is pointless. In such cases load balancing can be done at *connection creation time* by using a front-end transport or layer switch that distributes new connections in to different servers in the server pool [DCH+97].

The advantages of using endpoint migration in the service availability domain are twofold. First, the service is not interrupted. Another server takes over and resumes service handling. Secondly, the use of end-to-end transport layer mechanisms eliminates the need of intermediate proxy infrastructure, which is usually the bottleneck in the system.

There exist a number of solutions that can be used to realize service availability in this way. TCP Migrate (Section 7.1.2.2) has been validated in this context. Migratory TCP [SSI01, SSII02] was developed specifically for this purpose. Also SCTP can be used in this context. Migratory TCP and SCTP are discussed in more detail in the following sections.

Like TCP Migrate, Migratory TCP is also realized by means of TCP option exchange, but does not change the TCP state transition diagram. Migration is typically initiated by the client but can also be triggered by another server in the pool. The migration of the TCP endpoint to another server is handled by the server pool. After migration, the client can continue to use the service; service handling is never aborted and restarted. Moving an endpoint to another server also involves migrating application state to the new server, which can be very complex. This problem does not occur in the domain of endpoint mobility because that state, i.e. the entire application, moves along with the host.

Another protocol that was designed to be fault tolerant is SCTP [SXM+00]. SCTP is a datagram-oriented protocol but provides reliable communication. Applications establish SCTP *associations* rather than connections. The protocol allows multiplexing multiple data streams within a single association. An endpoint in an association supports multiple IP addresses, which is called multi-homing.

When a failure occurs on one IP address, the traffic is redirected to an alternate IP address. SCTP hence also performs endpoint migration. The main drawback of SCTP's approach is that the endpoints must be known at association creation time. In dynamic networks, endpoint movements are not predictable. To address this problem, an SCTP extension called Dynamic Address Reconfiguration [SRX+05] is being developed. SCTP with this extension is called mobile SCTP [RT04].

Evaluation Transport layer solutions that are developed for endpoint migration to improve network service availability support **address changes**. However, the possible address changes need to be known beforehand because network endpoints can only move inside the server pool. This is not the case in mobile, dynamic network environments. SCTP supports **protocol changes**, as it can handle a mix of both IPv4 and IPv6 addresses.

These solutions are not **transparent** for the application. The application may decide to migrate to another server when the service's efficiency has dropped below a certain threshold. Also, application state must often be moved to the new server in the pool. In that case, transparency on the server side is not possible.

These solutions usually do not support **disconnected operation**. It is always assumed that a server is available in the pool. Additional **infrastructure** is typically not needed: IP addresses are known beforehand and must not be looked up using a dynamic name service. However, Migratory TCP needs a separate network to transfer application specific state between servers in the pool. Without that network the solution fails if the network interface of the overloaded server has crashed.

Security is often not addressed when moving network endpoints in the server pool. Migratory TCP does not include any security provisions for connection hijacking. Because the IP addresses of the servers in the pool are known beforehand, it is also difficult to hijack a Migratory TCP connection from a different location. SCTP also does not address the possibility of hijacking an association. In case dynamic address reconfiguration is used, the user is only made aware of the security consequences. The dynamic address reconfiguration proposal does not address potential vulnerabilities.

Concerning **general applicability**, Migratory TCP depends on TCP options to realize network endpoint migration. The approach can hence not be applied in a transport protocol that does not offer the possibility for exchanging specialized information at connection establishment time. General applicability of the mobility solution when using SCTP is poor because its multihoming support is only available when using the protocol.

7.1.2.4 General TLMS evaluation

TLMSs are all end-to-end solutions and do not need extra infrastructure to realize the movement of an endpoint. They all address the problem of **address changes** as consequence of mobile behavior. They typically do not handle **protocol changes**, SCTP being the only exception. Compared to NLMS, these solutions do not suffer from triangular routing. In contradiction to proxy mobility solutions, TLMSs must not send all traffic to a proxy to obtain mobility functionality.

All mobility solutions rely on a name service or network layer mobility solution to locate the endpoint in a mobile network. Name services are widely available and accepted in contemporary networks; no specialized **network infrastructure** is needed. Only TCP-R generalizes the need for a name service to the need for solutions that offer compensative operation (i.e. tracking operation, see Section 3.2.3), such as a.o. Mobile IP. Also in the case both endpoints move, a name service must be contacted to relocate at least one endpoint (the name service is then used to track the endpoints). However, because name service lookups are typically implemented in the application layer, relocation of both endpoints is usually not supported, except in the case of TCP-R. The network model assumed by endpoint migration solutions for improved service availability also does not anticipate the movement of both endpoints. Endpoints only change at the server side.

The solutions are not **generally applicable**. All TLMSs are extensions to existing protocols (TCP). Only SCTP has been designed with multi-homing. Extensions to existing protocols require a detection mechanism to ensure that the peer implementation also supports the extension. TCP options have been designed to be ignored when they are not supported. The solutions that adapt the protocol specification by changing the TCP state transition diagram are backward compatible. Protocol extensions can not easily be applied to other transport layer protocols. The authors of TCP Migrate have applied the solution *concept* to other UDP based *application layer* protocols.

TLMSs can support **disconnected operation**. They can prevent timeouts which, in the case of TCP, may result in connection abortions. TCP Migrate does not offer disconnected operation and therefore expects endpoint migration to happen instantaneously. Endpoint migration solutions for improved service availability also do not support disconnection.

Application feedback does not result automatically from TLMSs. Most solutions do not involve the application in the endpoint migration handling. Migratory TCP allows the application to trigger endpoint migration to another server endpoint if it considers the current server endpoint no longer adequate to continue service handling. TCP Migrate allows the application to choose whether it needs mobility support.

TLMSs that are designed to support endpoint migration to locations that are unknown beforehand, contain additional **security** mechanisms. These mechanisms enforce that the endpoint that resumes the connection is the same endpoint

that established the connection. TLMSs that are developed for improved service availability (Section 7.1.2.3) do not have such security mechanisms.

7.1.3 Socket Layer Mobility Solutions

Although socket layer mobility solutions (SoLMSs) are also located between the transport layer and the application layer, they are different from session layer solutions. First, socket layer solutions do not offer the concept of a session to the application. Instead, they adapt the semantics of transport protocol connections so they are more adequate to be used in a dynamic network environment. Secondly, socket layer solutions are implemented in user space, and not inside the protocol stack. Hence they do not adapt transport layer protocols like TLMSs. The socket layer [Ste95] is typically implemented as an application layer library that offers the BSD socket layer interface to the application. SoLMSs offer the same socket API with modified semantics so mobile behavior becomes possible (see section 2.2.3). This section shortly outlines a number of socket layer solutions: Rocks/Racks (Reliable sockets/packets), MobileSocket and a mobile TCP socket and concludes with a summary of SoLMS properties.

7.1.3.1 Rocks and Racks

The main goal of Rocks [ZM02], which is short for Reliable Sockets, is to provide endpoint mobility for TCP connections using only user level mechanisms. Rocks is realized as a user level application library which is interposed between the application process and the system's socket library. This library offers the same socket API to the application process. Applications must not be adapted to use Rocks, they must only be linked to the Rocks library so they call into that library instead of the system's socket library. To realize the functionality of the socket API, Rocks uses the system's TCP implementation, but implements additional functionality to offer the application a TCP service that is resistant to the consequences of mobile behavior. Additionally, Rocks also offers a Rocks expanded API for mobile aware applications. This API allows mobile aware applications to set policies for mobile socket behavior. Racks performs exactly the same functionality but is realized as a user level protocol instead of a user level library, which is made possible by using a packet filter [MRA87]. The remainder of the text will refer to Rocks, but everything is applicable to both Rocks and Racks, unless explicitly stated otherwise.

When an application opens a TCP connection, a Rocks enabled system checks if a peer system is also Rocks enabled using the *Enhancement Detection Protocol* (EDP). Detection is realized by probing the peer using a rare use of TCP: Rocks establishes a connection with a peer server node, and immediately performs a half close. A Rocks enabled server node that detects this, sends an announcement on the connection to indicate that it is Rocks enabled. If the client node receives this,

it opens a new connection to the server node, also to announce that it is Rocks enabled. After the announcement, both peers negotiate what enhancement they will use (Rocks or racks) and initialize these enhancements. Initialization encompasses the exchange of a Diffie-Hellman key, which will be used to authenticate the endpoints in case of future address changes. Finally a UDP control socket is established, which is used to send control messages to the peer and is mainly used to send UDP heartbeats to detect if the peer node is still reachable. After detection, normal communication can proceed using the established TCP connection.

If a Rocks enabled system detects that the peer node has become unreachable, the Rocks socket is put into a suspended state, and tries to reconnect to the peer. During socket suspension, application calls on the socket are blocked. Reconnection encompasses the establishing a new data connection, authenticating to the peer Rocks system and establishing a new control socket. Additionally in flight data is recovered by applying a go-back-N retransmission algorithm.

Evaluation Rocks is an application layer solution, developed to support **address changes**. **Protocol changes** are not supported, mainly because Rocks depends on particular usage scenarios of TCP. **Virtual circuit continuity** is realized by establishing a new transport connection when the old transport connection is got aborted or when the Rocks notices that the peer has disappeared. **Byte stream consistency** is guaranteed by using a double buffering solution.

Rocks supports **disconnected operation**, by blocking socket calls when the data connection is aborted. Rocks is completely **transparent** for legacy applications while it still allows mobile aware application to specify special policies that adapt the behavior of Rocks. Rocks provides additional **security** measures to authenticate endpoints in the case of mobility events.

Rocks is an end-to-end solution and does not depend on specialised **network infrastructure** to operate. Rocks can use normal domain name lookups to locate a peer node, even after it moved. In the case of TLMSs, the problem of domain name lookups cannot be solved in the transport layer in the protocol stack but must be addressed differently (see compensative operation in section 7.1.2.1)

Rocks is an application layer approach and does not require any adaptations to transport protocols. However, the solution has a number of problems concerning **general applicability**. We shortly discuss three problems: problems with the use of interpositioning libraries, the dependency on TCP usage scenarios and the use of a separate data and control communication channel.

First, the authors experienced a number of difficulties due to the implementation nature of the solution. Implementing Rocks as an interpositioning library between the application and the system's protocol stack leads to a number of problems. Applications use sockets in a number of ways that are incompatible with the Rocks library. Example application behavior is the passing of socket descriptors to other processes, like processes that are created using an `exec` system call. In

such cases, the user level Rocks library state is lost. To solve this issue, a number of additional system calls must also be virtualized. This problem does not occur with Racks because it does not operate on the socket layer, but modifies and acts on TCP packets that are detected by the packet filter.

Secondly, Rocks depends on the use of a specific use of TCP to detect if a peer node also supports Rocks. The authors claim that this is no problem because using a half close on a TCP socket right after connection establishment is never done. Nevertheless, the EDP cannot be used with other transport protocols that do not support a half close, or do not support connection establishment.

Thirdly, the usage of multiple communication channels can be problematic. Rocks establishes multiple sockets and depends on the use of UDP heartbeat probes to detect if the peer host suddenly became unreachable. Rocks handles a separate control and data channel, mainly because it is difficult to combine application and control data in a single TCP connection. We chose to avoid this approach in he CAS because the separation of control results in a number of deployment obstacles. For example, FTP also separates control and data channels. Controlling access to FTP servers in a networking is hard to do without a stateful firewall because the TCP ports that must be used for the data channels are exchanged on the control channel.

7.1.3.2 MobileSocket

The goal of MobileSocket [OMTT00] is to ensure virtual circuit continuity for TCP connections in a mobile environment. MobileSocket realizes four requirements. Byte stream consistency must be maintained, the implementation must be minimal and simple, application modification must be avoided and mobile aware applications should dispose of means to influence the mobility solution.

MobileSocket is realized in Java, and implements the Java socket interface. Every application using the standard Java socket can use MobileSocket without any modification. MobileSocket also offers support to ask for explicit endpoint redirection by allowing the application to explicitly ask to suspend and resume a TCP connection through an extra interface. MobileSocket uses an AWT Event-based interface to notify the application of mobility events if desired.

A MobileSocket internally maintains a state transition diagram that indicates in what state the Socket is: closed, established, implicitly suspended or explicitly suspended. The socket is explicitly suspended when the application asked to suspend the socket.

MobileSocket internally uses standard Java sockets to realize TCP communication, similar to Rocks. When the Java socket breaks, the MobileSocket's socket is implicitly suspended. Subsequently, MobileSocket will try to establish a new Java socket to resume communication. The mechanism responsible for socket re-establishment is called *Dynamic Socket Switching* (DSS). DSS is complemented

with an *Application Layer Window* (ALW), which is a windowing system that guarantees byte stream consistency in the case of implicit suspension.

DSS uses three Java sockets for every MobileSocket that is created by the application. One socket is used to realize the data connection. If the data connection is successfully established, MobileSocket first uses it to exchange the data (a.o. port numbers and authentication information) that is needed to create a control socket. This control socket is used to exchange suspend requests between the endpoints. A third server socket, called the redirection socket, is created and listens on a port number that was received when the control socket was established. The Redirection server socket is contacted by the peer that wants to reconnect after moving to another access point.

Evaluation MobileSocket is also an end-to-end SoLMS that handles **address changes** as a consequence of mobile behavior. It has only been applied using TCP sockets and therefore does not support **protocol changes**. One of the main goals of the solution was to ensure **virtual circuit continuity** and **byte stream consistency**. This is realized by means of an application layer window (ALW) which is a double buffering algorithm.

MobileSocket is optionally **transparent** for the application. Sockets can suspend implicitly if one of the network endpoints disappears. If desired, the application can be notified of suspension and resumption events. It is also possible for an application to suspend explicitly if desired. **Disconnected operation** is supported.

MobileSocket does not provide any **security** measures to ensure that a reconnecting network endpoint is the same endpoint as the endpoint that originally established the MobileSocket.

Concerning **general applicability** of MobileSocket, there is one argument pro and two arguments against. An argument pro is that MobileSocket offers the same interface as a normal socket and adapts the behavior to be able to deal with address changes. MobileSocket uses the object-oriented programming techniques in Java to override normal socket behavior. It can therefore operate transparently for the application. Only the semantics of the calls on a socket object will be different as a consequence of the mobility measures taken by the MobileSocket. Mobile aware applications can optionally use the interface that MobileSocket offers to explicitly suspend and resume a connection and the event notification mechanism to be notified when sockets become suspended.

The first argument against general applicability of MobileSocket is that it is realized in Java and uses Java sockets. However, this does not necessarily mean that it cannot be realized on another system. Moreover, MobileSocket does not depend on specific TCP usage to realize the solution, like Rocks. MobileSocket could hence be used to migrate UDP endpoints too. Since UDP is an unreliable transport service, the ALW can be omitted. The control socket must still be

realized using a reliable transport protocol though, because MobileSocket depends on the reliable transmission of control messages. It must be noted that Rocks uses specific TCP usage scenarios to verify whether the remote endpoint supports Rocks support. MobileSocket does not depend on such scenarios, but also does not offer detection support.

The second argument against is the usage of multiple transport connections. Similar to Rocks, MobileSocket establishes an additional control connection and a server socket to support the migration of endpoint services. These additional channels carry protocol information, such as TCP connection ports, that affect the further handling of the protocol. Protocols with separate data and control channels require special treatment if they are deployed in a network where packet filters are used to enforce network security.

7.1.3.3 Mobile Socket Layer (MSL)

Mobile Socket Layer [QYB97b, QYB97a] also offers mobility services for TCP connections by extending the socket layer. No enhancements to the TCP protocol are required.

Applications that use MSL set up a *TCP association* with each other. From the point of view of an application, a TCP association is identical to a TCP connection: it is identified using a 4 tuple, consisting of 2 IP addresses and 2 protocol ports. This 4 tuple never changes when a node moves between access networks. To realize this, MSL uses the concept of a home IP and a virtual port. This home IP and virtual port never change in the lifetime of a TCP association and are used by the application. A TCP association uses a normal TCP connection to transmit application and control data. This TCP connection is established using the node's current IP address, which the node obtained when it arrived at the access network.

MSL maintains a mapping between the virtual addresses and ports and the associated TCP connection's addresses and ports. MSL uses a virtual port protocol (VPP) to establish such a mapping and to coordinate changes in these mappings when the old TCP connection is replaced by a new connection. The VPP sends application data and control data over the same TCP connection. Control data is encapsulated in *protocol data units* (PDU). The VPP extracts PDUs out of the data stream and maps them to the correct TCP associations.

When the TCP connection breaks, the TCP association takes additional measures to guarantee byte stream consistency. It introduces the concept of an association window: both endpoints register how many bytes they have sent and received successfully during the lifetime of the TCP association. When an association's connection fails, the MSL retrieves the content of the send window. When a new connection is established, both endpoints exchange their association window to get informed about how much data the peer has received. The data that was not acknowledged is resent on the new connection. In the absence of a TCP connection, the MSL buffers data that the application wants to transmit until a

new connection is established.

Evaluation MSL is a solution that handles address changes by using a two-tier addressing scheme. The application uses addresses and ports that never change. The addresses and ports of the associated TCP connection can always change. The MSL is responsible for maintaining the mapping between the two addresses. MSL does not support **protocol changes**, it is designed to run on TCP only. It is designed to realize **virtual circuit continuity** and ensures **byte stream consistency** using an association window and by extracting the TCP send window from the associated TCP connection.

MSL is totally **transparent** for the application. Loss of connection is perceived by the application as loss of bandwidth. The MSL can be realized by explicitly offering a new socket API to the application or by using the normal internet domain socket API and letting MSL handle both normal and mobile TCP connections. Only the latter case is transparent to the application, but requires a mechanism to detect if the peer also supports MSL. The authors of MSL propose the use of a TCP option or a magic number that is ignored by the correspondent node when MSL is not supported.

MSL is not **generally applicable**. It needs to be adapted before it can be applied to other protocols than TCP. However, the solution concept would remain the same: address changes can be masked by using a two-tier addressing scheme and a protocol that coordinates changes between the two tiers. To ensure byte stream consistency, it needs an association window to be able to synchronize after mobility, and a mechanism to extract the contents of the transport connection's send buffer.

MSL is and end-to-end solution that does not depend on additional **network infrastructure**. However, if both endpoints move, a dynamic name service will be required to locate the peer endpoint.

MSL does not contain any **security** measures to authenticate endpoints after they migrated to another network.

7.1.3.4 Persistent Connection

Persistent Connection (PerCon) [ZD95] is a system that offers a transport connection model that survives failures in general. Failures are transport protocol connections that break due to mobile behavior, but also system and application crashes. PerCon only addresses the persistence of transport protocol connections. It does not handle the persistence of application state in case of crashes.

PerCon introduces the principle of a persistent process which possess a number of a persistent connections. Persistent processes and connections only exist in a logical way and are realized by at most one physical process and transient (or physical) transport connection respectively. If the physical process or connection fails, it is replaced by a new one, as a continuation of the same persistent process or

connection. Replacement physical processes must not be created on the host where the previous process crashed; a replacement process may be created on a different location. Therefore persistent processes, connections and endpoints (sockets) are identified independent of their location: they are not identified using IP addresses.

PerCon manages the relations between persistent and physical connections on the socket layer, in conjunction with a centralized name service. Such a name service is required because processes can move to another host. When a physical process is started, it must declare to what persistent process it belongs. Every persistent connection between two processes must be registered with the name service. If the application crashes or a physical connection breaks, the affected persistent connections are passivated on the name service and on all the parties that were communicating with the application. When a newly created physical process indicates that it is the continuation of a persistent process, the name service triggers the reactivation of all passivated persistent connections, on all affected parties. PerCon does not offer any support to maintain byte stream consistency when physical connections fail. When this guarantee is required, it's the application's responsibility.

Evaluation PerCon's main goal is to provide a reliable computing environment where not only the transport protocol connection can break, but where the application can crash too. PerCon replaces crashed applications by new ones and coordinates the establishment of replacement connections. These connections may use other addresses and protocol ports; PerCon hence supports **address changes**. This support for address changes is not only used to realize host mobility, it also allows processes to migrate to other hosts. Migration is very limited though, because it does not support the migration of application state. The authors assume that applications are stateless or are transaction based.

Protocol changes are normally not supported since PerCon is designed to be used on a TCP/IP network. Because every broken connection is replaced by a new one, PerCon realizes **virtual circuit continuity**. However, PerCon does not maintain **byte stream consistency** for a persistent connection. PerCon does not take additional measures such as double buffering which is necessary when a transport connection fails unexpectedly. The solution is therefore only valuable for applications that use an extra reliability scheme besides TCP.

PerCon offers support for **disconnected operation** since broken connections are passivated on all the affected parties. PerCon does not perform any **security checks** to verify that a replacement process can be trusted.

PerCon is not **transparent** for the application. An application with PerCon support uses a different socket API than the traditional APIs offered by current operating systems. An application must be adapted to explicitly indicate the persistent process it belongs to, and must also register the names of persistent connections to the name service. However, the passivation and reactivation of a

persistent connection is hidden from the physical process.

PerCon is not a pure end-to-end solution because it depends on the availability of extra **infrastructure**. PerCon requires a name service that maintains information for every running process and every persistent connection these processes establish. This name service is essential to the correct operation of the solution. Most mobility solutions only require a name service to locate the correspondent node when initiating communication for the first time or if the correspondent node has also moved when trying to reconnect after migration.

General applicability of the solution is limited, mainly because of its dependency on a name service. Additionally, applications must all be adapted to use PerCon's logical naming scheme for persistent processes and persistent connections. When reliable communication is required, the application must be adapted to ensure reliability because Percon does not offer that service. Also if an application is stateful, the application is responsible to recover that state after a crash because PerCon does not provide any state handling.

7.1.3.5 General SoLMS evaluation

All SoLMSs support **address changes**, mostly by establishing replacement connections. Only MSL keeps the address changes explicitly hidden from the application by introducing a two-tier addressing scheme, similar to Mobile IP. All described socket layer solutions are designed for TCP. However, they should be able to cope with **protocol changes**. Because SoLMSs decouple the application socket from the transport connections that are used to communicate, the different connections could be realized using different protocols during the lifetime of a socket. But, even with SoLMSs, applications are still programmed with a particular protocol stack in mind. If an application is written to be deployed in an IPv4 network, it cannot be deployed in or migrated to an IPv6 network, unless the application supports these protocols or the SoLMS can handle these protocol discrepancies for the application.

All solutions offer support for **virtual circuit continuity**. If a TCP connection breaks, the SoLMS must take additional measures to ensure **byte stream consistency**. This is typically realized by means of a double buffering approach. The data in the send window of the TCP connection is also buffered in the SoLMS. This is necessary because otherwise the contents of the TCP window is lost when the connection is aborted. When a new connection is established, the communicating SoLMSs resynchronize their own buffers and resend information that was not received with the old, aborted connection. MSL is the only solution that does not buffer the data. It extracts the contents of the send window of the TCP layer instead. This means that MSL must be able to retrieve the send window from an aborted connection. This will often require changes to the transport protocol's implementation. Normally, such functionality does not affect the protocol specification of the transport protocol. Persistent Connection is the only solution that

does not ensure byte stream consistency.

Socket layer mobility solutions are typically **transparent** for the application. Transparency is not obligatory however. MobileSocket offers special interfaces that applications can use to adapt their behavior and explicitly request suspension. The use of these interfaces is optional; legacy applications that use the traditional socket API benefit from transparent mobility support. Persistent Connection is the only exception. To be able to use Persistent Connection, intrusive changes to the application are required.

All solutions offer support for **disconnection**, because disconnection is part of normal operation. The described SoLMSs cannot access transport protocol state to prevent transport connections from aborting. Failing transport connections are therefore considered a normal occurrence. When a connection breaks, the application remains disconnected until a replacement connection is established. In the meantime, the SoLMS blocks the application's communication directives, or notifies the application that it is disconnected.

SoLMSs do not need any specialized **network infrastructure**. Only Persistent Connection requires the use of a specialised name service. Concerning **general applicability**, all SoLMSs are specifically designed to be used with TCP. To realize a single mobile network endpoint, multiple transport protocol connections are often needed. Protocols using multiple transport connections typically require special treatment in the network. For example, network firewalls must know about the relation between the communication channels. Otherwise the chance exists that not all the connections will be allowed by the firewall.

Security is a property that is often neglected by SoLMSs. There are no guarantees that connections are not resumed by a third party.

7.1.4 Proxy Mobility Solutions (PMSs)

Proxies can be used in a multitude of environments, such as mobile environments. Proxies in mobile environments are often used to deal with the heterogeneity of the environment and the highly dynamic behavior of the mobile hosts that typically exacerbates the heterogeneity problem [ZD97]. Such proxies realize a great variety of tasks, that are usually application specific. Examples are proxies that drop unstructured data, such as frames in MPEG stream, proxies that compress data instead of dropping it before sending it on a low-speed mobile link, or proxies that use different transport protocols on mobile, wireless networks.

This section covers a special class of proxies: it describes general purpose proxies that provide support to migrate transport protocol endpoints. This section hence does not handle proxy solutions that offer such application specific mobility services.

Proxy mobility solutions are different from other network endpoint migration solutions because they depend on the typical *split connection scheme*. Proxies split a transport protocol connection in two parts. A connection from the mobile node

(MN) to the proxy, and a connection from the proxy to the server. PMSs expect that the connection from the MN to the proxy will break when the MN moves to another location. The connection from the proxy to the server remains intact until the communication session ends. The remainder of this section describes two PMSs: MSOCKS and Indirect TCP.

7.1.4.1 MSOCKS

MSOCKS [MB98, BMS02] is an extension to the SOCKS [LGL+96] protocol, which is typically used as an application layer internet firewall that provides fine-grained authentication support. MSOCKS extends the SOCKS protocol so that the transport connection from the proxy to the MN can be redirected. These extensions encompass the negotiation of a connection identifier when a MN creates a new TCP connection to the proxy, and a `RECONNECT` request that must be issued by the MN when it wishes to resume a previously created connection from a new location.

Byte stream consistency is maintained by the endpoints by using a proxy side technique called TCP Splice [MB99]. TCP Splice's main goal is to improve the performance of a proxy solution. To realize this, the two TCP connections are glued together in the transport layer, so data does not have to travel up to the application layer on the proxy server. When a TCP segment is received on one connection, the packet is immediately forwarded on the other TCP connection. The packet is altered so the headers, and therefore also the sequence numbers, are adapted to match the other connection. Therefore, the TCP protocols on the endpoints remain in control of byte stream consistency.

When a MN moves to a new location, the old TCP connection breaks and a new connection is established from the MN to the proxy. On the proxy, the old connection is *unspliced* and he new connection is *spliced* to the TCP connection that runs from the proxy to the server. The proxy is responsible for resynchronizing the sequence numbers of both connections. The MN is responsible for transferring transmission state from the broken to the new connection, so data that was sent but not yet acknowledged on the old connection is resent.

MSOCKS is realized on the MN, using a library interpositioning technique. Applications on the mobile node use the system's socket API, except they call into the MSOCKS library which alters the TCP socket's semantics. The MSOCKS library coordinates the transfer of TCP connection state between the old TCP connection and the replacement TCP connection. If the connection to the MN breaks, the MSOCKS library also buffers the data that the application on the MN wishes to send until the new connection is established.

On the proxy side, the TCP implementation must be adapted to support splicing. If the MN is absent, the MSOCKS proxy buffers data that is sent by the server.

Evaluation MSOCKS supports **address changes** on the MN by establishing a new transport connection when the old one breaks as a consequence of mobility. Protocol changes are not supported, MSOCKS was specifically designed for TCP. MSOCKS provides **virtual circuit continuity** and ensures **byte stream consistency** by means of an MSOCKS library on the MN that coordinates the transition between the old and new TCP connection and TCP splice which realizes TCP's end-to-end reliability semantics in a proxy setup.

MSOCKS is completely **transparent** on the MN. Mobility is also transparent on the server side, because the proxy hides mobility completely from the server by keeping the connection to the server intact while the MN connection is nonexistent. Both client and server are not mobility aware.

MSOCKS offers support for **disconnected operation**. As long as the MN remains disconnected, MSOCKS keeps buffering data for the application in the interposed library.

PMSs always depend on additional **network infrastructure**. The proxy always incurs extra overhead. Even with the performance improving splicing technique, data must still travel through the proxy, which is never placed ideally in the network. MSOCKS is not a true end-to-end solution, even though the endpoints remain responsible for byte stream consistency.

MSOCKS exchanges connection identifiers, but does not clarify if that happens in a **secure** way. Even if it does not happen securely, hijacking an MSOCKS connections will be difficult because the third party will typically not have access to the send window of the MN. A third party can never assure the channel will be in the correct state after hijacking it. Nevertheless, other end-to-end solutions, like TLMSs, that maintain transport protocol state if an endpoint moves, take security measures so connections cannot easily be taken over by requiring that endpoints must identify themselves in a secure way.

Concerning **general applicability**, MSOCKS also uses a library interposition technique on the mobile node. Problems that can occur with library interposition techniques have been discussed earlier in the section on SoLMSs (Section 7.1.3) and introduce limitations on the general applicability of the solution. The MSOCKS library also needs access to the send window of failed connections. Otherwise, MSOCKS can not resend data that was not yet acknowledged on the failed connection. An application layer library normally has no access to such information. Moreover, MSOCKS requires adaptations to the TCP implementation on the proxy to realize TCP Splice. However, because these adaptations are only required on the proxy, they have a minor effect on the general applicability of the solution.

7.1.4.2 Indirect TCP

The goal of Indirect TCP [BB95, BB97] is to improve the performance of wireless links that are inherently less performant and less reliable. TCP is inadequate in

such an environment because of its end-to-end semantics: it is difficult to determine the cause of packet loss on a TCP connection that spans multiple hops on the Internet. Indirect TCP (I-TCP) splits up a TCP connection in two connections. One connection spans the wired part of the connection, the other spans the wireless part of the connection. The transport protocols used on the wired and wireless parts may be different to better suit the network environment.

The proxy in I-TCP is the router that connects the wireless and the wired part and is called a mobility support router (MSR). This MSR router is always the gateway in a wireless cell. The mobile node (MN) establishes a transport connection to the MSR, which is always a single hop connection (there are no intermediate gateways between the MSR and MN). Using a proxy protocol, the MN requests to establish a connection to the correspondent node. The MSR will then establish a normal TCP connection on the wired part of the network.

Contrary to most proxy approaches, the proxy is not fixed in I-TCP. If a MN moves to another wireless network cell, the I-TCP connection is handed off to the MSR in the new cell. When a MN arrives in another cell, it sends a greeting to the local MSR indicating that a connection handover from its previous MSR is required. Both MSRs then cooperate to realize the handover. This handover only encompasses moving the connection state from the wired and wireless connection from the old MSR to the new MSR. The solution uses smart addressing tricks to avoid address changes in the handover. For the technical details we refer to [BB97]. The relocation of the proxy also affects the CN, because packets must suddenly be sent to another MSR. I-TCP relies on a NLMS (Columbia Mobile IP [IDJ91, IJ93]) to mask this mobile proxy behavior from the CN.

I-TCP realizes byte stream consistency by moving the state of the involved sockets from the old MSR to the new MSR. However, if an MSR fails or if an MN remains disconnected for a longer time period, TCP connections fail and I-TCP can no longer guarantee byte stream consistency.

Evaluation I-TCP avoids **address changes**: the MN always uses the same IP address. If IP address changes are required, the mobility solution can no longer be applied. I-TCP's main goal is to use **altered transport protocols** on the wireless network. It may be possible that other protocol enhancements are used after a handover. However, these enhancements mainly affect the transport layer, and not the network layer. The problem of address discrepancies between the two networks are not solved, especially because I-TCP uses special addressing tricks and depends on NLMSs to support proxy migration (see [BB97] for more details).

I-TCP realizes **virtual circuit continuity**, but does not guarantee **byte stream consistency** in case one of the TCP connections in the split connection scheme fails. The application must implement additional checks if full reliability is required.

I-TCP can be made completely **transparent** for the application, because I-

TCP does not support **application feedback**. The authors propose a number of choices on how the I-TCP proxy protocol can be implemented, ranging from a normal library to a kernel based trap mechanism. As a proof of concept prototype, I-TCP was implemented as a library that exports a specialised communication API.

I-TCP does not support **disconnected operation**, because it cannot guarantee byte stream consistency. Contrary to MSOCKS, acknowledgement handling is not completely controlled by the endpoints. I-TCP cannot splice the two transport connections together, because they may be different. The transport connection on the wireless network may be adapted to better suit the wireless network environment.

I-TCP does not seem to offer **security** measures to deal with connection hijackings by a malicious third party. I-TCP also does not take any security measures when an endpoint has to identify itself when it moves to another MSR.

General applicability of I-TCP is very limited. To be able to apply the solution, the mobile node's IP address must not change. It must be possible to migrate transport protocol connection state between MSRs, which requires an adapted TCP protocol. These adaptations only affect MSRs. The need for **additional** infrastructure is substantial. Apart from the MSR that functions as a proxy, also a NLMS is required. Otherwise the correspondent node cannot cope with proxy migration.

7.1.4.3 PMS evaluation

Only MSOCKS handles IP address changes. I-TCP avoids the address changing problem. Protocol changes are supported by I-TCP, the changes are only intended to improve communication over wireless channels. Virtual circuit continuity is realized by both solutions, I-TCP does not guarantee **byte stream consistency**.

Support for **disconnected operation** does not appear to be straightforward. The transport connection from the proxy to the MN is expected to break. In the case of I-TCP, disconnection should not exceed the timeouts of TCP. Otherwise it cannot guarantee byte stream consistency. MSOCKS on the other hand intervenes as a buffer for the time that the MN is not connected.

PMSs keep mobility events **transparent** for the application and do not provide **application feedback**. Applications can not adapt their business logic to the changing network environment. However the split connection scheme allows to add optimizations to transport connections that operate on the wireless channel. These optimizations can in principle be chosen by the application when the connection is established. PMSs change the semantics of a transport connection without having to change the protocol stack API. The implementation can be realized using a library interpositioning technique. Other implementation techniques such as adding another socket type or using a library with an explicit PMS API are possible but not transparent and can consequently not be used by legacy applications.

General applicability is limited in the sense that all applications require adaptations to the TCP protocol. These adaptations are limited to the proxy however.

PMSs all use additional **network infrastructure**: the proxy. Network traffic usually does not follow the most optimal path if it must pass through a proxy machine. Proxies are never placed optimal in the network. I-TCP is an exceptional solution because it allows the handover of a connection to another proxy. Additionally, because a router in the network functions as a I-TCP proxy server, packets will in that case follow the most optimal route. On the other hand, I-TCP requires the presence of a NLMSs, which typically also needs additional infrastructure. Because proxies are also realized in the application layer, data that is transmitted must always travel through the proxy's protocol stack to the application layer on the proxy, where it is immediately copied to the other connection. MSOCKS eliminates this overhead using TCP Splice.

Proxy mobility solutions appear to ignore any need for **security**. MNs and proxies do not appear to exchange connection identifiers in a secure way. It is therefore perfectly possible for a third party to take over a connection channel from a Mobile Node. This should not be an insurmountable problem as it should be relatively easy to add security measures to the proxy protocol.

7.2 Session layer solutions

A session layer or session layer services are not entirely new. The OSI model contains a session layer and more recently a number of session layer approaches for mobility have been proposed. Although the OSI session layer was not designed to address mobile endpoint behavior, it does address some of the challenges that we formulated for evaluating mobility solution categories in Section 2.2. The other session layer solutions discussed in this section are Migrate, TESLA and SLM and were specifically developed to address mobile endpoint behavior. For each solution, we discuss how they address the mobility solution challenges. Additionally, we discuss how they realize the session management tasks outlined in Section 3.5.

7.2.1 OSI model

The OSI protocol stack model [Zim80] encompasses a session layer. Like the OSI transport layer, the OSI session layer offers data transport, but defines some additional services [Tan96], such as dialog management, activity management and synchronization. Dialog management is used to determine which communicating peer is allowed to send data. Activity management allows a data stream to be split up in a number of activities. These activities are application defined, for example, a file transfer session will consist of one activity per file. Synchronization is used to resume communication after an error occurs.

The synchronization task of the OSI session layer resembles the communication semantics maintenance task (Section 3.5.7) of the session layer architecture proposed in this work. However, the synchronization of the OSI session layer is aimed to resolve problems that occur on higher levels in the protocol stack, like failing applications, whereas the session layer approach proposed in this work is aimed to address problems that originate in the network.

An application that runs on an OSI protocol stack can enter synchronization points in the data stream. If something goes wrong at either end of the data stream, the communicating applications can resynchronize by returning to the last valid synchronization point. The CAS does not allow the applications to enter synchronization points, instead it first determines if synchronization is needed (only with reliable transport protocols) and then applies a synchronization policy based on transport protocol parameters. For example, the TCP window size determines how much data the CAS must buffer.

7.2.1.1 Discussion

The OSI session layer is not really intended to support **address and protocol changes** because it was not developed to be deployed in dynamic networks. **Security measures** to prevent security holes introduced because of dynamic behavior are hence not included. The OSI session layer does support **disconnected operation**. The synchronization task of the session layer realizes **virtual circuit continuity** and **byte stream consistency**. A session is realized by multiple consecutive transport connections. A broken connection is replaced by a new one. Data loss is prevented by means of synchronization points in the data stream. It is not the goal of the OSI session layer to be **transparent** for the application. Instead it offers generic session services that can be optionally used by the application. **General applicability** of the OSI session layer solution is obvious, as it is part of the OSI specification. It does not depend on additional **network infrastructure**.

7.2.2 Migrate

Migrate [Sno03] is a session layer solution that addresses the consequences of Internet mobility, i.e. host mobility in an IP network. It realizes the session layer solution guidelines that were outlined in Section 3.2, but its architectural properties differ from those introduced in Section 3.4. This section summarizes the Migrate solution and then discusses how the session management tasks that were introduced in Section 3.5 are realized.

Migrate provides the application with a session concept that is very different from the session concept the CAS provides. Migrate's session is a collection of multiple transport connections. These transport connections do not have to be of the same type, for example, a session can consist of a TCP connection and an

RTP connection. A session can also have no associated connection; the absence of a connection does not automatically result in session suspension. The application determines whether a transport protocol connection (TPC) belongs to a session. The application must create a session, create a TPC and then add the TPC to the session. Migrate does not have a session socket. The application uses the transport protocol socket for communication. Session management, i.e. session creation, session teardown and session migration, is performed using a separate session application programming interface (API). In comparison, a CAS session corresponds only with one TPC.

If a host moves to another network attachment point, Migrate handles the consequences of mobile behavior for the TPCs of a session. An application can also migrate explicitly to another network attachment point, which is possible when the application runs on a host with multiple network interfaces. After migration, the mobile host resumes sessions by contacting the peer endpoints at the same IP address as before. If the peer is not responding, the session is suspended (disconnected operation). In case both communicating endpoints move, Migrate can no longer contact the correspondent endpoint at its previous IP address. Migrate then performs a new lookup using the naming system of choice. The application can register a name resolution callback function for every session.

Migrate ensures virtual circuit continuity and byte stream consistency when migrating sessions that consist of reliable transport connections like TCP. Migrate supports two mechanisms: connection virtualization and endpoint rebinding. *Connection virtualization* introduces an indirection layer between the application and the protocol stack which mediates for the application. If the application requests a socket, the indirection layer creates two sockets: one that the application uses (the application socket), and a second one that is connected with the destination (the communication socket). The indirection layer splices the two sockets together; every call that the application makes on the application socket is forwarded to the communication socket. When a network attachment point change occurs, the indirection layer establishes a new connection using a new communication socket, unsplices the application socket and the old communication socket and splices the application socket with the new communication socket. Byte stream consistency is ensured for reliable transport protocols using double buffering. Migrate applies connection virtualization for both TCP and UDP. Migrate also supports *Endpoint rebinding* to ensure virtual circuit continuity. Endpoint rebinding does not require establishing a new TPC every time a network endpoint migrates to another network attachment point. Instead, the network endpoints of the existing connections can be moved to another location. This requires support from the transport protocol: TCP Migrate (see Section 7.1.2.2), which was developed by the same authors, is a transport protocol that supports endpoint rebinding. Byte stream consistency is ensured by the protocol itself. It must be noted however that TCP Migrate does not support disconnected operation. In comparison, the CAS only

support connection virtualization.

Migrate supports application feedback in two ways. First, it is possible to register a mobility handler for every session. This handler is a callback function, which is called when Migrate detects a change of network attachment point by the local or remote host. Secondly, Migrate also supports *session continuations* which are more powerful than a callback approach. Session continuations are a snapshot of an application's state and a function that adapts that state to reflect the new network situation when the session is resumed. This function may for instance evaluate network bandwidth or security measures available at the new network attachment point and change application state accordingly, for example by modifying the video compression algorithm or changing the encryption technique. An application is responsible for defining its own session continuation. A session continuation is normally created lazily upon Migrate's request, usually when a session becomes suspended. To preserve resources, Migrate can move session continuations to secondary storage and remove the application from system memory until the session can be resumed. In comparison, the CAS currently only supports the callback approach.

Migrate is realized as a user level Migrate daemon and a dynamically loadable library (DLL). The daemon manages all open sessions in conjunction with a connectivity monitor and policy engine. The Migrate daemon takes care of session continuation management and also coordinates session resumption. Migrate daemons on two hosts engaged in a session maintain a control channel. This control channel is used to negotiate session control parameters. For example, when resuming a session, the control channel is first established from the new network attachment point. The Migrate daemons then exchange the ports of the new replacement transport connection. The policy engine is used to determine when to change attachment points in case multiple access points become available. The advantage of this policy engine is that it can be used by both Migrate aware and legacy applications. The connectivity monitor offers limited support to verify whether an endpoint is still active, among others by monitoring local network interfacing, monitoring socket errors and sending probes on the control channel. The DLL must be used by applications that want to use Migrate. The library cooperates with the Migrate daemon for session management. The DLL is an interpositioning agent: it is used to intercept system calls to transport protocol sockets, which is necessary to realize connection virtualization. Also legacy applications can use Migrate by linking with the DLL. Migrate then transparently creates a session for every transport connection created by a legacy application.

7.2.2.1 Discussion

Migrate supports **address changes** by means of virtual circuit continuity. Migrate provides virtual circuit continuity using both connection virtualization and endpoint rebinding. Byte stream consistency is realized by means of double buffer-

ing when connection virtualization is used. In the case of endpoint rebinding, if the transport protocol that supports migrating endpoints is a reliable transport protocol, the transport protocol is also responsible to guarantee byte stream consistency. The goal of Migrate is to let the application keep control of the used protocol and the protocol's configuration parameters. **Protocol changes** are by consequence not supported by Migrate. The connection virtualization technique is supported for both TCP and UDP.

Migrate supports **disconnected operation**. If a TPC belonging to a session aborts, the session becomes suspended.

Migrate provides extensive **application feedback** mechanisms. When a handover occurs, Migrate calls the application's mobility handler if the application has registered one. When suspension occurs, Migrate can obtain a session continuation from a mobility aware application, and possibly remove the application from the system to preserve system resources. The session continuation allows the application to resume when the host reconnects and adapt to the new network circumstances.

Migrate is **generally applicable**. No protocol changes are necessary if connection virtualization is used. When using endpoint rebinding, both communicating endpoints must be equipped with the transport protocol that supports the endpoint rebinding techniques. Migrate can also be used for legacy applications by means of an interpositioning library that transparently creates a session for every TPC. Interpositioning libraries do have their disadvantages, as discussed in Section 7.1.3.1.

Migrate does not depend on specialized **network infrastructure**. Like all mobility solutions, Migrate needs a name service when both endpoints move. However, Migrate does not dictate the naming system the applications must use. Instead the application must provide a name resolver that Migrate can consult when necessary.

Migrate supports the basic **security** needs. When a session resumes from another endpoint, Migrate guarantees that it is the same endpoint that resumes a session.

7.2.2.2 Session management tasks

Because Migrate is a session layer solution, we shortly discuss how it realizes the session management tasks that were outlined in Section 3.5.

Transport and network protocol independent session identification. Migrate sessions are identified using a simple numbering scheme. Migrate's sessions do not depend on the identification mechanisms used by a particular transport or network protocol and are consequently not identified by parameters of these protocols. Since a session can consist of multiple TPCs, this would also not be feasible.

Protocol and address hiding. Migrate does not hide what protocols and addresses are currently used from the application. One of Migrate's goals is to let the application keep control of the communication protocols it wishes to use. Migrate hence does not hide protocols from the application, which impedes protocol changes when migrating. Applications communicate with the standard protocol sockets in contemporary protocol stack implementations. An application must configure such sockets explicitly with the peer's address, and the locally used addresses are also visible on such sockets.

Session state management. Migrate handles the state of a session using the Migrate daemon. Migrate sessions can be in a number of states: Connecting, Established, Migrating, Frozen, Lost and Unsupported. The state of a session is continuously monitored by means of a connectivity monitor.

Session negotiation protocol. Migrate daemons use a separate control channel to negotiate session establishment, suspension and resumption. This is comparable to FTP's control channel, that is used to initiate file transfers on separate data channels. Instead, the CAS uses session headers instead, to avoid an extra control channel.

Session support detection. The advantage of a separate control channel is its easiness to detect whether a peer also supports Migrate. When establishing a new session, Migrate tries to establish a control channel to the peer host to verify that it also supports Migrate. If the control channel cannot be established, Migrate assumes that the peer does not support Migrate. The Migrate session is then put in the Unsupported state, which means that the connections belonging to that session will not survive network attachment point changes.

Transport protocol management. Transport protocol management is handled by both the application and Migrate. The application is responsible for establishing a TPC and then informing Migrate that the TPC belongs to a particular Migrate session. Migrate is then responsible for handling the consequences of mobile behavior for the TPC.

Communication semantics maintenance. When reliable transport protocols are used, migrate realizes a double buffering mechanism to preserve byte stream consistency. It also supports transport protocols that realize endpoint rebinding. CAS only supports double buffering, because it wishes to remain transport protocol independent.

Offering application feedback. Migrate offers extensive application feedback, by allowing an application to register a mobility handler, which Migrate calls when mobility events occur. Session continuations can be created for applications with suspended sessions. Such session continuations are mainly used to preserve system resources.

7.2.3 TESLA

TESLA [Sal02] is a session layer solution that offers the programmer a generic framework to implement session layer services. TESLA is hence not only used to provide mobility, but has been evaluated with other session layer functionalities like encryption and compression.

TESLA's session concept is a *data flow*. Session layer functionality is realized as a *flow handler*. Flow handlers operate on a flow using an API which resembles the API of a traditional socket, only the semantics of the function calls are slightly adapted. For example, write and read operations never block and are always guaranteed to complete. A complete session layer solution is created by building a pipeline of flow handlers. For example, a pipeline of an encryption flow handler and a migration flow handler realizes a session layer solution that encrypts data while supporting mobility as well.

TESLA is realized as an interpositioning library. Socket calls are transformed to flow API calls. What the application perceives as a transport connection hence corresponds to one TESLA flow. Because a TESLA application uses a generic API, the session layer functionality is kept transparent for the application. However, TESLA can also be extended with session functionality specific interfaces. The use of these interfaces is usually optional, and allows the application to interact with a particular flow handler if that is needed and/or desired.

The Migration flow handler creates a new flow between itself and the transport layer every time the old flow breaks. Virtual circuit continuity is hence realized by means of connection virtualization. Byte stream consistency is realized by means of a double buffer that is maintained for every connection (flow) created by the application.

TESLA's migration flow handler is used by the Migrate session layer solution (Section 7.2.2). According to the authors, Migrate has replaced its own ad-hoc interpositioning library with TESLA. Migrate uses the flow handler to provide virtual circuit continuity and byte stream consistency for a single transport protocol connection, if the transport protocol does not support endpoint rebinding. Migrate also uses TESLA to offer Migrate specific functionality to the application: session management, application feedback and session continuations. The two solutions are hence complementary.

7.2.3.1 Discussion

Because TESLA is used in conjunction with Migrate, TESLA shares a lot of properties with Migrate. However, because TESLA introduces its own session concept and aims to be transparent, we shortly outline the properties of TESLA's migration flow handler without Migrate extensions.

TESLA supports **address changes**. Virtual circuit continuity and byte stream consistency are realized by means of connection virtualization and double buffering. **Protocol changes** are not supported. The application uses the traditional transport protocol sockets that must be configured explicitly with the protocols needed to communicate.

TESLA aims to be **transparent** for the application. This is obtained by implementing TESLA as an interpositioning library that intercepts socket calls. TESLA hides network errors in the migration flow handler. **Disconnected operation** is also hidden from the application.

TESLA is **generally applicable**, despite the known problems with the interpositioning library approach. TESLA also acknowledges these problems and provides workarounds. TESLA's Migration flow handler is an end-to-end solution that does not need additional infrastructure. Of course, like every mobility solution, if both endpoints move, TESLA will need to consult a name service. **Security** is not provided by the Migrate flow handler, but it can be provided by means of another flow handler if necessary.

7.2.3.2 Session management tasks

TESLA allows to create session layer services as stand-alone flow handlers. However, when using the migration flow handler, it depends on the Migrate session layer solution. TESLA hence only realizes a subset of the session management tasks that were outlined in Section 3.5. For the other tasks it depends on Migrate. This section discusses what tasks TESLA realizes, and what tasks are delegated to Migrate.

Transport and network protocol independent session identification. A TESLA flow handler is a different session concept than Migrate's session. TESLA identifies a flow handler by means of IP addresses and transport protocol ports. Flows can hence be identified using addresses and ports that no longer reflect the current network situation.

Protocol and address hiding. TESLA does not hide the protocols that are used to communicate, because the application uses traditional, protocol specific transport protocol sockets. Addresses are not hidden from the application. It is not known whether address changes are visible for the application because it is not

sure if the splicing mechanism used by the migration flow handler exposes these changes upward to the application.

Session state management. TESLA state management differs from flow handler to flow handler. In case of the migration flow handler, it is not clear what states a flow can be in. Presumably a flow can be in the same states of a Migrate session: Connecting, Established, Migrating Frozen and Lost. A TESLA flow will probably never be in the Unsupported state (see session support detection).

Session negotiation protocol. TESLA does not implement a negotiation protocol. It assumes that the peer host uses the same set of flow handlers. If not, a TESLA will not behave as expected. For example, when a host encrypts a data flow and the peer does not decrypt, the system simply won't function. The migration flow handler depends on Migrate for session negotiation tasks such as synchronizing a reliable data stream.

Session support detection. TESLA does not offer support to detect whether the remote host supports TESLA. The migration flow handler depends on Migrate to detect whether the peer system is also equipped with TESLA.

Transport protocol management. TESLA creates transport connections to realize the actual communication behavior. Because TESLA is realized as an application layer library, it can use the normal system calls.

Communication semantics maintenance. TESLA's migration flow handler uses a double buffer to maintain byte stream consistency in case the application requests a reliable data stream.

Offering application feedback By default TESLA's migration handler operates transparently for the application. TESLA supports exporting other functionality than the normal functionality of a traditional transport protocol socket. This capability is used to bring Migrate's application feedback mechanisms, being callbacks and session continuations, to the application.

7.2.4 SLM

The purpose of SLM [LLIS99] is to provide a mobility solution for the Internet that supports both device mobility and user mobility: next to moving devices between networks, it should also be possible to move applications between devices. For example, if a user arrives at his office, a conference call that was started on his cellphone can be transfered to his personal computer, where it is possible to improve the audio compression quality and use the call's video feed.

An SLM session normally corresponds with a single transport protocol connection, but it is possible to group several connections in a session. The only supported transport protocol connections are TCP connections. Every session is identified using a unique ID. To realize mobile behavior, SLM uses connection virtualization by introducing a socket connector between application and the operating system's protocol stack. The application holds a reference to a socket connector for the lifetime of the session, while the underlying transport connections can change. TCP semantics are maintained by means of a double buffering approach.

SLM's naming system is user based instead of home based. This is necessary because of the user mobility requirement: because an application does not necessarily stay on the same host, SLM requires a means to perform user tracking. SLM uses a special User Location Server (ULS) to track the location of SLM users. The functionality of the ULS is similar to the user location protocols used in SIP [HSSR99].

SLM uses a session protocol to perform session handovers to other locations or other devices. However, how session management information is exchanged is not known. Also the potentially complex session state transfer issue when migrating applications between devices is left unspecified. The SLM is transparent for the application: if a replacement TCP connection is established, this is hidden from the application. It is unclear how an application can adapt to the network situation.

7.2.4.1 Discussion

Like every mobility solution, SLM supports **address changes**. Address changes can be the consequence of device mobility but also of application migration between different hosts. All other mobility solutions, including CAS do not support application migration. SLM provides virtual circuit continuity by means of a connection virtualization approach, complemented with a double buffering technique to realize byte stream consistency. When performing user mobility, it is not clear how session state is transfered to the another host. **Protocol changes** are presumably not supported, SLM operates only above TCP.

SLM's connection virtualization is kept **transparent** for the application. Applications can hence not adapt to the currently applying network conditions when performing host mobility. The authors propose the use of application level proxies to adapt to new network conditions (low bandwidth) or new device conditions (low processing power).

Application awareness is required in the case of user mobility. It must be possible to migrate application state to the new device, and applications must be made aware that they are continuing a session that was started elsewhere. **General applicability** is limited, because applications must be adapted to support session handover in case of user mobility.

Although not mentioned explicitly, SLM presumably does support **disconnected operation**. A connection virtualization approach can typically support

basic disconnection support by blocking the application's networking system calls.

SLM depends on specialized **network infrastructure**: it requires a name service that tracks users instead of hosts. SLM does not provide any **security measures** in case a malicious application or host tries to resume a session.

7.2.4.2 Session Management Tasks

Little technical details of SLM are divulged in the literature. Nevertheless, we shortly discuss how SLM realizes the session management tasks outlined in Section 3.5.

Transport and network protocol independent session identification. A session is identified by a unique identifier. It is not known whether this ID is based on protocol parameters such as network layer addresses and transport layer addresses or on a different naming/numbering scheme.

Protocol and address hiding. SLM hides address changes for the application by transparently establishing replacing broken transport connections. It is not known if applications can query the used communication socket for the exact addresses that are used at a particular moment in time. SLM probably does not hide the used protocols, because SLM only supports TCP. It is not known whether both TCP/IPv4 and TCP/IPv6 connections are supported during the lifetime of one session.

Session state management. SLM probably uses a pretty complex state management algorithm because it also supports the handoff of sessions to other devices. Detailed state management is not discussed in the literature, however.

Session negotiation protocol. SLM uses a session negotiation protocol to migrate a session. In case of user mobility, three parties are involved in the protocol: the device that hosts the session, the device that will resume the session and the correspondent host. Details of the protocols are not discussed in the literature.

Session support detection. It is not known if SLM supports the detection of peer SLM support.

Transport protocol management. SLM uses a special socket connector that connects the application socket with the transport connection that is used to transport the data. SLM presumably uses normal TCP sockets to establish transport protocols.

Communication semantics maintenance. In case a TCP connection breaks as a consequence of mobile behavior, reliable transport is realized by means of double buffering.

Offering application feedback. SLM is transparent for the application: if a transport connection breaks, it is transparently replaced by a new one. However, if a session is handed over to another application running on a different device, there will be some interaction required between SLM and the application to transfer session specific application state. How SLM handles this is not discussed in the literature.

7.3 Mobility in the GSM world

The most successful mobility technology is without doubt GSM, or mobile telephony in general. We shortly outline the GSM architecture in Section 7.3.1 and describe how mobility is handled in that architecture. Section 7.3.2 discusses how mobile telephone systems relate to mobile computer networks and what the role of a session layer solution can be in such an environment.

7.3.1 Overview of the GSM system

The GSM system [Rah93, MP92] offers a number of services such as telephony, short message service (SMS) and data services such as GPRS. The system was designed as a mobile system from its inception, and was standardized so these services could be offered to the public over the entire european continent. GSM networks, or Public Land Mobile Networks (PLMN), are limited geographically. The range of a PLMN is limited by the borders of a country. Small overlaps between PLMN's are allowed because it is technically not possible to stop the network's coverage exactly at the country's border. In a country, there can be multiple overlapping PLMN's.

From the network point of view, a PLMN is cell based. The network consists of adjacent cells. A cell is realized by a Base Transceiver Station (BTS), which comprises an antenna and realizes the wireless communication. A mobile station (MS), i.e. a mobile phone, is always located in exactly one cell. From the management point of view, the PLMN's internal structure is hierarchical: At the highest level in the hierarchy are a number of Mobile Services Switching Centers (MSC), which are responsible for call setup between GSM users. Every MSC manages a number of Base Station Controllers (BSC) who control the Base Transceiver Stations (BTS).

In the GSM system, the MS can move between cells. To be able to continue communication a handover must be performed: the MS is moved from one BTS to another. It is possible that a MS moves to a BTS that managed by another

BSC, and on top of that, this BSC may even be managed by another MSC. The GSM system is able to handle such handovers without loss of service.

Moving between different PLMN's is called *roaming* in the GSM system. Administratively, it requires exchanging subscription information between the home PLMN and the visited PLMN, which is required to set up billing and to deduce the services the MS may use in the visited PLMN. Technically, a new cell and PLMN must be selected out of the available ones.

To perform a handover while roaming to another PLMN, subscription information must be exchanged, a new cell must be selected *and* a handover to the MSC that manages the new cell must be performed. To guarantee continued service to the user, this must happen fast enough so the call is not interrupted.

7.3.2 Discussion

Mobile behavior in a GSM network is mainly realized by the infrastructure. The infrastructure is responsible for coordinating the handovers. In a computer network, all mobility solutions that are realized above the network layer, thus also session layer mobility solutions, attempt to realize mobility in an end-to-end way. Changes to the network infrastructure are avoided because it is easier and cheaper to adapt the powerful and configurable end hosts in a computer network than to change the network infrastructure.

The main reason why computer networks and GSM networks approach mobility in a different way can be explained by the type of service they offer. The main goal of a GSM network is to offer uninterrupted voice services while moving. When offering voice services, it is not feasible to disconnect from the network because that is perceived as service failure. To avoid disconnection, the GSM network must ensure complete geographical coverage. This is realized by a carefully designed network infrastructure: a GSM network consists of a large number of adjacent cells controlled by a single network operator. The network operators have mutual agreements to enable roaming. This approach has proven to be very successful. Contemporary mobile, wireless computer networks offer best effort, easily accessible, world wide data communication services. Compared to voice services, disconnection for data communication services does not necessarily mean service failure. For example, in case of file transfer services, disconnections are not a problem as long as the files eventually arrive at their destination. The infrastructure has been realized in an ad-hoc way by installing wireless hotspots in public locations such as airports, bars and parks. Computer network infrastructure does not realize complete geographical coverage, and connectivity is very location based.

The service differences between computer network and mobile phone systems are disappearing though. Computer networks are increasingly used for voice services (Voice over IP, Skype, ...) because network availability is increasing. Also the new generation mobile telecommunication networks are becoming data driven (GPRS, UMTS) rather than voice driven.

The general belief is that future communication networks will consist of hybrid wired and wireless network technologies to exchange both voice and data communication. This trend is usually denominated as *4th generation networks*. This will enable the end user to use technologies that are slow but provide a large area coverage, like GSM networks, and to switch to short range, faster and cheaper access technologies when they become available, such as WiFi hotspots.

In such hybrid environments, a session layer system may be the enabling technology. Performing vertical handovers between different communication technologies that employ different protocol stacks will be greatly simplified when coordinated by a session layer solution. The CAS/AMS hides the technical consequences of such vertical handovers from the application layer, while still allowing application awareness. Additionally, session layer solutions are easier to apply in a hybrid communication environment because they are end-to-end solutions and typically do not depend on communication technology specific network infrastructure.

Chapter 8

Conclusion

In this dissertation we have realized a mobility solution in the protocol stack's session layer that can cope with mobility device behavior in heterogeneous networks. Section 8.1 summarizes the main contributions of this dissertation. The results obtained by this work create interesting topics for future research. A number of these topics are described in Section 8.2. Some final considerations concerning the position and relevance of the Session Layer Mobility Solutions in general with respect to other mobility solutions are given in Section 8.3.

8.1 Summary and contributions

The main contribution of this work is the development of a session layer mobility solution for dynamic networks. We have defined a dynamic network as a network that consists of mobile, heterogeneous computing devices (laptops, palmtops ...) which communicate by means of heterogeneous network technologies (WiFi, Bluetooth, UMTS ...). Dynamic networks have led to new ways of network computing. We have described a number of such new network paradigms that possess dynamic network properties, such as mobile networks, wireless networks and overlay networks. The mobility solutions domain is very broad: mobility solutions and their supporting technologies exist both in the system's protocol stack and on the middleware level. We limited the scope of this work to mobility solutions that are located in the protocol stack. Such solutions are more generally applicable because they are part of the operating system and are less dependent on particular deployment environments or software engineering tools.

In this work we have identified four challenges that should be addressed by mobility solutions when facing dynamic networks (see Chapter 2). The first challenge is the ability of a mobility solution to cope with address and protocol changes as a consequence of device mobility in a dynamic network. Most contemporary

mobility solutions cope with address changes, but not protocol changes. The second challenge concerns application involvement. If an application's host device is moved to a slower network, it might want to be informed in order to adapt its business logic. This is also the case if the device is disconnected from the network. The third challenge acknowledges that security is an important requirement for mobility solutions. The device's identity must be confirmed when it suddenly appears on another location in the network. The fourth challenge identifies the need for protocol stack and mobility solution flexibility. A mobility solution that supports protocol changes is not sufficient; the device's protocol stack must also be equipped with the necessary mechanisms to allow runtime changes to the protocol stack configuration. A mobility solution must be generally applicable without unexpected side effects and should not depend on specialized, solution specific network infrastructure.

Next to the identification of the four challenges, Chapter 2 also introduces a taxonomy of mobility solutions that classifies mobility solutions according to their location in the protocol stack. We have identified six types of mobility solutions: network layer mobility solutions, transport layer mobility solutions, session layer mobility solutions, socket layer mobility solutions, proxy mobility solutions and application specific mobility solutions. For every mobility solution type we have evaluated how they can realize the dynamic network challenges.

We motivate our choice for a session layer mobility solution in Chapter 3: in the session layer, it is possible to define a solution independent of transport and network layer protocols, which improves its applicability. Additionally, if the mobility solution is realized in the operating system's protocol stack instead of in an ad-hoc way in user space, there is no need to resolve problems with altered system call semantics.

Chapter 3 also presents the developed session layer approach. A session is defined as a communication channel between two communicating applications, similar to a transport protocol connection. However, a session is able to survive device mobility, where a normal transport protocol connection usually fails[1]. In the case there is no access network available, a session will be suspended. A session is suspended unanticipatedly if network access disappears unexpectedly. A session can also be suspended anticipatedly if disconnection can be predicted.

A general architecture for session layer solutions is introduced. This architecture explains the relationship between the session layer and the application layer and the relationship between the session layer and the transport layer. The application uses the session layer's services by means of a session socket. A session is realized by a normal transport protocol connection. If that connection breaks as consequence of mobile device behavior, it is replaced by a new transport protocol connection.

[1] We assume there is no transport layer or network layer mobility solution present in the protocol stack.

The developed session layer mobility solution (SeLMS) is responsible for eight tasks. First, a SeLMS must be able to detect if a peer is also equipped with the SeLMS. Secondly, a SeLMS is responsible for a session identification mechanism. This mechanism should not depend on transport protocol connection identification parameters or network layer addresses. Thirdly, the SeLMS must hide the protocols and addresses used to communicate from the application, because they are able to change in a dynamic network. An application should not have to cope with the technical aspects of such changes. Fourthly, a SeLMS is responsible for session state management. A session can be in a number of states depending on the network situation; a session can be closed, connecting, suspended, active, reconnecting. The behavior of the SeLMS depends this state. Fifthly, the SeLMS must implement and adhere to a session negotiation protocol when establishing, suspending anticipatedly, resuming and terminating a session with a peer SeLMS. Sixthly, a SeLMS is responsible for transport protocol management. A SeLMS depends on the transport layer's services to realize data exchange. The SeLMS must be able to establish and destroy transport connections. To be able to suspend a session, it must also detect transport connection failures. Seventhly, the SeLMS is responsible for communication semantics maintenance. If an application requires a reliable data stream service, the SeLMS must be able to guarantee reliable communication when the application moves or becomes disconnected for an arbitrary amount of time. Finally, the SeLMS must be able to inform the application when network events occur that might interest the application.

Chapters 4 and 5 respectively describe the connection abstraction system (CAS) and the address management system (AMS). These solutions adhere to the proposed session layer architecture and realize seven of the eight SeLMS tasks. The CAS realizes all tasks except SeLMS detection (task 1) and address and protocol hiding (task 3). The address and protocol hiding (task 3) is realized by the AMS. SeLMS detection (task 1) is currently not yet supported.

The CAS is realized as a communication protocol in the protocol stack. To use the CAS, applications must obtain a session socket from the operating system. The CAS protocol supports four protocol actions: session establishment, anticipated session suspension, session resumption and session termination. The CAS assumes that the transport layer only offers simple, unreliable data transport services. Because that's the most basic service of a transport protocol, the CAS remains transport protocol independent. Every protocol action is realized as a three way handshake to deal with potential loss of protocol messages. Protocol messages are encapsulated in CAS headers and are sent using the same transport protocol connection used for sending application data. We chose not to use a separate control channel to exchange session management information because protocols that do that are often difficult to deploy in contemporary networks. For example, the FTP protocol requires special treatment in a network that consists of firewalls and NATs. In case the session is realized using reliable transport protocol connec-

tions, the CAS buffers data that might be lost in case of connection failure. The CAS resynchronizes when resuming the session and can resend data that was lost. The CAS notifies the application when a session becomes suspended, is resumed, or when an immediate handover occurs. The CAS offers security mechanisms to protect against session hijacking and denial of service. A CAS offers the mechanisms to incorporate an authentication protocol when establishing and resuming a session. Denial of service attacks are avoided by limiting the rate of protocol requests for a session and providing a cleanup policy for suspended sessions.

The AMS uses generic addresses to hide the exact protocols and addresses used to communicate from the application. Generic addresses contain all possible protocols and addresses that can be used to communicate with a service. Applications only use generic addresses to identify a service. The address and protocol information is only used by the AMS. An application that wants to communicate with a service hands the generic address to the AMS, which selects a protocol combination from the generic address that takes the current network situation of the service client into account. The CAS uses this selection to establish a connection. If the application's host device moves, a new selection is made, and the CAS can establish a replacement connection that reflects the application's new network situation.

The design and implementation of the CAS and AMS in the DiPS+ protocol stack framework is described in Chapter 6. The DiPS+ framework supports runtime reconfiguration of the protocol stack and addresses the fourth dynamic network challenge concerning protocol stack flexibility. Both the CAS and AMS have been successfully applied in industrial projects. We have also evaluated the performance of the implementation by studying the processing overhead of the CAS implementation. The test results indicate that a CAS enabled protocol stack uses 10% more computing resources to exchange the same data than a protocol stack without the CAS. We believe this is an acceptable overhead, considering the solution's added value of endpoint mobility support, session suspension in case of disconnection and mechanisms for application notification. The overhead of the session protocol indicates that it is pointless to use the CAS for short communication sessions. Memory usage of the implementation depends on the buffers sizes per session. The buffer sizes of a session depend on the buffer sizes of the session's transport protocol connection. The code size of the implementation is smaller than the code size of TCP in DiPS+. We have no conclusive numbers of the code size if the CAS would be optimized to run on small, mobile, embedded devices. However, we have reasons to believe that the code size can be smaller than TCP's code size. The reaction speed of the proposed solution to mobility events depends on disconnection detection time and the overhead introduced by the three way handshake protocol actions. Detection is problematic when using connectionless transport protocols. In case of connection-oriented protocols, detection time depends on the timeout policies of the transport protocol. The speed

of the three way handshake protocol actions depends on the round trip time of the session.

Finally, the developed session layer architecture is compared with mobility solutions found in the literature (Chapter 7). Existing mobility solutions are categorized according to the mobility taxonomy presented in Chapter 2. In Chapter 2 we discussed how every solution type can typically address the dynamic network challenges. Existing mobility solutions usually only address a part of these challenges, mainly because they were not developed to operate in a dynamic network. For SeLMSs, we also discuss what session management tasks they realize. One SeLMS, Migrate, realizes all management tasks, except the task of address and protocol hiding (task 3). To our knowledge there are no other session layer mobility solutions than CAS/AMS that support protocol changes during the lifetime of a session.

8.2 Future work

Although the developed Session Layer Mobility Solution is a great improvement in the domain of mobility solutions, there is still much room for future research. We discuss five improvements for the CAS that must still be investigated.

First, the CAS does not support the task concerning session layer mobility solution detection (task 1). Before the CAS can be used it must determine whether the correspondent protocol stack is also equipped with the CAS. Since the CAS is an end-to-end solution not belonging to the network core, it is not certain that every host will be equipped with the CAS. We proposed the use of a third party network service to check CAS support on a particular endpoint. However, we prefer not to depend on additional network infrastructure. Because no separate control channel is used to exchange protocol messages, it is also not possible to try to establish a control channel to a well known transport layer port on the remote host and decide that the peer supports the CAS extension if the connection is successfully established. This approach is also not preferred because it makes the CAS dependent on a particular transport protocol; it listens on a particular port using a particular transport layer protocol. Once a transport connection is established to a service, it must already be known if the system supports CAS or not. It is hence not possible to incorporate detection as part of the session protocol. If a CAS header is sent to a system without CAS support, that header might violate the application layer service protocol.

Secondly, although the CAS provides the mechanisms to realize key establishment during session establishment and endpoint authentication during session resumption, this does not automatically guarantee that endpoint mobility is secure. Devising a sound authentication protocol with the required security properties is a difficult task and should be further investigated.

Thirdly, application feedback is currently limited to notifications of disconnec-

tion, reconnection and immediate handovers. It is still up to the application to determine the characteristics of the new network. Application feedback could be semantically richer; it could contain information about the type of network, the available bandwidth, signal strength in case of wireless networks.... Providing semantically richer application feedback does not require changes to the feedback mechanism. It may require more intensive network status monitoring. Next to disconnection and reconnection, also bandwidth or signal strength alterations must be detected.

Fourthly, the CAS currently only supports application mobility as a consequence of device mobility. An application could also be moved in the network by migrating it to another computing device. To support such mobility, the CAS needs to be extended. Application mobility requires the capturing of application state and the transfer of that state between devices. State capturing would be similar to Migrate's session continuation (discussed in Section 7.2.2), which contains application specific state and defines the required functionality to resume the application in a new network situation.

Fifthly, the three way handshake of the CAS resumption protocol decreases handover speed. When continuous network service must be realized, handover should be performed seamlessly. Handover speed must therefore be optimized. For this reason, horizontal handovers are handled on the data link layer. A higher layer mobility solution, (very often Mobile IP) deals with the consequences of possible address changes. If no address changes occur, horizontal handovers occur transparently without service disruption. Vertical handover speed can be improved by connecting to another network *before* disconnecting from the old network, performing the handover and then disconnect from the old network. Improved handover speed of a Session Layer Mobility Solution could be realized by performing session resumption for an established session. The Session Layer Mobility Solution would establish a new transport protocol connection using another network access point. When this connection is available, the session can use the new transport protocol connection. To obtain this, the session protocol must be altered and extra care must be taken to ensure communication semantics.

8.3 Some final considerations

This work contributes a Session Layer Mobility Solution that is flexible enough to cope with network heterogeneity. The solution has a number of properties other solutions do not have, like support for application involvement or the support for protocol changes. So do Session Layer Mobility Solutions suddenly make other mobility solutions inferior, invalid on unnecessary? Obviously, the answer is no. We give two possible reasons why other mobility solutions remain important.

First, the selection of a mobility solution depends on the requirements it must fulfill. It is possible that not every property of a Session Layer Mobility Solution is

required. For example, the need for application awareness may not be needed if the network supports seamless handovers and changes in bandwidth are not necessary or do not affect the application's business logic. In that case a transparent mobility solution suffices. Often, the properties of Network Layer Mobility Solutions appear to be sufficient. Moreover, Network Layer Mobility Solutions are the most popular solutions and hence have already proven their worth.

Secondly, mobility solutions are not mutually exclusive but can cooperate to realize a better mobility service. For example, a Session Layer Mobility Solution, which offers support for disconnected operation, can use a Transport Layer Mobility Solution that realizes reliable communication in case of endpoint movement. In turn, that Transport Layer Mobility Solution may delegate the task of handling address changes to a Network Layer Mobility Solution, instead of handling address changes separately for every transport connection itself. Finally, the Network Layer Mobility Solution implements the necessary algorithms to realize secure address changes. The resulting mobility service is secure and performant, with support for disconnection.

Nevertheless, Session Layer Mobility Solutions will gain importance. The related work described in this dissertation (Chapter 2) indicates that more and more recent mobility solutions are realized higher in the protocol stack, closer to the application. Additionally, the taxonomy of mobility solutions in Chapter 2 shows that, the higher a solution is located in the protocol stack, the more dynamic network challenges it can address. Both observations indicate that dynamic networks are becoming a reality: The specified dynamic network challenges are relevant because they are addressed by more and more mobility solutions and addressing these challenges higher in the protocol stack has become widely accepted. We therefore believe that Session Layer Mobility Solutions are the next step in the evolution of mobility solutions. Session Layer Mobility Solutions strive above all to be flexible, which will be one of the main requirement in the computer networks of tomorrow.

Glossary

Anticipated session suspension Anticipated session suspension happens when the Session Layer Mobility Solution explicitly requests the correspondent Session Layer Mobility Solution to suspend the session. Anticipated session suspension can happen on application request or when network access loss can be anticipated., *46*, *51*, 56, 62, 63, 66, 68, 82, 87, 89, 103, 125, 178, 179

Attack-equivalence A network environment where mobile devices frequently change access point require additional security measures to ensure that malicious users do not assume the identity of one of the communication partners. Additional security measures in such an environment typically only offer attack equivalence [Sno03]: the additional measures are only intended to address security risks that are the consequence of device movements. The security solutions offer a dynamic network environment where an attack can be reduced to an equivalent attack on a non-dynamic network., 16, *80*, 81, 82

Byte stream consistency Byte stream consistency indicates that no byte should be lost or reordered by events that are the consequence of dynamic endpoint behavior, such as endpoint mobility, disconnection and protocol changes., 18, 21, 25–28, 30, 42, 104, 150–156, 158–161, 163–165, 167–171

CAS client session A CAS client session is used to exchange data with the peer that participates in the session. A CAS client session is created by an application that initiates a session with a peer. A CAS client session is also created as the result of an `accept()` call on a session socket., 54, *54*, 55, 58, 63–65, 85, 86

CAS listen session A listen session is mainly an administrative means to create an access point that clients can connect to if they wish to use an application server's services. The result of contacting a CAS listen session is a client session which is used to exchange data., *54*, 55

Connection virtualization Connection virtualization [Sno03] is a session layer technique to obtain virtual circuit continuity by introducing a virtual connection independent of the transport protocol connection that realizes the virtual circuit. A virtual connection survives fatal failures that occur to the transport protocol connection and are the consequence of mobile behavior of the endpoint. To obtain virtual circuit continuity, a virtual circuit is created to replace the one that was aborted. To maintain byte stream consistency, the aborted and the new transport connection must be synchronized, for example, by means of a double buffering technique., 164, *164*, 165, 166, 168–171

Double buffering Double buffering [Sno03] is a technique to maintain byte stream consistency that is often used by session layer solutions that use connection virtualization. When a transport protocol connection breaks, it is replaced with another connection. All buffer content and other status information from a broken connection is cleaned up, making it impossible for the mobility solution to check what data has already been sent and received by the peer communication endpoint. Because it is not possible to access buffers and other protocol status information from the transport layer, mobility solutions on a higher layer maintain a send and receive buffer themselves, to ensure that no bytes are reordered. Hence, data buffering is twice, both in the transport layer and in the session layer mobility solution., 30, *67*, 104, 127, 150, 152, 155, 156, 164, 165, 167–172

Dynamic network In a dynamic network, the endpoints move between heterogeneous access points. Such endpoint mobility is not only enabled by portable devices (mobile networks) and wireless access technologies. Devices such as laptops allow a user to unplug the computer, move it to another access point and attach it to the network again. Wireless access technologies allow a user to do this without physically detaching and reattaching the device. Progress in broadband technology aims at the selection of access networks based on the requested service. This results in logical access network mobility instead of physical mobility: although the device is still attached using the same physical access point, it is now communicating on another network., 1–4, 6–8, 11–13, 15–17, 34, 40, 42, 44, 47, 50, 51, 55, 62, 90, 99, 101, 102, 104, 135, 138, 146, 147, 149, 177, 178

Endpoint mobility Endpoint mobility is the capability to cope with network attachment point changes during communication. A network endpoint can be moved because the device that hosts the endpoint is moved physically, or the application that owns the network endpoint can be moved to another host. In this work, endpoint mobility always refers to the former kind of endpoint movement., 2, 6–8, 36, 146, 149

Endpoint rebinding Endpoint rebinding [Sno03] is a transport layer mobility solution that offers virtual circuit continuity by allowing both network endpoints to change their network layer address during the lifetime of the virtual circuit. Network layer address changes are usually fatal for virtual circuits that are realized by contemporary transport layer protocols, because the virtual circuit is identified by means of the network layer address. Endpoint rebinding solutions identify the virtual circuit independent of the network layer address, allow mobile devices to change their network layer address and rebind themselves to the virtual circuit by authenticating themselves using the network layer address-independent identifier. Byte stream consistency is realized by the transport protocol's reliability measures., 164–168

Generic Address Generic addresses are used by the AMS to identify a service. A generic address is a container that holds all protocol and addressing information that can be used to communicate with a particular peer application. If an application wants to communicate with that service, the AMS selects adequate protocols and addresses from the generic address., 94, *94*, 95–99, 117, 119, 128, 134, 180

Handover Moving from one wireless access point to another access point requires a handover: moving the access connection from the antenna in the old access point to an antenna in the new access point. There exist two types of handover: horizontal handover and vertical handover., 4, *4*, 80, 81, 105, 120, 125, 131, 132, 135, 140, 141, 160, 166, 171, 173, 174, 179, 181

Horizontal handover Horizontal handover is a handover between access points of the same technology. An example of a horizontal handover is the movement from one WiFi (802.11g) hotspot to another WiFi (802.11g) hotspot, usually in the same management domain., *5*, 6, 36

Library interpositioning technique Library interpositioning is an implementation technique where function calls to particular library functions are intercepted and redirected to another, intermediary library to handle these functions. This interposed library handles the requested function by executing additional functionality and forwarding the function call to the library for which the call was originally intended. In the context of this work, this technique is often applied to add additional session layer functionality to system calls that are used to establish transport protocol connections, tear down connections, send data and receive data., *34*, 149, 150, 158, 159, 161, 165, 166, 168, 169

Management domain A management domain is an area that consists of a number of network access points, potentially of different access technologies, that

are managed by the same organization. Because these access points are managed by the same organization, it is possible, but not a guarantee, that network layer addresses do not change when moving between access points., 1, 2, 4

Mobility Solution Mobility solution is a general term for system software that address the consequences of mobile behavior, such as address changes, bandwidth changes and longer periods of disconnections. Without a mobility solution, most applications cannot function in a mobile environment., 7, *7*, 8, 11, 13, 15, 18, 19, 23, 26–28, 33–36, 40, 90, 131, 135, 169–171, 177, 178, 181

Network attachment point A network attachment point is a point where a mobile device connects to the access network to obtain communication capabilities. Attachment points are both wired and wireless access points. Attachment points can have different network characteristics with respect to bandwidth, jitter, packet loss, etc., 2, *2*, 4–7, 11–15, 18, 20, 27, 28, 36, 38, 42, 138, 144, 145, 164, 165, 167

Network endpoint or endpoint A network endpoint in this work is an application layer endpoint, more specifically a socket. In traditional protocol stacks implementations, such endpoints behave badly when a mobile device is moved to another network attachment point, because the underlying protocol stack can not cope with the consequences of such mobile behavior: address changes, protocol changes, long periods of disconnection, etc., 2, 12, 13, 16, 21, 23, 26, 29, 30, 33, 35–38, 40, 42, 44–47, 55, 86, 90, 101, 103, 137, 138, 142–150, 152–154, 157–159, 161, 164–166, 169, 181

Network layer address A network layer address is an address that is used by the protocols that exist on the protocol stack's network layer (layer 3). They are usually used to identify a host and are usually part of the identification schemes used to identify a network endpoint., *11*, 12, 16, 21–23, 25–29, 31, 43, 55, 88, 93–96, 108, 119, 135, 138, 141, 143, 172

Physical communication channel A physical communication channel is a transport protocol connection that is used to exchange both application data and protocol headers., 38, 40, 42

Protocol address A protocol address is a part of a generic address used by the AMS. A protocol address is an address that is used as identification mechanism by a particular protocol. For IP, a protocol address is an IP address. For TCP a protocol address is a TCP port., 94–96

Transport layer address A transport layer address is an address that is used by the protocols on layer 4. These addresses are used to identify transport layer connections, often in conjunction with network layer addresses. Transport layer addresses are also referred to in this work as transport protocol ports., *21*, 88, 93, 95, 119, 172

Transport Protocol Connection (TPC) A transport protocol connection is a communication channel, usually between two communicating peers, that is realized by a transport layer protocol. Normally, only transport protocols that offer data stream services (e.g. TCP) establish a connection (virtual circuit). Datagram protocols do not require actual connection establishment and can therefore send data to multiple peer parties without setting up a connection. However, datagram protocols are also often used to exchange data between two peers because the reliability mechanisms of the data stream protocols are too restrictive. For example, streaming applications or computer games often do not require full reliability. Therefore, in this work, a transport layer connection is defined as any type of data exchange that happens between two network endpoints., 7, 21–23, 29, 31, 35, 38, 43, 45–48, 54, 94, 97, 102–104, 109–111, 113, 115, 116, 125, 138, 149, 154, 155, 157, 163, 168, 170, 178–180

Triangular routing Triangular routing is usually a consequence of introducing a proxy system in the network. Network traffic is not sent immediately between peers, but through the intermediate proxy. The result is that network traffic usually does not follow the most optimal route between two communicating partners because of the involvement of the third party, hence the term triangular routing. Mobile IP is an example technology t hat introduces triangular routing in the network because all the data that is destined for the mobile host must be routed through the home agent. Improvements to Mobile IP contain optimizations so bypass the home agent., 27, 139, *139*, 141, 143, 147

Unanticipated session suspension Unanticipated session suspension occurs in case of unexpected network access loss. There are no means to inform the corresponding CAS that a session will be suspended., 51, *51*, 62, 63, 65–69, 74, 75, 82, 89, 111, 113, 127, 131, 178

Vertical handover A vertical handover is a handover between access points of a different technology. For example, a vertical handover from a WiFi (802.11g) hotspot to a UMTS wireless network., 5, 175

Virtual circuit continuity Virtual circuit continuity [OMTT00] is the capability of a communication system to keep a virtual connection between two

applications alive when the location of the host changes. This includes maintaining byte stream consistency and the ability to cope with longer periods of disconnection. Longer periods of disconnection are usually fatal to traditional virtual circuit based protocols like TCP because of timeouts that occur when the peer communication endpoint is not responding in time., 18, *18*, 21, 24–26, 143, 150–152, 154–156, 158, 160, 161, 163–165, 168, 169, 171

Bibliography

[ABKM01] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 131–145, Banff, Canada, October 2001.

[Alc03] Alcatel Bell and K.U.Leuven. Towards Service Centric Access Networks (SCAN), IWT research project #010319, August 2001–2003.

[App06] Apple Computer, Inc. Quicktime home page. http://developer.apple.com/quicktime/, 2006.

[AWSBL99] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The Design and Implementation of an Intentional Naming System. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, pages 186–201, Kiawah Island, SC, USA, December 1999.

[BB95] A. Bakre and B. R. Badrinath. I-TCP: Indirect TCP for Mobile Hosts. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS)*, pages 136–143, May 1995.

[BB97] A. V. Bakre and B. R. Badrinath. Implementation and Performance Evaluation of Indirect TCP. *IEEE Transactions on Computers*, 46(3):260–278, March 1997.

[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.

[BHT+03] S. Burleigh, A. Hooke, L. Torgerson, K. Fall, V. Cerf, B. Durst, K. Scott, and H. Weiss. Delay-Tolerant Networking: an Approach to Interplanetary Internet. *IEEE Communications Magazine*, 41:128–136, 2003.

[Bin01] J. Binkley. An Integrated IPSec and Mobile-IP for FreeBSD. Technical Report PSU Technical Report 01-10, Portland State University, October 2001.

[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.

[BMS02] P. Bhagwat, D. A. Maltz, and A. Segall. MSOCKS+: an Architecture for Transport Layer Mobility. *Computer Networks*, 39(4):385–403, July 2002.

[BPT96] P. Bhagwat, C. Perkins, and S. Tripathi. Network Layer Mobility: An Architecture and Survey. *IEEE Personal Communications*, 3(3):54–64, June 1996.

[BSAK95] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz. Improving TCP/IP Performance over Wireless Networks. In *Proceedings of the 1st ACM Conference on Mobile Computing and Networking (MOBICOM'95)*, Berkeley, CA, USA, November 1995.

[CB96] S. Cheshire and M. Baker. Internet Mobility 4x4. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures and Protocols for Computer Communications*, pages 318–329, Standford, California, August 1996.

[CBZS98] K. L. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz. Directions in Active Networks. *IEEE Communications Magazine, Special Issue on Programmable Networks*, 36(10):72–78, October 1998.

[CDK+99] A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. B. Vicente, and D. Villela. A Survey of Programmable Networks. *ACM SIGCOMM Computer Communications Review*, 29(2):7–23, April 1999.

[CER96] CERT. CERT Advisory CA-1996-21 TCP SYN flooding and IP spoofing attacks, September 1996.

[CGK+02] A. T. Campbell, J. Gomez, S. Kim, Z. R. Turányi, A. G. Valkó, and C.-Y. Wan. Internet micromobility. *Journal of High Speed Networks*, 11(3–4):177–198, 2002.

[CGKW02] A. T. Campbell, J. Gomez, S. Kim, and C.Y. Wan. Comparison of IP Micromobility Protocols. *IEEE Wireless Communications*, 9(1):72–82, February 2002.

[CI94] R. Cáceres and L. Iftode. The Effects of Mobility on Reliable Transport Protocols. In *Proceedings of the 14th International Conference on Distributed Computing Systems (ICDCS'94)*, pages 12–20, June 1994.

[Cis06] Cisco Systems, Inc. Cisco Mobile VPN – Enabling Cisco End-Device Based IP Mobility. http://www.cisco.com/application/pdf/en/us/guest/products/ps6744/c1244/cdccont_0900aecd803a837c.pdf, February 2006.

[Coh03] B. Cohen. Incentives Build Robustness in BitTorrent. In *The First Workshop on the Economics of Peer-To-Peer Systems*, Berkeley, CA, USA, June 2003.

[DCH+97] O. P. Damani, P. E. Chung, Y. Huang, C. Kintala, and Y.-M. Wang. ONE-IP: Techniques for Hosting a Service on a Cluster of Machines. *Computer Networks*, 29(8-13):1019–1027, 1997.

[DH76] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[DRG] K.U.Leuven DistriNet Research Group. DiPS home page. http://www.cs.kuleuven.be/cwis/research/distrinet/projects/DIPS/.

[Dun03] A. Dunkels. Full TCP/IP for 8-bit Architectures. In *Proceedings of the First International Conference on Mobile Applications, Systems and Services (MOBISYS 2003)*, May 2003.

[DvOW92] W. Diffie, P. C. van Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, 1992.

[Fal03] K. Fall. A Delay-Tolerant Network Architecture for Challenged Internets. In *Proceedings of SIGCOMM'03*, pages 27–34, Karlsruhe, Germany, August 2003.

[FYT97] D. Funato, K. Yasuda, and H. Tokuda. TCP-R: TCP Mobility Support for Continuous Operation. In *Proceedings of the IEEE International Conference on Network Protocols*, pages 229–236, Atlanta, Georgia, October 1997.

[Gam85] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in cryptology*, pages 10–18, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[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.

[GJP04] E. Gustafsson, A. Jonsson, and C. E. Perkins. Mobile IPv4 Regional Registration. http://tools.ietf.org/wg/mobileip/draft-ietf-mobileip-reg-tunnel/draft-ietf-mobileip-reg-tunnel-09.txt, June 2004. Request for Comments, Internet Draft, draft-ietf-mobileip-reg-tunnel-09.txt.

[HSSR99] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. Sip: Session initiation protocol. http://www.ietf.org/rfc/rfc2543.txt, 1999. Request for Comments: 2543 (Category: Standards Track).

[Hua01] W. Hua. Winsock 2: QoS API Fine-Tunes Networked App Throughput and Reliability. *MSDN Magazine*, 16(4), April 2001.

[IDJ91] J. Ioannidis, D. Duchamp, and G. Q. Maguire Jr. IP-based Protocols for Mobile Internetworking. In *Proceedings of the ACM SIGCOMM'91 Symposium on Communications, Architectures and Protocols*, pages 235–245, Zürich, Switzerland, September 1991.

[IJ93] J. Ioannidis and G. Q. Maguire Jr. The Design and Implementation of a Mobile Internetworking Architecture. In *Proceedings of the 1993 Usenix Winter Technical Conference*, pages 489–500, San Diego, CA, USA, January 1993.

[Int06] Internet Assigned Numbers Athority (IANA). IP Address Services. http://www.iana.org/ipaddress/ip-addresses.htm, 2006.

[ISI81] University of Southern California Information Sciences Institute. Transmission control protocol specification (tcp). http://www.ietf.org/rfc/rfc0793.txt, 1981. Request for Comments: 793 (Category: Standards Track).

[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.

[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 (ECOOP-WCOP 2002)*, 2002.

[JPA04] D. Johnson, C. Perkins, and J. Arkko. Mobility Support in IPv6. http://www.ietf.org/rfc/rfc3775.txt, June 2004. Request for Comments: 3775 (Category: Standards Track).

[JTK97] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. Mobile computing with the Rover toolkit. *IEEE Transactions on Computers*, 40(3):337–352, March 1997.

[KA98] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. http://www.ietf.org/rfc/rfc2401.txt, November 1998. Request for Comments: 2401 (Category: Standards Track).

[KB96] R. H. Katz and E. A. Brewer. The Case for Wireless Overlay Networks. In Tomasz Imielinski and Henry F. Korth, editors, *Mobile Computing*, pages 621–650. Kluwer Academic Publishers, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1996.

[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.

[Kob87] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[LGL+96] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS Protocol Version 5. http://www.ietf.org/rfc/rfc1928.txt, 1996. Request for Comments: 1928 (Category: Standards Track).

[LLIS99] B. Landfeldt, T. Larsson, Y. Ismailov, and A. Seneviratne. SLM, A Framework for Session Layer Mobility Management. In *International Conference on Computer Communications and Networks (ICCCN)*, pages 452–456, 1999.

[LS98] P. J. Leach and R. Salz. Uuids and guids. http://www.ics.uci.edu/~ejw/authoring/uuid-guid/draft-leach-uuids-guids-01.txt, 1998.

[Mat99] F. Matthijs. *Component Framework Technology for Protocol Stacks*. PhD thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, December 1999.

[MB97] J. Mysore and V. Bharghavan. A New Multicasting-Based Architecture for Internet Host Mobility. In *Proceedings of ACM MobiCom*, pages 161–72, Budapest, Hungary, September 1997.

[MB98] David A. Maltz and Pravin Bhagwat. MSOCKS: An architecture for transport layer mobility. In *Proceedings of IEEE INFOCOM 1998*, pages 1037–1045, Mar 1998.

[MB99] D. A. Maltz and P. Bhagwat. TCP Splicing for Application Layer Proxy Performance. *Journal of High Speed Networks*, 8(3):235–240, 1999. Also available as IBM Technical Report RC 21139.

[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.

[MDJV04] S. Michiels, L. Desmet, W. Joosen, and P. Verbaeten. The DiPS+ Software Architecture for Self-Healing Protocol Stacks. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture*, Oslo, Norway, June 2004.

[MES95] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 143–155, Copper Mountain Resort, CO, USA, December 1995.

[Mic03] S. Michiels. *Component Framework Technology for Adaptable and Manageable Protocol Stacks*. PhD thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, November 2003.

[MJD+05] S. Michiels, N. Janssens, L. Desmet, T. Mahieu, W. Joosen, and P. Verbaeten. Connecting Embedded Devices Using a Component Platform for Adaptable Protocol Stacks. In C. Atkinson, C.Bunse, H.-G. Gross, and C. Pepper, editors, *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends*, Lecture Notes in Computer Science, pages 185–208. Springer-Verlag, 2005.

[MJV03] T. Mahieu, W. Joosen, and P. Verbaeten. System Support for Dynamic Computer Networks. In *Proceedings of the 1st Flanders Engineering PhD Symposium*, Brussels, Belgium, December 2003.

[MMFR96] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanov. TCP Selective Acknowledgement Options. http://www.ietf.org/rfc/rfc2018.

`txt`, October 1996. Request for Comments: 2018 (Category: Standards Track).

[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.

[MMV04] T. Mahieu, S. Michiels, and P. Verbaeten. Session Layer Services for Vertical Handover in Overlay Networks. In *Wireless 2004, Proceedings*, Calgary, Alberta, Canada, September 2004.

[MP92] M. Mouly and M.-B. Pautet. *The GSM System for Mobile Communications*. Published by the authors, 1992.

[MRA87] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, pages 39–51. ACM Press, November 1987.

[MTI86] T. Matsumoto, Y. Takashima, and H. Imai. On Seeking Smart Public-Key Distribution Systems. *Transactions of the IECE of Japan*, E. 69(2):99–106, Feb 1986.

[MVJ05] T. Mahieu, P. Verbaeten, and W. Joosen. A Session Layer Concept for Overlay Networks. *Wireless Personal Communications*, 35(1–2):111–121, October 2005.

[MvOV97] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press LLC, 2000 N.W. Corporate Blvd., Boca Raton, Florida 33431, USA, 1997.

[MWJV02a] S. Michiels, D. Walravens, N. Janssens, and P. Verbaeten. DiPS: Filling the Gap between System Software and Testing. In *Proceedings of Workshop on Testing in XP (WiTXP2002)*, Alghero, Italy, May 2002.

[MWJV02b] 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.

[Myh03] K. Myhre. Policybased Networking in a Mobile, Multiconnected Environment. Master's thesis, University of Oslo, Department of Informatics, May 2003.

[Net99] Netscape. Persistent Client State HTTP Cookies. http://wp.netscape.com/newsref/std/cookie_spec.html, 1999.

[OMTT00] T. Okoshi, M. Mochizuki, Y. Tobe, and H. Tokuda. Mobilesocket: Session layer continuous operation support for java applications. *IPSJ Journal (DPS)*, 41(6):2573–2584, 2 2000.

[PB94] C. E. Perkins and P. Bhagwat. A Mobile Networking System Based on Internet Protocol. *IEEE Personal Communications*, 1(1):32–41, February 1994.

[Per96] C. Perkins. IP Mobility Support. http://www.ietf.org/rfc/rfc2002.txt, October 1996. Request for Comments: 2002 (Category: Standards Track).

[Per02] C. E. Perkins. IP mobility support for IPv4. http://www.ietf.org/rfc/rfc3220.txt, January 2002. Request for Comments: 3220 (Category: Standards Track).

[PJ01] C. Perkins and D. B. Johnson. Route Optimization in Mobile IP. http://k-lug.org/~griswold/Drafts-RFCs/draft-ietf-mobileip-optim-11.txt, September 2001. Request for Comments, Internet Draft, draft-ietf-mobileip-optim-11.txt.

[PMR99] G. P. Picco, A. L. Murphy, and G-C. Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 368–377, Los Angeles, CA, USA, May 1999. ACM Press.

[QYB97a] X. Qu, J. X. Yu, and R. P. Brent. A Mobile TCP Socket. Joint Computer Science Technical Report Series TR-CS-97-08, The Australian National University, Dept. of Computer Science and Computer Sciences Laboratory, April 1997.

[QYB97b] X. Qu, J. X. Yu, and R.P. Brent. A Mobile TCP Socket. In *Proceedings of the IASTED International Conference on Software Engineering*, November 1997.

[Rah93] M. Rahnema. Overview of the GSM system and protocol architecture. *IEEE Communications Magazine*, 31(4):92–100, April 1993.

[Rea06] RealNetworks, Inc. RealNetworks home page. http://www.realnetworks.com/, 2006.

[RT04] M. Riegel and M. Tuexen. Mobile SCTP. http://www.ietf.org/internet-drafts/draft-riegel-tuexen-mobile-sctp-04.txt, October 2004.

[RVS+02] R. Ramjee, K. Varadhan, L. Salgarelli, S. R. Thuel, S.-Y. Wang, and T. La Porta. Hawaii: a domain-based approach for supporting mobility in wide-area wireless networks. *IEEE/ACM Trans. Netw.*, 10(3):396–410, 2002.

[Sal02] J. M. Salz. TESLA: A transparent, extensible session-layer framework for end-to-end network services. Master's thesis, Massachusetts Institute of Technology, May 2002.

[SB00] A. C. Snoeren and H. Balakrishnan. An End-to-End Approach to Host Mobility. In *Proc. 6th International Conference on Mobile Computing and Networking (MobiCom)*, 2000.

[SBK01] A. C. Snoeren, H. Balakrishnan, and M. F. Kaashoek. Reconsidering Internet Mobility. In *Proceedings 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 41–46, Elmau, Germany, May 2001. IEEE Computer Society Press.

[SCEB05] H. Soliman, C. Castelluccia, K. El Malki, and L. Bellier. Hierarchical Mobile IPv6 Mobility Management. http://www.ietf.org/rfc/rfc4140.txt, August 2005. Request for Comments: 4140.

[SCFJ96] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. http://www.ietf.org/rfc/rfc1889.txt, January 1996. Request for Comments: 1889.

[SG96] M. Shaw and D. Garlan. *Software Architecture - Perspectives on an emerging discipline*. Prentice-Hall, 1996.

[ŞMBV03] I. Şora, F. Matthijs, Y. Berbers, and P. Verbaeten. *Automatic composition of systems from components with anonymous dependencies specified by semantic-unaware properties*, volume 732 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, 2003.

[Sno03] A. C. Snoeren. *A Session-Based Architecture for Internet Mobility*. PhD thesis, Massachusetts Institute of Technology, February 2003.

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov 1984.

[SRX+05] R. Stewart, M. Ramalho, Q. Xie, M. Tuexen, and P. Conrad. Stream Control Transmission Protocol (sctp) Dynamic Address Configuration. http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-addip-sctp-10.txt, January 2005.

[SSI01] F. Sultan, K. Srinivasan, and L. Iftode. Transport layer support for highly-available network services. In *The 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.

[SSII02] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: Connection Migration for Service Continuity over the Internet. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS '02)*, July 2002.

[Ste90] W. R. Stevens. *Unix Network Programming.* Prentice Hall PTR, 1990.

[Ste95] W. R. Stevens. *TCP/IP Illustrated, Volume 2: The Implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[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.

[ŞVB02] I. Şora, P. Verbaeten, and Y. Berbers. Using Component Composition for Self-customizable Systems. In *Proceedings of Workshop On Component-Based Software Engineering: Composing Systems from Components*, pages 23–26. IEEE, 2002.

[SXM+00] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream control transmission protocol. http://www.ietf.org/rfc/rfc2960.txt, 2000.

[Tan96] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, 1996.

[TRB97] S. Thomson, Y. Rkhter, and J. Bound. Dynamic Updates in the Domain Name System. http://www.ietf.org/rfc/rfc2136.txt, April 1997. Request for Comments: 2136 (Category: Standards Track).

[TYT91] F. Teraoka, Y. Yokore, and M. Tokoro. A Network Architecture Providing Host Migration Transparency. In *Proceedings of the ACM SIGCOMM'91 Symposium on Communications, Architectures and Protocols*, pages 209–220, Zürich, Switzerland, September 1991.

[Val99] A. G. Valkó. Cellular IP: a new approach to Internet host mobility. *SIGCOMM Comput. Commun. Rev.*, 29(1):50–65, 1999.

[WC02] D. G. Waddington and F. Chang. Realizing the Transition to IPv6. *IEEE Communications Magazine*, 6(3):138–148, June 2002.

[ZD95] Y. Zhang and S. Dao. A "Persistent Connection" Model for Mobile and Distributed Systems. In *Proceedings of the 4th International Conference on Computer Communications and Networks*, Las Vegas, NV, USA, September 1995.

[ZD97] B. Zenel and D. Duchamp. General purpose proxies: Solved and unsolved problems. In *6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pages 87–92, Cape Cod, Massachusetts, USA, May 1997.

[Zim80] H. Zimmermann. Osi Reference Model — The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions On Communications*, COM-28(4), April 1980.

[ZM02] V. C. Zandy and B. P. Miller. Reliable network connections. In *Proceedings of the eight ACM Internation Conference on Mobile Computing and Networking (MOBICOM '02)*, pages 95–106, Atlanta, GA, USA, Sep 2002.

List of Publications

Articles in international reviewed journals

- "A Session Layer Concept for Overlay Networks." Tom Mahieu, Pierre Verbaeten, and Wouter Joosen. Wireless Personal Communications 35(1-2), pp. 111-121, October, 2005.

Contributions at international conferences, published in proceedings

- "Session Layer services for Vertical Handover in Overlay Networks." Tom Mahieu, Sam Michiels, and Pierre Verbaeten. In *Wireless 2004 Proceedings*, July, 2004.
- "A Session Layer Concept for Overlay Networks." Tom Mahieu, Wouter Joosen, and Pierre Verbaeten. In *Proceedings of The Seventh International Symposium on Wireless Personal Multimedia Communications*, September, 2004.
- "Dynamic Protocol Stack Composition: Protocol Independent Addressing." Sam Michiels, Tom Mahieu, Frank Matthijs, and Pierre Verbaeten. In *Proceedings of The Fourth ECOOP Workshop on Object-Orientation and Operating Systems*, June, 2001.
- "Towards Transparent Hot-Swapping Support for Producer-Consumer Components." Nico Janssens, Sam Michiels, Tom Mahieu, and Pierre Verbaeten. In *Proceedings of the Second International Workshop on Unanticipated Software Evolution*, April, 2003.
- "Self-Adapting Concurrency: The DMonA Architecture." Sam Michiels, Lieven Desmet, Nico Janssens, Tom Mahieu, and Pierre Verbaeten. In *Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)*, November, 2002.
- "Towards Hot-Swappable System Software: The DiPS/CuPS Component Framework." Nico Janssens, Sam Michiels, Tom Mahieu, and Pierre Verbaeten. In *Proceedings of The Seventh International Workshop on Component-Oriented Programming*, June, 2002.

Contributions at other conferences, not published or only as abstract

- "System Support for Dynamic Computer Networks." Tom Mahieu, Wouter Joosen, and Pierre Verbaeten. First Flanders Engineering Phd Symposium,

Brussels, December, 2003.

Parts of books

- "A Component Platform for Flexible Protocol Stacks." Sam Michiels, Nico Janssens, Lieven Desmet, Tom Mahieu, Wouter Joosen, and Pierre Verbaeten. In *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends*, (Atkinson, C. and Bunse, C. and Gross, H-G. and Peper, C., eds.), vol. 3778/2005, pp.185-208, Lecture Notes in Computer Science, Springer-Verlag, GmbH, November, 2005.

Curriculum vitae

Tom Mahieu was born in Tielt (Belgium) on September 14, 1975. He received a Bachelor's degree in computer sciences (Kandidaat Informatica) at the Katholieke Universiteit Leuven Campus Kortrijk (KULAK) and a Master's degree (Licentiaat Informatica) at the Katholieke Universiteit Leuven (K.U.Leuven). In 1997 he graduated at the K.U.Leuven as Licentiaat Informatica, with the thesis "Autonomous agent routing in Correlate" under supervision of Professor Yolande Berbers. The same year, he started working part time for the Domain Name Registration office of the .be top-level domain and was admitted as a part time teaching assistant in the DistriNet (Distributed systems and computer Networks) research group. In 1998 he became a full time research assistant in the Distrinet research group. Besides his doctoral research, his task mainly consists of guiding students during their Master's thesis.