Implementation and Assessment of Modern Host-based Multipath Solutions

Sébastien Barré

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Engineering Sciences

October 22, 2011

ICTEAM
Louvain School of Engineering
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis Committee:

Marcelo **Bagnulo**	UC3M, Madrid
Olivier **Bonaventure** (Advisor)	UCL/ICTEAM, Belgium
Christophe **De Vleeschouwer** (Chair)	UCL/ICTEAM, Belgium
Mark **Handley**	UCL, London
Marc **Lobelle**	UCL/ICTEAM, Belgium
Rolf **Winter**	University of Applied Sciences Augsburg

Implementation and Assessment of Modern Host-based Multipath Solutions
by Sébastien Barré

© Sébastien Barré, 2011
ICTEAM
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

This work was partially supported by a grant from FRIA (Fonds pour la Formation à la Recherche dans l'Industrie et l'Agriculture, rue d'Egmont, 5 - 1000 Bruxelles, Belgium), by the European project Trilogy (ICT-216372) and by the European project ECODE (ICT-223936).

To my wife, parents and sister

“Yahvé, je n’ai pas le coeur fier, ni le regard hautain. Je n’ai pas pris un chemin de grandeurs ni de prodiges qui me dépassent. Non, je tiens mon âme en paix et silence; comme un petit enfant, telle est mon âme en moi. Mets ton espoir, Israël, en Yahvé, dès maintenant et à jamais !”
Psaume 131

“Yahweh, my heart is not haughty, I do not set my sights too high. I have taken no part in great affairs, in wonders beyond my scope. No, I hold myself in quiet and silence, like a little child in its mother’s arms, like a little child, so I keep myself. Let Israel hope in Yahweh henceforth and for ever.”
Psalm 131

Preamble

Since the early days of the Internet, efforts have been undertaken to ensure the reliability of the communications [Cla88]. Survivability in case of failure was indeed a major design priority of the Internet, and the choice of a packet-switched architecture instead of a circuit-switched architecture was a good basis to support that goal. Defining the Internet as a set of gateways that can forward datagrams without keeping any state allowed paths through the Internet to be changed at will, without needing to update the connection state in the communicating hosts in case of path change.

Current research is still ongoing to improve the resilience of networks, both in the fields of inter-domain routing [BFF07] and intra-domain routing [FB07]. But an obvious requirement in any fault-tolerant system is to duplicate the resources, so that the sytem can fallback to the backup resources when the main ones fail. In the Internet the resources are links and routers, and duplicating them only for backup purposes would be expensive. To ensure the profitability of their investments, Internet operators use various Traffic Engineering techniques [L+06] that allow the traffic load to be spread across the available links.

Besides, addressing has increasingly become a problem over recent years. We have now reached a point where the current address pool (for the Internet Protocol version 4) is almost fully depleted [NRO10]. Efforts have been pursued to mitigate the problem, by better dividing the address space [FL06] and sharing one address amongst several machines [SE01]. Although this has been useful for many years, it is not enough anymore. Fortunately a new version of the Internet Protocol (IP version 6) has been designed, and is now in the deployment phase. IPv6 uses 128-bit long addresses, which solves the address exhaustion problem, but also brings the risk for routing tables sizes to increase exponentially, if the address allocation policy is not adapted consequently. In order to achieve efficient IPv6 routing, allocation authorities try to enforce a hierarchical allocation model [CSP+11]. However, an interesting side-effect of this hierarchical allocation model is that the end-hosts, which used to receive in most cases one IPv4 address only, can now be given several IPv6 addresses if their provider is multihomed (that is, connected to more than one upstream provider)[1]. In that case the end-host can use its multiple addresses to

[1] Strictly speaking, this is also true for IPv4, but it is expected to happen more often in the IPv6 case due to the larger available address pool. Moreover, IPv4 stacks are usually not prepared to handle efficiently multiple IPv4 addresses

influence the path taken to reach a peer. A particular case of multihoming is when the host itself uses two interfaces, with one address per interface. In summary, with IPv6, the path control moves from being fully controlled by the network to being partly controlled by the end-hosts. An important consequence is that part of the failure tolerance problem described above must now be handled by the end-host.

After having evaluated many alternatives [dB07, Hus05] to solving the multihoming problem in IPv6, the IETF chartered the `shim6` working group to develop a host-based IPv6 multihoming solution [NB09] that operates at the network layer. That solution has the advantage of being deployable without modifying the core of the Internet. The applications inside the end-hosts also do not need to be upgraded, as everything is managed transparently inside the Operating System.

Finally, several proposals chose to solve the multihoming problem in the transport layer [MK01, HS02, ROA05, ZLK04]. They showed that adding multipath capability to TCP allows to achieve *parallel* use of the available links *for the same TCP flow*. This cannot be done with shim6 because the TCP connections would badly react to the incurred reordering. The transport layer approach to multipath support can hence potentially multiply the experienced goodput by the number of links. Moreover, [WHB08, WRGH11] mention yet another benefit of transport-level multipath: if used at large scale in the Internet, or more likely in the short term, if used within a smaller scale network (e.g. a data-centre [RPB+10]), it can achieve an improved resource utilisation, compared to the techniques deployed today. Such changes to the TCP stack are ambitious however, and none of the aforementioned proposals have been implemented in practice. The SCTP [SXM+00] protocol, by its design, allows easier addition of multipath capability, and this has indeed been implemented in an extension called SCTP-CMT [IAS06]. But SCTP suffers from deployment problems and is not widely supported by current applications. Recently, the IETF chartered the MPTCP working group to develop a solution that would be readily deployable in the current Internet, and would bring immediate benefit.

Thesis statement: Modern multihoming protocols offer improvements to the end-user experience despite an increased complexity of the end-host networking stack, and they are implementable in current operating systems in a modular way.

The contributions of this thesis belong to both the shim6 and MPTCP landscapes, studying and improving multihoming in the network and transport layers. **The main goal of this thesis is to understand the implications of new multihoming protocols on the end-hosts**. We examine these implications from three angles:

- **Usability:** To what extent is a new solution actually useable from the end-user point of view ? In particular some solutions require complex security mechanisms that should be hidden to the user, yet protecting efficiently the communications. Another usability aspect is to examine how efficiently a

solution solves a particular problem. For example, how well can a multihoming protocol solve a mobility problem ? Under what conditions can Multipath TCP improve data-centre communications ?

- **Performance:** New protocols should use a limited amount of system memory and CPU cycles. We also want to ensure that the user experience is actually improved (e.g. short failure recovery time, higher experienced goodputs).

- **Integration with other mechanisms:** This aspect encompasses both protocol and system design. From a protocol point of view, we note that many protocols share functionalities, and would benefit from being combined, so that the combined solution can do more at a lower cost. The same holds for Operating System design. The success of a solution does not only depend on its advantages or integration to existing networks. It also depends on its integration to existing Operating Systems. We develop architectures that allow reusing existing Linux frameworks, but also that can be easily extended to support future protocols.

We use an experimental approach, through real Linux kernel implementations, to study, validate and improve the behaviour of shim6 and MPTCP. We also connect shim6 to a mobility protocol, MIPv6, continuing and extending the work by Bagnulo et al. [BGMA07]. In the context of MPTCP, we propose an architecture that decouples the MPTCP machinery (path selection, congestion control) from the technologies used to detect and manage the available paths across a network. We believe that such an architecture will facilitate the future evolutions of MPTCP to new path management protocols. Our implementations of shim6, *MipShim6* (its mobility-capable companion), and MPTCP are, to our knowledge, the first and most complete implementations available in an Operating System kernel.

Road map

This thesis is organized in five Chapters. **Chapter 1** introduces the building blocks of our work: it presents in particular the concepts of multihoming, mobility as they are currently used today and the IETF proposals that we study and improve in this thesis: MPTCP and shim6.

Chapter 2 goes more in depth into the shim6 protocol. It presents our implementation and evaluation for shim6 and the associated path exploration protocol, REAP. We also show how shim6 can be combined with Mobile IPv6 to enhance it with mobility support.

MPTCP is still a recent protocol, and no general overview has been written yet. We provide one in **Chapter 3**.

In **Chapter 4**, we present the architectural challenges for a Multipath TCP implementation in detail, taking our Linux implementation as a starting point, but emphasising questions that any system implementer needs to solve.

In **Chapter 5**, we evaluate the performance of MPTCP, based on lab measurements. We also present how MPTCP can bring significant benefits in data-centre environments, and present measurement results for this use case.

We conclude the thesis by comparing the two approaches, shim6 and MPTCP, and provide perspectives for future work.

Bibliographic Notes

Most of the work presented in this thesis appeared in previously published conference proceedings and journals. The list of related publications is shown hereafter:

- S. Barré and O. Bonaventure. Implementing SHIM6 using the Linux XFRM framework. In *Routing In Next Generation workshop*, Madrid, Spain, December 2007

- S. Barré and O. Bonaventure. Improved Path Exploration in shim6-based Multihoming. In *SIGCOMM Workshop "IPv6 and the Future of the Internet"*, Kyoto, Japan, August 2007. ACM

- S. Barré, A. Dhraief, N. Montavont, and O. Bonaventure. MipShim6: une approche combinée pour la mobilité et la multi-domiciliation. In *Colloque Francophone sur l'Ingénierie des Protocoles (CFIP)*, 14, pages 113–124. HAL - CCSD, October 2009

- S. Barré, C. Raiciu, O. Bonaventure, and M. Handley. Experimenting with Multipath TCP. In *SIGCOMM 2010 Demo*, September 2010

- C. Raiciu, C. Pluntke, S. Barré, A. Greenhalgh, D. Wischik, and M. Handley. Data Centre Networking with Multipath TCP. In *Workshop on Hot Topics in Networks (HotNets)*, IX, Monterey, California, US, October 2010. ACM

- S. Barré, C. Paasch, and O. Bonaventure. MultiPath TCP: From Theory to Practice. In *IFIP Networking*, May 2011
- S. Barré, J. Ronan, and O. Bonaventure. Implementation and evaluation of the Shim6 protocol in the Linux kernel. *Comput. Commun.*, 34(14):1685–1695, September 2011
- A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural Guidelines for Multipath TCP Development. RFC 6182 (Informational), March 2011
- C. Raiciu, S. Barré, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving data center performance and robustness with multipath TCP. In *SIGCOMM*, Toronto, Canada, August 2011

Several contributions were also submitted to the IETF. Here is the list of corresponding Internet-Drafts :

- S. Barré and O. Bonaventure. Shim6 Implementation Report : LinShim6. Internet draft, draft-barre-shim6-impl-03.txt, expired, September 2009
- S. Barré, C. Paasch, and O. Bonaventure. MultiPath TCP - Guidelines for implementers. Internet draft, draft-barre-mptcp-impl-00.txt, expired, March 2011

Implementations

- *LinShim6*:
  url: `http://inl.info.ucl.ac.be/LinShim6`
  current release : 0.9.1
  size: around 30000 lines in a user space daemon (including around 14000 lines of third-party code, mainly DoCoMo SEND [KWRG04] and UMIP (`http://umip.linux-ipv6.org`), 3556 lines in the Linux kernel.
  references: this publicly available implementation is known to have been referenced by external researchers in the following papers:
  [MKS+07, RBKY08, DM08, RM10, DM09, Mek07, RA08, AKP11]
- Linux MPTCP:
  url: `http://inl.info.ucl.ac.be/mptcp`
  current release : 0.6
  size: around 10000 lines in the Linux kernel. This implementation is now a public open source project, and includes code from other contributors, in particular from Christoph Paasch and Jaakko Korkeaniemi (details on the above URL).
  references: it is known to have been referenced by external researchers in the following papers: [SBS+10, NZNP11, SCW+11, SGTG+11].

Acknowledgements

The fulfilment of this thesis would not have been possible without the support of many people. First, I want to thank my wife, Laurence Delvaux de Fenffe, for her love and her continuous encouragement required to complete this thesis. I am grateful as well to my parents Brigitte Lemasle and Guy Debeck for their support. I would also like to thank the Community of Tibériade at Lavaux-Sainte-Anne; they give me a big support and always help me better understanding how wonderful it is to live with the Catholic faith.

I am also very indebted to Olivier Bonaventure, my advisor, who has guided me throughout this thesis. Through his enthusiasm and unlimited support, he helped me to complete that thesis. He gave me most of the major directions of this work and ensured that I actually turned implementation work into research. I appreciated very much Olivier's very pragmatic approach to networking. Olivier is also an inexhaustible source of networking references. Finally he patiently gave time and contributed to solve last minute problems, in particular when submitting papers, but also when submitting the thesis itself.

My colleagues in the INL team and department have been very precious as well. Bruno Quoitin has been a very nice office neighbour, and shared a passion for low-level programming. I've had many enthusiast discussions about IPv6 with Damien Leroy. Pierre François and Pascal Mérindol had discussions with me on all topics, often very constructive, around a cigar. Christoph Paasch helped me a lot in stabilising the Linux MPTCP implementation and will continue to maintain and improve it with Fabien Duchêne, ensuring this work remains alive. Regarding this implementation, Jaakko Korkeaniemi also took the time to dig into my code and contributed with the IPv6 support.

I am grateful to all my coauthors: Adam Greenhalgh, Costin Raiciu, Alan Ford, Mark Handley, Janardhan Iyengar, John Ronan, Christopher Pluntke, Damon Wischik, Christoph Paasch, Olivier Bonaventure, Amine Dhraief, Nicolas Montavont. I have had long and interesting discussions with most of them, and some of their writing appears in this document.

I also owe a special thank to the University College London Networking team, especially Adam Greenhalgh, Mark Handley and Costin Raiciu (Costin Raiciu is now at the university of Bucharest). Beside being co-autors for several papers, they made available the hen testbed, which is a wonderful tool for networking experiments that involve full kernel implementations. Adam Greenhalgh has been

especially fast in solving problems that inevitably happened with the testbed.

I am thankful to the members of my thesis committee, Marcelo Bagnulo, Mark Handley, Marc Lobelle, Rolf Winter and Christophe De Vleeschouwer who took time to review my thesis and to give me very constructive comments.

Let me also thank the other members (former or present) of Olivier's team for their direct or indirect collaboration, to complete this thesis. Bruno Quoitin, Damien Leroy, Damien Saucez, Benoit Donnet, Pierre François, Christoph Paach, Grégory Detal, Laurent Vanbever, Simon van der Linden, Fabien Duchêne, Virginie Van den Schrieck, Cristel Pelsser, Luigi Ianonne, Sébastien Tandel, Steve Uhlig, Mickaël Hoerdt.

Finally, I would like to thank all the persons who supported me and helped me to have some good time here at UCLouvain and during my thesis in general, as well as my friends who had to face strange reactions from me when I was too involved in my work and started to be less friendly. I also thank Véronique Couvreur, who encouraged me at the end of the thesis and hinted me about the idea to start the document with a Bible quote. After carefully looking for a quote that represents best my state of mind and suits the document, I finally came up with the same quote as her: Psalm 131.

Sébastien Barré
October 22, 2011

Acronyms

ALTO	Application-Layer Traffic Optimization
API	Application Programming Interface
AS	Autonomous System
Back	Binding acknowledgement
BGP	Border Gateway Protocol
BST	Binary Search Tree
BU	Binding Update
CDF	Cumulative Distribution Function
CGA	Cryptographically Generated Address
CIDR	Classless Inter-Domain Routing
CN	Correspondent Node
CoA	Care-of Address
CPU	Central Processing Unit
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DoS	Denial of Service
DSN	Data Sequence Number
DSS	Data Sequence Signal
ECMP	Equal Cost Multi-Path
FD	Full Duplex
FTP	File Transfer Protocol
HA	Home Agent
HBA	Hash Based Address
HIP	Host Identity Protocol
HMAC	Hash-based Message Authentication Code
HoA	Home Address
HTTP	HyperText Transfer Protocol
IAB	Internet Architecture Board
ICMP	Internet Control Message Protocol
IETF	Internet Engineering Task Force
IRTF	Internet Research Task Force
ISN	Initial Sequence Number
ISP	Internet Service Provider
IP	Internet Protocol

IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ILNP	Identifier-Locator Network Protocol
LISP	Locator/Identifier Separation Protocol
MAC	Message Authentication Code
MAC address	Media Access Control address
MIP	Mobile Internet Protocol
MN	Mobile Node
MSS	Maximum Segment Size
MPTCP	MultiPath Transmission Control Protocol
MT	Multipath Transport
NAT	Network Address Translation
NIC	Network Interface Card
PA	Provider Aggregatable
PEP	Performance Enhancing Proxy
PI	Provider Independent
PM	Path Manager
RAM	Random Access Memory
REAP	REAchability Protocol
RFC	Request For Comments
RIR	Regional Internet Registry
RO	Routing Optimisation
RSA	Rivest, Shamir and Adleman
RTO	Retransmission TimeOut
RTT	Round Trip Time
SCTP	Stream Control Transmission Protocol
SHA-1	Secure Hash Algorithm 1
SRTT	Smoothed Round Trip Time
TCP	Transmission Control Protocol
TSO	TCP Segmentation Offload
UDP	User Datagram Protocol
ULID	Upper Layer IDentifier
VPN	Virtual Private Network

Contents

Preamble **i**

Acknowledgements **vii**

Acronyms **ix**

Table of Content **xi**

List of Figures **xv**

List of Tables **xix**

1 Background **1**

1.1 Multihoming 1
1.2 Multihoming in IP version 4 2
1.3 The Internet Protocol Version 6 5
1.3.1 IPv6: core goals 5
1.3.2 IPv6 addressing scheme 7
1.3.3 Neighbour discovery 9
1.3.4 Cryptographically Generated Addresses 10
1.4 Shim6 host-based IPv6 multihoming 12
1.5 Mobility in IPv6: MIPv6 13
1.6 Multihoming in the Transport layer: MPTCP 15
1.7 Conclusions 17

2 Shim6: implementation and evaluation **19**

2.1 Introduction 19
2.2 Shim6 20
2.2.1 Securing locator sets 23
2.2.2 Failure detection and recovery 24
2.3 The LinShim6 implementation 26
2.3.1 The xfrm framework 26
2.3.2 LinShim6 0.9.1 : overall architecture 27
2.3.3 Heuristic for initiating a new shim6 context 30

2.3.4 Incoming packets . . . 33
2.3.5 Keeping one context for each direction . . . 34
2.4 HBA/CGA . . . 34
2.4.1 HBA/CGA address generation . . . 35
2.4.2 Address signature and verification . . . 36
2.5 Improving Shim6 path exploration . . . 38
2.5.1 Validation . . . 39
2.5.2 Exploration time . . . 39
2.5.3 Paths with different delays . . . 41
2.6 Improving failure recovery time . . . 42
2.7 Cost of state maintenance . . . 45
2.8 Combining Shim6 with Mobile IPv6 . . . 47
2.8.1 Movement with MipShim6 . . . 50
2.8.2 Validation . . . 51
2.9 Open Issues with shim6 multihoming . . . 51
2.10 Related Work . . . 53
2.11 Conclusions . . . 54

3 Understanding Multipath TCP 57
3.1 Regular TCP . . . 57
3.2 Multipath protocols . . . 60
3.3 Starting a new MPTCP session . . . 61
3.4 Exchanging data across multiple flows . . . 63
3.5 Terminating an MPTCP connection . . . 65
3.6 Security mechanisms for MPTCP . . . 67
3.7 Conclusions . . . 68

4 Linux-MPTCP: A modular MPTCP implementation 71
4.1 Introduction . . . 71
4.1.1 Terminology . . . 72
4.2 An architecture for Multipath transport . . . 73
4.2.1 MPTCP architecture . . . 74
4.2.2 Structure of the Multipath Transport . . . 76
4.2.3 Structure of the Path Manager . . . 76
4.3 MPTCP challenges for the OS . . . 77
4.3.1 Charging the application for its CPU cycles . . . 78
4.3.2 At connection/subflow establishment . . . 78
4.3.3 Locking strategy . . . 80
4.3.4 Subflow management . . . 84
4.3.5 At the data sink . . . 84
4.3.6 At the data source . . . 89
4.3.7 At connection/subflow termination . . . 97
4.4 Implementing alternative Path Managers . . . 98
4.4.1 Next-hop selection . . . 99

4.4.2 Shim6-based Path Manager 100
4.4.3 Link aggregation 101
4.4.4 Remotely controlled Path Managers 101
4.4.5 Combining Path Managers 103
4.5 Configuring the OS for MPTCP 105
4.5.1 Source address based routing 105
4.5.2 Buffer configuration 108
4.6 Future work 108
4.7 Conclusions 110

5 MPTCP evaluation 113
5.1 Introduction 113
5.2 MPTCP performance 113
5.3 MPTCP congestion control 119
5.4 MPTCP in the data-centre 121
5.4.1 Context 121
5.4.2 Experimental evaluation 124
5.5 Conclusions 127

Conclusion 129

Bibliography 135

List of Figures

1.1 Example AS interconnection 2
1.2 The layered model for the Internet data plane 3
1.3 Multihoming with a PI address block 4
1.4 Multihoming with a PA address block 4
1.5 IPv6 global unicast address format 8
1.6 Automatic address allocation 9
1.7 CGA Address generation 10
1.8 Example Shim6 network configuration 13
1.9 MIPv6 with tunnelling 14
1.10 MIPv6 with Routing Optimisation 14

2.1 Basic operation of a shim6 host 20
2.2 Networking stack with shim6 21
2.3 Shim6 session establishment 22
2.4 Example of failure detection and recovery 25
2.5 Dynamically created path for IPsec packets 27
2.6 Shim6 overall architecture 28
2.7 *LinShim6* example sequence diagram 28
2.8 Flow-based evaluation: Flow duration 31
2.9 Flow-based evaluation: Flow size 31
2.10 Traffic classification 32
2.11 Incoming packet flow 34
2.12 HBA/CGA generation time 36
2.13 HBA/CGA evaluation 37
2.14 Shim6 testbed 1 38
2.15 Evolution of throughput for an iperf TCP session 39
2.16 CDF of exploration times when n paths are broken 40
2.17 Proportions of use for paths with different delays 42
2.18 Shim6 testbed 2 43
2.19 Tsend impact on ART 43
2.20 I2 generation time under high load 45
2.21 Case study of state maintenance on selected servers 46
2.22 Features supported by Mipv6, Shim6 et MipShim6 47
2.23 Architecture of the MipShim6 stack 48

2.24 Double jump scenario with MipShim6 49
2.25 Message sequence during a move 50
2.26 Recovery time after a failure of the Home Agent vs TSend timer . 51
2.27 TCP goodput before/after a double jump 52

3.1 Example TCP exchange 58
3.2 MPTCP is transparent to both the network and the applications . . 61
3.3 MPTCP initial exchange and subflow establishment 62
3.4 MPTCP Data Sequence Numbers (DSNs) 63
3.5 MPTCP retransmission 63
3.6 MPTCP example connection termination 66
3.7 MPTCP authentication 67

4.1 Overview of the multipath architecture 73
4.2 Functional separation of MPTCP in the transport layer 75
4.3 User context socket locking 81
4.4 Releasing a socket locked with *lock_sock()* (user context) 82
4.5 A deadlock may happen if MPTCP uses separate receive windows 86
4.6 Structures used to optimise segment reordering at the receiver . . 87
4.7 Send queue configuration 90
4.8 Send queue configuration 93
4.9 Retransmission mechanism 94
4.10 Retransmission algorithm 95
4.11 Extended Path Manager with per-packet actions 99
4.12 Path management with next-hop selection 100
4.13 Remotely controlled Path Manager 102
4.14 Example instantiation of cascaded Path Managers 103
4.15 Cascaded Path Managers, only the built-in PM is active 104
4.16 Cascaded Path Managers, final setup 104
4.17 Example Routing table configuration for Multipath TCP with two interfaces 107

5.1 Testbed used for the performance evaluation 113
5.2 Impact of the maximum receive buffer size 114
5.3 Impact of packet losses 115
5.4 Testbed with 3 Gigabit links 116
5.5 Impact of the MSS on the performance 116
5.6 Effect of ofo receive algorithms on CPU load 118
5.7 Congestion testbed 119
5.8 Multipath TCP with coupled congestion control behaves like a single TCP connection over a shared bottleneck with respect to regular TCP. 120
5.9 With coupled congestion control on the subflows, Multipath TCP is not unfair to TCP when increasing the number of subflows. . . . 120

5.10 Simple data-centre topology 122
5.11 Clos-based data-centre topology (Al-Fares et al.) 123
5.12 BCube data-centre topology ($k = 1, n = 4$) 124
5.13 Time to transfer short files 125
5.14 12 hours of throughput, all paths between forty EC2 nodes 126

List of Tables

4.1 (event,action) pairs implemented in the built-in PM 77
4.2 (event,action) pairs implemented in the Multipath Transport queue management 96
4.3 Example mapping table for a colouring PM 99
4.4 Mapping table for the colouring Path Manager (depth 1) 105
4.5 Example mapping table for the built-in PM (depth 0) 105

5.1 Optimisations enabled for Figure 5.6 117

Chapter 1

Background

1.1 Multihoming

The core of the Internet is managed by many different stakeholders. They all share the goal of being able to reach each other with maximum efficiency, reliability, and with minimum cost. The glue which makes this possible is the *Border Gateway Protocol* (BGP) [RLH06]. In BGP terminology, the stakeholders are called *Autonomous Systems*. Autonomous Systems (ASes) are defined as a set of network resources (routers, switches, links, ...), that share a single routing policy. Each Autonomous system is allocated a number, called the AS number. BGP allows ASes to offer transit services to others (for money), or establish agreements of free mutual transit[1] (this is called a *peering business relationship* [Gao01]). Peering refers to an AS relationship where the partners agree to exchange traffic from their respective customers free of charge. If, on the other hand, an AS (provider) is paid to offer a transit service to another AS (customer), the agreement is referred to as a *customer-provider relationship* [Gao01]. A simplified Internet is shown in Figure 1.1. The figure shows that some Autonomous Systems (AS6,7,8) provide Internet connectivity to their customers (C1,C2,C3 in the example). They are located at the edge of the Internet and are called *stub ASes*. On the other hand, some ASes provide only transport services to other ASes. They are located at the core of the Internet, and are called *transit ASes*. In this thesis, we will often use the term *ISP* (Internet Service Provider), which refers to an entity (e.g. a stub AS or a transit AS) that provides Internet connectivity to customers (e.g. end-users or other ISPs) [Nor01].

A customer will typically be influenced, in its ISP choice, by the price and the quality (reliability, performance) of the offered Internet connectivity. This requirements translates, for the ISPs, into an attempt to minimize costs and maximize the reliability and efficiency of the data transfers. In Figure 1.1, AS6 is vulnerable to a failure in AS3. Even if several physical links connect AS3 to AS6, a simple

[1] By *free mutual transit*, it is meant that one partner can freely exchange traffic with the other partner or its customers, but not send or receive data from the partner's providers.

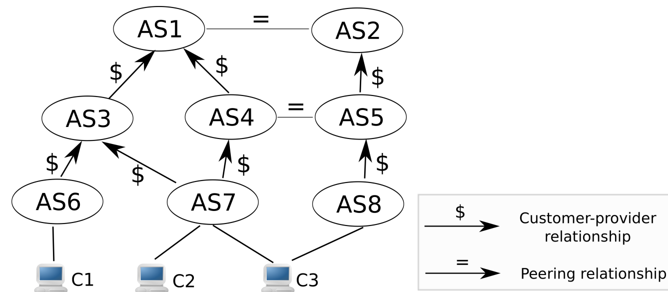

Figure 1.1: Example AS interconnection

configuration error in AS3 could disconnect AS6. In contrast, if AS7 experienced a similar failure to AS3, it can simply redirect its traffic to AS4.

When possible, Autonomous Systems try to negotiate shared cost peerings, like AS4 and AS5. AS4 needs to pay AS1 for the connectivity between AS7 and AS6 (although in this case AS7 will probably use AS4 for reaching AS6 only in case of failure of the AS7-AS3 link). But it can reach AS8 for free (The path AS4-AS1-AS2-AS5-AS8 would have been possible as well, but it is obviously not preferred). Shared cost peerings are possible when the involved ASes provide a similar level of connectivity to other ASes.

In the past, end-users used to be connected to one provider only (e.g. C1 in Figure 1.1). This is changing, and multihoming tends to be present everywhere. Smart-phones are now equipped with 3G and WiFi interfaces. Many companies and even individuals now routinely buy Internet connectivity from two providers (e.g. C3), to improve the resilience of their connection, because a loss of connectivity now becomes increasingly costly for them. Data-centres are designed with many redundant paths, to achieve load balancing and failure resilience. Finally, the core of the Internet is heavily redundant and measurements have shown that per-flow load balancing is a widely used technique [Aug10]. We will come back to the case of C3 later in this chapter.

Currently, most of the end-users still choose only one Internet Service Provider (ISP) and rely on the provider to ensure failure resilience. For example C2 is only connected to AS7, which in turn is connected to AS3 and AS4. AS7 is said to be *multihomed*. In general, **an AS is said to be multihomed if it can provide connectivity to its customers through more than one upstream provider** [de 05]. By extension, in this thesis we will as well qualify C3-like hosts as multihomed. That is, we will call an end-host multihomed if it has Internet connectivity through more than one ISP. In contrast C2 is not considered as multihomed.

1.2 Multihoming in IP version 4

BGP is the protocol that translates business relationships into network connectivity. The Internet Protocol (IP) [Pos81a], is the protocol that transports the data based

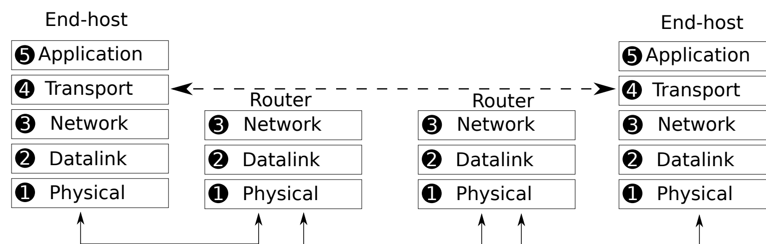

Figure 1.2: The layered model for the Internet data plane

on the reachability information provided by BGP. In other words, BGP forms part of the control plane of the Internet, while IP forms part of the data plane. The data plane, for Internet communications, is organized as shown in Figure 1.2 [ISO94][2]. The layers are interchangeably referenced by their name or layer number. Layers 1 and 2 are related to access mediums (WiFi, 3G, Ethernet, ...). Layer 3 is the network layer, and provides unreliable packet delivery between two end-hosts[3], anywhere in the Internet. Today, the Internet Protocol (IP) is the most widely used layer 3 technology. Layer 4 relies on layer 3 to find the way to a peer, identified by an address (in general, layer n relies on the services provided by layer $n-1$). While the job of the network layer is to find a host in the Internet, the job of the transport layer is to address an application (referenced by a port number) inside the destination. It can also provide a reliable, in order delivery service to the application layer. For instance, the User Datagram Protocol (UDP) [Pos80] guarantees the integrity of the transmitted data (using a checksum), but not the ordering or the reliability. The Transmission Control Protocol [Pos81b], in contrast, offers a stream-based interface to the application. It guarantees the integrity of data (using checksums like UDP), the ordering through sequence numbers, and the reliability by using timers and retransmitting upon loss. Additionnally, TCP is able to dynamically adapt its sending rate to the capacity of the links and the congestion levels, thanks to a congestion control mechanism. Today, more than 95% of the total Internet traffic is driven by TCP or UDP [LIJM$^+$10]. Finally the application layer holds specific protocols tuned to fit the needs of any particular application (e.g., http for web pages, ftp for file transfers, ...).

IP multihoming with Provider Independent address blocks : IPv4 addresses are 32-bits long. An Autonomous System can request to be assigned an address block by its *Regional Internet Registry (RIR)* (RIPE in Europe). If, in the example of Figure 1.1, AS 7 receives the prefix C/24, it will announce it to its two upstream providers AS 3 and AS 4, through BGP. This is shown in Figure 1.3. The announcement is then propagated in the Internet until the world knows that AS 7

[2]The original OSI model defines two additional layers between transport and application (resp. the Session and Presentation layers), but we can safely ignore them in the context of this thesis.

[3]*Unreliable packet delivery* means that a packet may be lost, duplicated or modified on its way to the destination.

can be reached through AS 3 or AS 4. Note, however, that part of the Internet will know only one or the other. For example AS 1 will receive a BGP advertisement for prefix $C/24$ from both AS 3 and AS 4. It will then run the BGP decision algorithm [RLH06, Section 9.1] and only advertise to its neighbours one of the available paths, considered best by the algorithm.

Multihoming with Provider Independent (PI) address blocks is no longer an appropriate technique for two reasons. First, the IPv4 address space is now almost depleted [NRO10], so it is now almost impossible to receive a new PI prefix from the RIR. Second, the use of PI prefixes (as opposed to Provider Aggregatable (PA) prefixes) does not allow prefix aggregation, and hence contributes to the growth of the Internet routing tables [ALD+05].

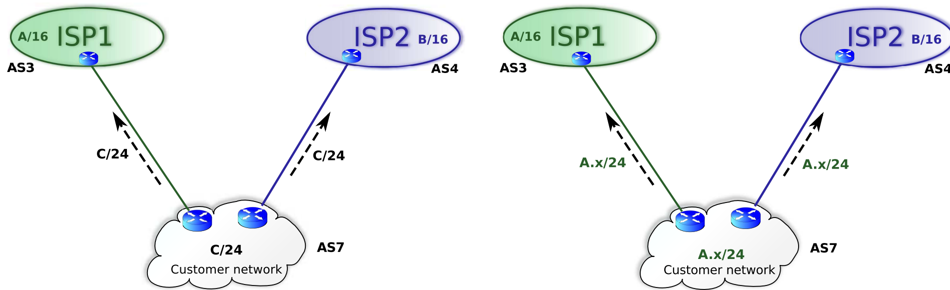

Figure 1.3: Multihoming with a PI address block

Figure 1.4: Multihoming with a PA address block

IP multihoming with Provider Aggregatable addresses : In contrast to Provider Independent addresses, which are received from the RIR, Provider Aggregatable (PA) addresses are received from a provider, as illustrated in Figure 1.4. In that figure, AS 7 receives an address block, $A.x/24$ from provider ISP1 (AS 3). This is a subset of the ISP1 address block (A/16). This mitigates the address depletion problem mentioned above (although ISP1 neither has an infinite pool of addresses, and is eventually dependent on the RIR to get an address block). Besides, PA address allocation allows for better prefix aggregation, hence smaller Internet routing tables. ISP 1 needs to only advertise one prefix ($A/16$) instead of two in the PI case ($A/16$ and propagate $C/24$).

Multihoming somehow breaks this advantage, however. When a site is multihomed, it does not want to ask for an IPv4 prefix from *each* of its providers, because addresses are scarce (hence expensive). Also, current end-host networking stacks are not prepared to efficiently deal with multiple allocated IPv4 addresses. A common IPv4 multihoming technique, for PA-addressed sites, is then to advertise the PA prefix to each peer, just as if it were a PI address block. Unfortunately this contributes to the growth of the routing tables because, in the example of Figure 1.4, ISP2 still needs to propagate the $A.x/24$ prefix. It can also happen that the PA prefix is too long (e.g. longer than a /24), and filtered by the transit ASes. In that

case the multihomed site loses part of the benefits it could otherwise expect from multihoming [ALD+05]. Finally, the customer needs to renumber[4] if it wants to change its main provider (ISP1 in the example).

In both the PI and PA IPv4 multihoming schemes, the multihomed site needs to run BGP and have an AS number. This service is thus not available to home users or small networks.

1.3 The Internet Protocol Version 6

The Internet Protocol version 4 (still most widely in use today) suffers from several problems. The main problem it that its address length, of 32 bits, is too short [NRO10]. Despite several efforts to make more efficient use of the address pool, like Classless Inter-Domain Routing (CIDR) [FL06] or Network Address Translation [SE01] it has effectively depleted. But increasing the address length is a major change in the protocol, and the IPv6 [DH98] designers decided to rethink other aspects of the protocol, that we briefly describe in the first subsection. We then describe several aspects of IPv6 that matter to properly understand this thesis.

1.3.1 IPv6: core goals

Size of the routing tables: The previous section explains that the IPv4 PI addressing scheme favours scattered address allocations, which in turn makes aggregation by the routers located in the core of the Internet impossible. Multihoming makes things even worse by injecting small prefixes in the routing tables. One of the original intentions, in the IPv6 design, was to enforce a hierarchical allocation of addresses, hence forcing the PA allocation scheme in most cases [AAN07].

Unfortunately PA allocation has also drawbacks. In particular, if a PA-addressed client wants to change its provider, it will need to renumber its site. Although research has been conducted to facilitate IPv6 renumbering [LB09], companies have not been convinced about the benefits of PA allocations, and are now asking for PI assignments [Fer11]. Recent revisions of IPv6 allocation policies allow PI allocations, although PA remains the preferred allocation model [CSP+11].

Packet processing in the core: The idea was to allow the heavily loaded core of the Internet to concentrate on its core business: packet forwarding. This can be achieved by pushing state to the edges. To facilitate packet processing in routers, the IPv6 header has been simplified [DH98], and several aspects of the protocol have been removed from the main header and replaced by options located in subsequent headers. For example, fragmentation can now be done only in the end-hosts, while routers were allowed to fragment in IPv4.

[4] *renumbering* is the process of changing the address of each host inside a network, due to the allocated prefix having changed. This usually happens when the prefix is allocated by the provider, and the network administrator decides to change to a different provider.

No more NATs: NATs were initially designed to solve the address allocation problem. With IPv6, this problem is solved as addresses are now four times longer (i.e. four times more bits). Moreover, NATs were seen as dirty network hacks, and it was thought that they should be removed because they go against the end-to-end principle. On the other hand, operators like them, because they "accidentally" have an interesting property from the "security" viewpoint: NATs allow hiding the network topology and services located behind the NAT. They can also be used to facilitate multihoming and avoid renumbering [TZL10]. Solutions have been developed to provide the side-benefits of NATs [dVHD+07] without having the drawbacks, but they have yet to convince the users, who often demonstrate an important inertia in the technology choices. In RFC5902 [TZL10] the IAB has summarised the pros and cons of IPv6 Network Address Translation, without taking a clear position about whether or not it should be used. A hybrid solution has been proposed in [WB11], where only prefix translation is performed. That solution mainly facilitates IPv6 multihoming, but may still break communications that perform referrals (that is, exchanging an IPv6 address in the data flow).

Automatic configuration: With IP version 4, an end-host must configure its addresses itself. Although protocols (the most common one being DHCP [Dro97]) and software allow to automate the process to some extent, it remains common for a user to manipulate IPv4 addresses. A simple example is the connection of two laptops through an Ethernet cable. In that case, there is no DHCP server in place, and the users need to agree on the addresses they will use, and configure them manually. With IPv6, a full address auto-configuration mechanism has been defined [TNJ07], that allows hosts to configure addresses without any help from a user or DHCP. The auto-configuration mechanism relies on a *Neighbour Discovery protocol* [NNSS07], that allows discovering the other hosts in the network (hence replacing the ARP protocol used in IPv4), as well as the available routers (to auto-configure the routing table). Later, an auto-configuration mechanism has also been defined for IPv4 [CAG05], but it is limited to defining link-local addresses. Compared to DHCP, IPv6 address auto-configuration has the advantage of requiring no state in the network, while DHCP needs to maintain a pool of available addresses. Still, there exists an IPv6 version of DHCP [DBV+03], which is useful in cases where there is a desire to centrally manage addresses, but also to easily obtain configuration parameters such as DNS or NTP servers, or to register DNS host names. For an in-depth comparison between IPv6 Neighbour Discovery and the corresponding mechanisms in IPv4, see [NNSS07, Section 3.1].

The IPv6 autoconfiguration process is described in Section 1.3.3.

Improved mobility, multihoming and security: Support for mobility has been added to IPv4 [Per10] long after IPv4 itself had been designed. This created several constraints that limited the flexibility and efficiency of the Mobile IP architecture [Ciz05]. In particular it was not possible to bidirectionnally communicate

without using tunnelling to a middlebox (called the Home Agent), which increases the distance between hosts and inserts a single point of failure in the path. In IPv6, Mobile IP is fully integrated in the protocol [JPA04], and includes a *Routing Optimisation* service, that provides the option to communicate directly between two hosts, without passing through a Home Agent[5].

IPv6 also intended to improve the way multihoming was performed. Section 1.2 describes the current IPv4 multihoming practice, and emphasises that it contributes to a size increase for the routing tables in the core of the Internet, in addition to requiring the multihomed network to own an AS number. The most deployable IPv6 multihoming mechanism is probably shim6 [NB09], although other mechanisms have been proposed that generally require deeper changes of the Internet architecture (e.g. HIP [MNJH08], ILNP [Atk11], LISP [FFML11]). The shim6 protocol is described in more detail in Section 1.4.

At the IP layer (both v4 and v6), the IPsec protocol [KS05] provides authentication and encryption services for IP communications. In IPv4 this was an optional addition to the protocol, but it has been made a requirement for any IPv6 implementation [Lou06, Section 8.1] (it has been recently proposed, however, to modify this to a recommendation instead of a requirement [JLN11]).

1.3.2 IPv6 addressing scheme

Notation: As IPv6 addresses are 128-bits long, it has been necessary to define a different notation, compared to IPv4. An IPv6 address is composed of 8 groups of 4 hexadecimal digits, separated by a colon, as illustrated in the following example: `1234:0000:0000:0000:5678:9ABC:0000:0023`. As this is very long, a short notation has been defined. First, a set of consecutive zeros can be omitted in the writing. However only one such set can be omitted, to avoid ambiguities. Second, in any four-digits group, leading zeros can be omitted as well. Applying this rule, the above address can also be written: `1234::5678:9ABC:0:23`. Prefixes are written the same as in IPv4, e.g. `1234::/16`.

Special addresses: Just like in IPv4, some addresses or prefixes are reserved for special purposes:

- `::` means “no address”. It can be used only in the source field of a packet and is useful in the autoconfiguration process (when the local address is not known yet).
- `::1` is the loopback address (as was `127.0.0.1` in IPv4).

[5] Routing Optimisation has also been proposed for IPv4 [PJ01], but it never acquired the RFC status, probably due to the requirement in IPv4 to avoid changing the implementation of the correspondent node, for deployment reasons [Ciz05]. IPv6 does not suffer from that limitation because mobility support has been part of the IPv6 design from the beginning.

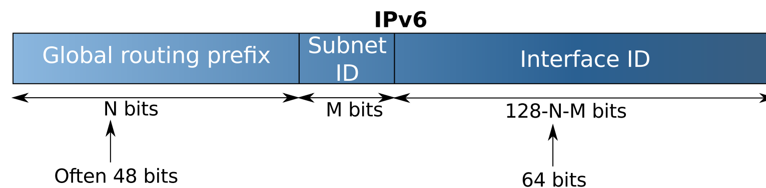

Figure 1.5: IPv6 global unicast address format

- link-local addresses: To be used only for communications with directly connected hosts. Their prefix is `FE80::/64`.
- Other special types: `FF00::/8` is reserved for multicast. `::/96` are used for transition from IPv4 to IPv6 (IPv4-compatible IPv6 addresses [GN00]). They are useful to traverse non-IPv6 network sections (by tunnelling).
  `::FFFF:0:0/96` (IPv4-mapped IPv6 addresses) is also for transition from IPv4 but helps communicating with non-IPv6 end-hosts [HD06].
  `FC00::/7` (Unique Local Addresses or ULAs) [HH05] are the equivalent of IPv4 private addresses.

Global unicast addresses: They are the equivalent of the IPv4 public, routable addresses and are taken from the prefix `2000::/3` [HD06]. A global unicast address is made up of three fields, as shown in Figure 1.5. The first is a global routing prefix, encoded in 48 bits. Then comes the subnet prefix, 16 bits, and finally an interface identifier encoded in 64 bits following the *modified-EUI-64* specification [HD06]. This number references an interface rather than a machine. Thus, each interface has a different address. An interface may otherwise have multiple addresses (e.g. link-local and global unicast). The identifier is generally calculated from the Layer 2 address, such as the MAC address. As this address is encoded in 48 bits, a mechanism is set to convert it to the EUI-64 format [Cra98]. Sometimes it is not possible to use the link-layer address to generate the IPv6 identifier, or it is not desired for confidentiality reasons. In this case it is possible to define an EUI-64, the uniqueness of which is only guaranteed locally, the only constraint being that each IPv6 address must be unique. Hence, an IPv6 identifier need only be unique within the local sub-network (since the 64-bits prefix provides global uniqueness).

Two bits of an EUI-64 identifier have a special meaning:

- Bit ’u’ (Universal / Local): when set to 1, it indicates that the identifier is globally unique, otherwise its uniqueness is guaranteed only locally.
- Bit ’g’ (Individual / group): indicates if the identifier corresponds to a group. This means that the address formed from such an identifier is multicast.

These bits are respectively the seventh and eighth bit from the left of the identifier.

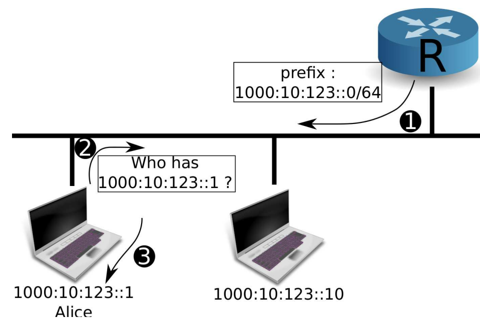

Figure 1.6: Automatic address allocation

1.3.3 Neighbour discovery

Figure 1.6 shows an example of automatic address allocation in IPv6, using the Neighbour Discovery Protocol [NNSS07]. The allocation happens in three steps (numbered in the picture):

1. Router R periodically advertises all prefixes that can be used on the link. In this case it is the prefix `1000:10:123::0/64`. Alice can trigger the sending of this message (Router Advertisement) by sending a Router Sollicitation to the *all routers* multicast address (`FF02::2`).

2. Alice generates an interface identifier, that is, the 64 least significant bits of the address. Several methods exist to generate this identifier, see e.g. [Cra98, Aur05, ND01]. Since the interface identifier is generated locally, we must now check its uniqueness (on the link only, because the prefix advertised by the router ensures uniqueness at the Internet level). This step (number 2 in Figure 1.6), is handled by the DAD algorithm (Duplicate Address Detection [TNJ07]): Alice asks if someone already owns the address she wants to take.

3. Since nobody answered, she stores it and can use it.

The above paragraph only illustrates a portion of the Neighbour Discovery protocol, but it can actually do more. Link local addresses allow hosts to communicate locally even in the absence of a router. They are also generated as shown in Figure 1.6, but starting at step 2 and using the prefix `FE80::/64`. Finally, Neighbour Discovery allows nodes to monitor the reachability of others hosts and routers, and find alternate routers when the current one stops being reachable.

Security in Neighbour Discovery: The principle explained above works well if one has confidence in people who connect to the network. It's less and less the case today, notably due to the wide development of wireless hotspots in airports, motorway service stations etc. If we consider Figure 1.6, we observe that a malicious user may intervene in the Duplicate Address Detection algorithm, answering

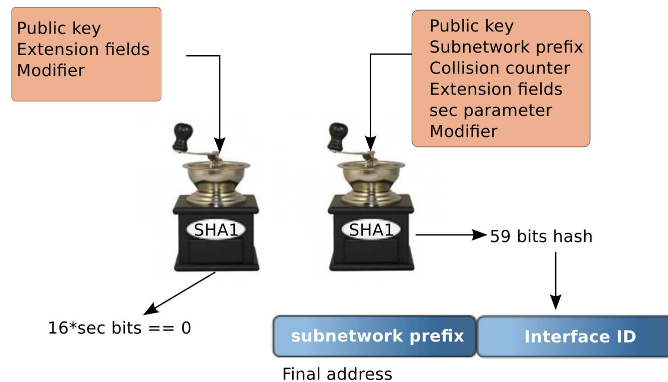

Figure 1.7: CGA Address generation

all DAD queries by claiming he possesses any address. Thus every tentative address is immediately considered as invalid, and Alice is unable to connect. Other examples of possible attacks to Neighbour Discovery can be found in [AAK$^+$02]. The obvious solution to the problem is to sign the Neighbour Discovery protocol messages. This is what is proposed in [AKZN05], which defines SEND (Secure Neighbour Discovery). Messages now include a signature and public key to verify the message. If we take the example above, where Alice is trying to acquire address A, and assuming Bob (whose machine is not shown in the drawing) actually owns this address, it will send a NA (Neighbour Advertisement) message indicating that he already holds the address. His message is signed and the public key is attached. It is observed that the public key is used here to prove, not that the issuer is Bob, but rather that the issuer is the holder of the address. Therefore, rather than relying on a certificate mechanism, it is enough to bind the public key to Bob's address, thus defining a new type of address. This is the main idea behind Cryptographically Generated Addresses (CGAs) [Aur05], which are covered in the next subsection.

1.3.4 Cryptographically Generated Addresses

IPv6 CGAs (Cryptographically Generated Addresses) differ from others only by the 64-bit long interface identifier. As mentioned in Section 1.3.1, this interface identifier can be calculated from the Layer 2 address (e.g. MAC) or generated by another process. The goal here is to bind an address to a public key. For this, a device wishing to use an address will first generate a public/private key pair. The public key is then used as input to calculate a hash, which provides a result of 59 bits. These 59 bits, associated with bits u and g (described in Section 1.3.1) and the three bits of the security parameter (described below) form the 64-bit interface identifiers (see Figure 1.5). The combination of these 64 bits with the 64 bits prefix is the final useable address.

Basing the security on a 59-bits hash is obviously too weak: a brute force attack could crack the hash in 2^{59} iterations. The security (sec) parameter is designed to

overcome this weakness. Depending on the desired security level, the sec parameter can take a value ranging from 0 to 7 in order to *artificially* extend the length of the hash by $sec * 16$ bits. The principle, since only 59 bits are included in the address, is to force one of its generation parameters (modifier, see Figure 1.7), so that the calculation of a hash of this modifier gives 0. Unfortunately, the legitimate address generator must check this constraint in a brute force manner (2^{16*sec} iterations): This is to increment the modifier and recalculate the left-hand hash of Figure 1.7, until the $16 * sec$ most significant bits are zero. This additional cost is acceptable since the calculation is made only once per address, and *needs not be done by the verifier* as the modifier is sent along with the verification parameters.

In summary, the price of a calculation is of 2^{16*sec} iterations for the creator of the address. It requires, for an attacker to be successful, to make a brute force calculation of $2^{59+16*sec}$ iterations. An attacker could also create a dictionary of pre-calculated public/private keypairs, and try to scan networks for a known address. To complicate this type of attack, the subnet prefix is included in the hash computation. This forces an attacker to build one dictionary per subnet (as the hash-result, hence the address, becomes subnet-dependent). The CGA generation algorithm makes collisions improbable but not impossible. The DAD (Duplicate Address Detection) algorithm is still needed to check if an address is already assigned. The technique of CGA address generation takes into account the possibility of collisions, and provides a “collision count”. It is incremented each time someone claims to already own the address, thus providing a new address directly (in one iteration, since only the right-hand hash of Figure 1.7 needs to be recalculated). Note that this parameter can not exceed 2, since the collision count would otherwise facilitate the work of an attacker (who could also try several addresses for the same price) and experiencing a number of collisions above three certainly indicates an anomaly [Aur05].

Finally, the CGA designers have anticipated that one could want to link something else than a public key to an address. For this purpose, an extension field can be attached as an input to calculate the hash. We describe the use of such an extension in Section 2.2.1.

If we come back to the address allocation example (Figure 1.6), adapting it so that it uses CGA addresses, we get the following behaviour (standardized as *SEcure Neighbour Discovery* [AKZN05]):

- Alice gets the prefix `1000:10:123::/64` from router R, and uses it to generate a CGA address. The generated identifier is `cga_a`, which leads, by concatenating the prefix and identifier, to the construction of the address `1000:10:123:0:cga_a`.

- Alice sends a Neighbour Sollicitation to ask if someone already has this address. As Alice does not assert anything, but merely asking for information, it is pointless to sign.

- If a normal user has the same address, he also owns a private key (not neces-

sarily the same as the one that Alice uses) allowing him to sign a NA message (Neighbour Advertisement). The NA message contains not only the address, but also a signature made with the private key associated with the public key that was used to generate the address. All generation parameters are included in the message (i.e. the public key, collision count, modifier and extension field). Alice can verify the signature **and** the binding between the public key and the address. This verifies that the NA originator actually owns the address already, and Alice can increment the collision count to get a new address.

- If an attacker wants to claim the ownership of the address, then he must sign his message with a certain private key. He must find a private key whose associated public key provides the same address according to the generation algorithm described above. This will cost $2^{59+16*sec}$ iterations.

A more detailed description of CGA addresses and the SEND protocol can be found in [Aur03] and [AAK^{+}02], respectively.

1.4 Shim6 host-based IPv6 multihoming

Before delving into the details of shim6, consider that there are at least two scenarios that can provide multihoming. The first type is when a single host has two or more IPv6 addresses assigned to two or more layer-2 interfaces connected to separate networks. This can be the case of a laptop having both WiFi and 3G network interfaces, or servers having multiple Ethernet interfaces. In these cases, the multihomed host would like to either be able to efficiently use both interfaces simultaneously or use a primary interface, with automatic redirection of all packets over another interface upon failure of the primary one.

The second type of multihoming occurs when a campus, corporate or ISP network is attached to two different service providers. In such a network, each host gets an address from each service provider, and is accessible over both. A host in such a multihomed network can select, for itself, the provider to use for a given flow, through appropriate selection of the source address[6]. Shim6 was designed with the latter form of multihoming in mind but also supports the former.

As described previously, in today's IPv4 Internet, when a network is multihomed, it receives one IPv4 address range, and uses BGP to advertise its IPv4 prefix to its upstream providers which, in turn, advertise the network to the global Internet. This contributes to the growth of the BGP routing tables. If a link between the multihomed network and one of its providers fails, BGP re-converges, to ensure that the multihomed network remains reachable via its other providers. However, a network relying on shim6 for its multihoming behaves differently. The main difference from IPv4 multihoming is that each shim6 host has several IPv6

[6]Note that this requires support for source-address based routing in the network, as the source address now carries information on what outgoing path to use.

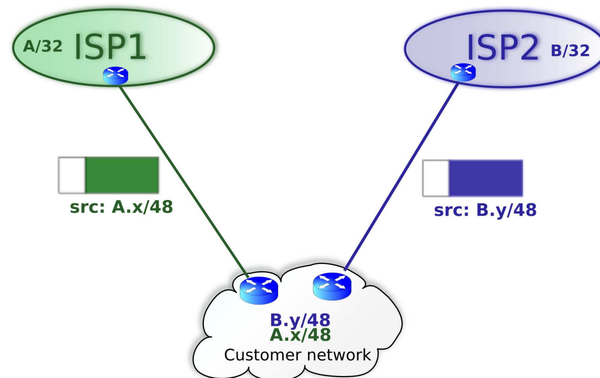

Figure 1.8: Example Shim6 network configuration

addresses, one from each of its providers or one on each of its interfaces. This is illustrated in Figure 1.8. The corporate network shown at the bottom of the figure is attached to `ISP1` and `ISP2`. Each ISP has allocated a prefix to the corporate network. Each shim6 host has one IPv6 address inside each of these subnets. From a BGP routing table viewpoint, the main advantage of shim6 host-based multihoming is that `ISP1` and `ISP2` only need to advertise their global `/32` IPv6 prefix and not the more specific prefixes allocated to their customers. However, this also implies that if the link between the corporate network and `ISP1` fails, BGP will not announce the failure to the global Internet. This problem is solved in shim6 by using a new failure detection and recovery mechanism, the REAP protocol [AvB09], that allows shim6 hosts to detect a failure and switch traffic to an available working path.

1.5 Mobility in IPv6: MIPv6

While multihoming is about making use of several paths that are available *in parallel*, mobility is about making use of paths that are available *sequentially*. The problem is hence not completely different, as we will show in more detail in Section 2.8. In this background chapter, however, we keep describing the default mobility mechanism defined for IPv6: MIPv6.

The goal of Mobile IPv6 (MIPv6) [JPA04] is to ensure session continuity while an end-host is on the move. In a mobile environment, the location of hosts is always changing. At each network change, the mobile nodes must modify the routing of data packets without breaking the ongoing communications. For this, MIPv6 (as does MIPv4) introduces a new element in the network architecture: the Home Agent (HA). The Home Agent assigns IPv6 addresses to hosts present in its network (called the home network). Such an address is called the Home Address (HoA) and represents the permanent identity of the mobile node. This

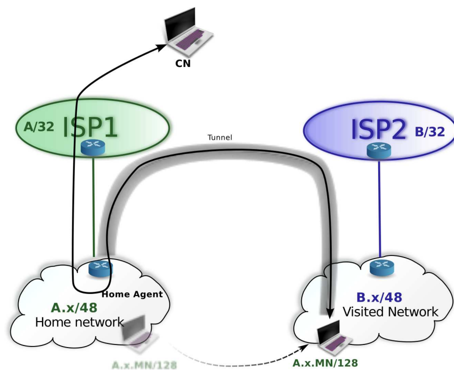

Figure 1.9: MIPv6 with tunnelling

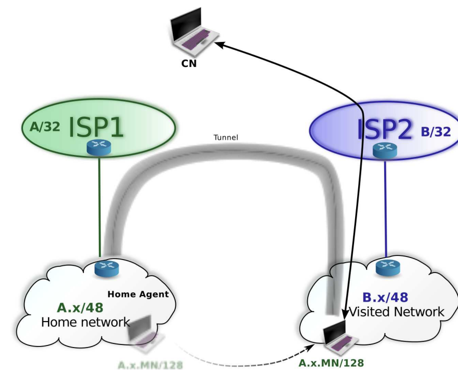

Figure 1.10: MIPv6 with Routing Optimisation

is the address to be used by applications to start communications, even if at the routing level other addresses may be used. During its visit, a mobile node gets temporary addresses (or Care-of Address, CoA) that are topologically correct and can be used for routing packets to the current location of the mobile node.

The first feature provided by the Home Agent is to maintain the mapping between the identity of a node and its current location, that is, between the home and care-of addresses of the mobile node. For this, the HA keeps a cache (called the Binding Cache) updated by the mobile node each time a new address is acquired or after a certain period of time. The update of the Binding Cache is done by exchanging Binding Update (BU) and Binding Ack (Back) messages. The BU message, originated by the mobile node, contains its HoA and its new CoA, and is acknowledged by a Back sent by the HA. The second feature provided by the Home Agent is to capture all traffic destined for the mobile node and relay it to its new location. This is achieved by establishing an IPv6-in-IPv6 tunnel [CD98] between the HA and the new location of the mobile node, as illustrated in Figure 1.9.

The outgoing traffic of the mobile node is also relayed to its correspondents through the Home Agent via the same tunnel. Thus, any exchange between the mobile node and its correspondents passes through the Home Agent, creating a so-called *triangular routing* situation (unless routing optimisation is used, as described in the next paragraph). The third feature provided by the Home Agent is that it can act as a rendez-vous point. Indeed, two mobile nodes can move simultaneously from one network to another one, and still continue to communicate through their respective Home Agent. This is referred to as *double jump*.

One strong point of MIPv6 is that it does not require a corresponding node to implement MIPv6. However, if the corresponding node does also support MIPv6, it has the possibility to use a direct path to the mobile (thus avoiding the triangular routing problem). This is made possible by the MIPv6 *Routing Optimisation* (RO), which consists in updating not only the HA upon a move, but also the corresponding node. No tunnel is needed in this case, as shown in Figure 1.10.

1.6 Multihoming in the Transport layer: MPTCP

We have so far concentrated on network layer multihoming, mainly describing the shim6 protocol. Although shim6 does provide failure recovery capability, it cannot be used as a way to *simultaneously* use several paths for a single transport connection, because the resulting packet reordering would badly impact the TCP congestion control mechanism. To achieve efficient load balancing across multiple paths, a modification of the transport layer is required. The side effect of modifying the transport layer instead of the network layer is that the resulting multipath transport protocol can be used over both IPv4 and IPv6.

Several attempts to do that already happened in the past, first as extensions to the TCP protocol [MK01, HS02, ROA05, ZLK04]. However, to our knowledge these extensions have never been implemented nor deployed. The Stream Control Transmission Protocol (SCTP) [Ste07] protocol was designed with multihoming in mind and supports fail-over. Several SCTP extensions [IAS06, ASL04, LWZ08] enable hosts to use multiple paths at the same time. Although implemented in several operating systems [IAS06], SCTP is still not widely used besides specific applications. The main drawbacks of SCTP on the global Internet are first that application developers need to change their application to use SCTP. Second, various types of middle-boxes such as NATs or firewalls do not understand SCTP and block all SCTP packets.

During the last two years, the MPTCP working group of the IETF has been developing multipath extensions to TCP [FRHB11] that enable hosts to use several paths, possibly through multiple interfaces, to carry the packets that belong to a single connection. This is probably the most ambitious extension to TCP to be standardized within the IETF.

Multipath TCP [FRHB11] is different from existing TCP extensions like the large windows, timestamps or selective acknowledgement extensions. These older extensions defined new options that slightly change the reaction of hosts when they receive segments. Multipath TCP allows a pair of hosts to use several paths to exchange the segments that carry the data of a single connection.

When an application opens a new TCP socket in a multipath-enabled networking stack, the underlying stack actually discovers the number of paths available to reach the peer, and opens as many TCP subflows as its internal heuristic dictates, up to the maximum number of known paths. The detailed establishment procedure for Multipath TCP is described in Section 3.3. The data produced by the client and the server can be sent over any of the subflows that compose a Multipath TCP connection, and if a subflow fails, data may need to be retransmitted over another subflow. For this, Multipath TCP relies on two principles. First, each subflow is equivalent to a normal TCP connection with its own 32-bits sequence numbering space. This is important to allow Multipath TCP to traverse complex middle-boxes like transparent proxies or traffic normalizers. Second, Multipath TCP maintains a 64-bits data sequence numbering space. When a host sends a TCP segment over one subflow, it indicates inside the segment, by using the Data Sequence Signal

(DSS [FRHB11]) option, the mapping between the 64-bits data sequence number and the 32-bits sequence number used by the subflow. Thanks to this mapping, the receiving host can reorder the data received, possibly out-of-sequence over the different subflows. In Multipath TCP, a received segment is acknowledged at two different levels. First, the TCP cumulative or selective acknowledgements are used to acknowledge the reception of the segments on each subflow. Second, a connection-level acknowledgement is returned by the receiving host to provide cumulative acknowledgements at the data sequence level. The same DSS option is used to inform the peer about the connection level sequence number and the connection-level acknowledgement. When a segment is lost, the receiver detects the gap in the received 32-bits sequence number and traditional TCP retransmission mechanisms are triggered to recover from the loss. When a subflow fails, Multipath TCP detects the failure and retransmits the unacknowledged data over another subflow that is still active.

Another important difference between Multipath TCP and regular TCP is the congestion control scheme. Multipath TCP cannot use the standard TCP control scheme without being unfair to normal TCP flows. Consider two hosts sharing a single bottleneck link. If both hosts use regular TCP and open one TCP connection, they should achieve almost the same throughput. If one host opens several subflows for a single Multipath TCP connection that all pass through the bottleneck link, it should not be able to use more than its fair share of the link. This is achieved by the coupled congestion control scheme that is discussed in details in [RHW11, WRGH11]. The standard TCP congestion control [APB09] increases and decreases the congestion window and slow-start threshold upon reception of acknowledgements and detection of losses. The coupled congestion control scheme also relies on a congestion window, but it is updated according to the following principle [RHW11]:

- For each non-duplicate ack on subflow i, increase the congestion window of subflow i by $min(\frac{\alpha * bytes_acked * mss_i}{cwnd_{tot}}, \frac{bytes_acked * mss_i}{cwnd_i})$ (where $cwnd_{tot}$ is the total congestion window of all the subflows and $\alpha = cwnd_{tot}\frac{\max_i(\frac{cwnd_i}{RTT_i^2})}{(\sum_i \frac{cwnd_i}{RTT_i})^2}$).
- Upon detection of a loss on subflow i, decrease the subflow congestion window by $cwnd_i/2$.

The goal of being fair to competing TCP flows is achieved in the above algorithm by constraining the windows in two ways. First, window increases are capped at the increase value that would be applied by a regular TCP flow. This ensures that MPTCP does not take more of the available bandwidth compared to regular TCP flows, on any of its subflows. Second, the α parameter controls the aggressiveness of the increases. Its formula comes from solving the equilibrium equation (where window increases and decreases balance out), under the constraint that any combination of paths cannot take more capacity than a regular TCP flow using the best

of those paths. This prevents a set of multipath subflows sharing a bottleneck link from taking more capacity than a competing regular TCP flow [WRGH11].

1.7 Conclusions

The presence of multiple available network paths is now increasingly becoming the norm in many environments. Smartphones can now connect to the Internet through WiFi or 3G, data-centres use many redundant paths to achieve increased bandwidth and failure tolerance. Even the transition from IPv4 to IPv6 will allow endpoints to choose over which paths to exchange data.

In this chapter, we have set the foundations for this thesis. The central point of this thesis being the use of multiple paths, we have covered the main options to make use of multiple paths in today's Internet. We emphasised that the current most deployed IPv4 multihoming technique (using BGP) is not ideal, given it makes use of the routing system which is not initially designed for that, and requires from a multihomed network to own an AS number.

As IPv4 is going to be progressively replaced by IPv6, we presented the main solution envisaged by the IETF to solve the multihoming problem (without BGP) in the case of IPv6. We note that mobility is also about using multiple paths, the only difference being that the paths are available sequentially instead of simultaneously. We thus presented Mobile IPv6, as we will show later in this thesis an interesting use case for Shim6 as part of an integrated architecture that includes MIPv6 as well.

Finally, we moved one layer up, where multihoming can also be dealt with, although differently. We explained that several efforts had been done in past research to achieve transport layer multihoming, but none of them got to the point of real deployment. We also explained that the most recent of those efforts, MPTCP, is receiving much interest from the IETF community, and is especially interesting in that it is *designed for deployment*, that is, it can be deployed in the current Internet as it is, even robust to many of the existing middleboxes that are more and more placed on Internet paths.

One clear question that arises from this first chapter is *Should we support multiple paths at the transport or at the network layer ?* Working at the network layer seems more natural, and is still the current way of handling multihoming (by using BGP). Shim6 is still located in the network layer, but its functionality is now located in the endpoints instead of the Internet core. From that viewpoint, MPTCP is the continuation of a trend to push multipath control to the edge. Working at the transport layer actually brings impressive additional benefits compared to network layer multihoming. Most noteworthy is the possibility to use the paths *simultaneously*, still handling elegantly the case of shared bottlenecks, by coupling the congestion control of subflows. The other advantage of transport layer multihoming is that failure handling is more timely, as timeouts can be computed based on the available RTT estimations.

In the rest of this thesis, we will show that **modern multihoming protocols offer improvements to the end-user experience despite an increased complexity of the end-host networking stack, and they are implementable in current operating systems in a modular way.** We will present our contributions in both kinds of multihoming, shim6 and MPTCP. We will conclude the thesis with lessons learnt from studying multihoming at the network and transport layers, and briefly compare the benefits and drawbacks of these two approaches.

Chapter 2

Shim6: implementation and evaluation

2.1 Introduction

The shim6 approach relies on an ID/locator split concept, where the mapping is done inside the end-hosts, thanks to a new shim sublayer located in the IPv6 part of the networking stack. If a host owns several locators (IPv6 addresses), an application willing to connect to another end point will use one of them as Upper Layer IDentifier (ULID), and the new shim6 sublayer will provide the ability to change the locators at will while keeping the identifiers constant (rewriting the source and destination address fields on-the-fly). Another protocol, REAP [AvB09] (for REAchability Protocol), adds failure detection and recovery capabilities to shim6. It is able to detect a failure and sends probes to the available locator pairs until a new working path is found, after which the shim6 layer is told to change the currently used locators and the communication can continue without any change in the application.

Because this approach allows a host to change locators during an exchange, it is necessary to provide a means to verify that the used locators are actually owned by the peer. Shim6 can use two mechanisms for that : HBA and CGA. Hash Based Addresses [Bag09] are a set of addresses linked together, so that one can verify that two addresses have been generated by the same host. Cryptographically Generated Addresses [Aur05] are a hash of a public key and allow, together with a signature, to verify that the sender of a signed message is the actual owner of the CGA address used. CGA addresses have been detailed in Section 1.3.4. HBA addresses have been proposed in parallel with shim6, although they are expected to be useful in other environments as well [AKZN05]. We describe them in Section 2.2.1.

Since 2006, we published evolving versions of the first publicly available implementation of shim6, *LinShim6*. In this chapter, we present the architecture of the latest version, 0.9.1[1]. The design provides good performance and is easily ex-

[1] `http://inl.info.ucl.ac.be/LinShim6`

tensible to support cooperation with other protocols that, like *LinShim6*, use the xfrm[2] architecture (e.g. IPsec [KME04] and MIPv6 [MN04]).

In this chapter, we first provide more details of the shim6 protocol and our implementation of it. Then we provide a detailed evaluation of many aspects of the protocol, including security, efficiency, failure recovery and mobility. We close the chapter with a summary of the open issues and related work.

2.2 Shim6

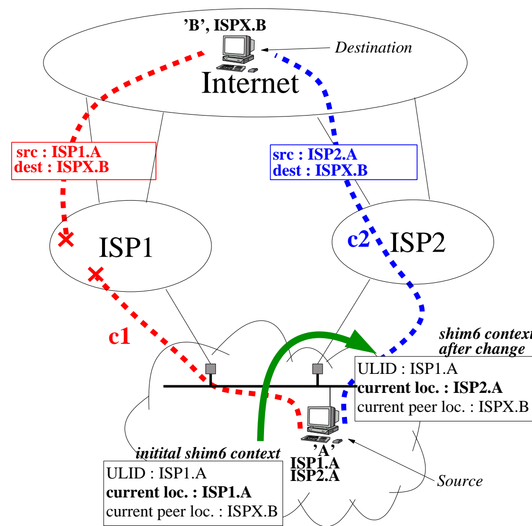

Figure 2.1: Basic operation of a shim6 host

A shim6 host has several IPv6 addresses. All these addresses are locators, i.e. they identify where a network interface is located within the global routing context. For example, in Figure 2.1, a packet whose destination is `ISP1.A` will be delivered via `ISP1`. On the other hand, a packet whose destination is `ISP2.A` will be delivered via `ISP2`. A current best practice [BS04] recommends that ISPs verify the source address of packets received from their customers: a packet produced by host `A` that contains `ISP1.A` as its source address must always be sent via `ISP1`. Such a packet will never be forwarded by `ISP2` if it implements [BS04].

When an application on host A contacts an application on host B using an upper-layer protocol (ULP), the default address selected [Dra03] by host A is determined to be the upper-layer identifier (ULID) to identify the transport flows between the hosts. Conceptually, the shim6 sublayer belongs to the network layer and

[2]xfrm stands for *transform*. It is a modular architecture that allows dynamically inserting sublayers in the networking stack, and was originally designed for IPSec

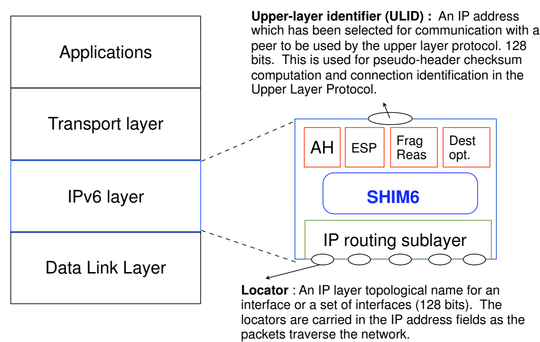

Figure 2.2: Networking stack with shim6

the locators are attached to the lower part of the network layer while the identifier is attached to the upper part of the network layer (Figure 2.2).

The main purpose of shim6 is to preserve established flows despite of network failures, while operating transparently to upper-layer protocols such as TCP or UDP. This is illustrated in Figure 2.1. Host A has established a flow between ULID `ISP1.A` and destination `ISPX.B`. In addition to its ULID, host A also has the `ISP2.A` locator. Upon failure of the path between `ISP1.A` and `ISPX.B`, host A will use shim6 to switch its flow on the `ISP2.A`→`ISPX.B` path. For this, all of host A's packets destined to `ISPX.B` must be sent from source `ISP2.A`. Shim6 ensures the transparency of this operation to the applications.

The shim6 sublayer performs three different tasks. Firstly, two communicating shim6 hosts need to discover their respective locator sets. This is performed during the establishment of the shim6 session. Secondly, during the lifetime of a flow, it may be necessary to switch from the current path to an alternate, e.g. after a failure. Thirdly, shim6 can be used to advertise any change in the set of locators available on a host.

Discovering locator sets: This happens at the beginning of a communication. When an application is requested to initiate an exchange towards a host (i.e. a http or other such request), the usual process is that its name is looked up from the Domain Name System (DNS). The DNS answers with one or several addresses. The application then initiates a connection with one of the obtained addresses (through default address selection) [Dra03, MFHK08a, MFHK08b].

A heuristic on one of the shim6-enabled hosts determines whether it is worth the extra shim6 overhead to protect the communication flow. In the case where the host decides that it is worth the effort, the end hosts communicate to each other their entire set of locators. This is the shim6 initial exchange. After this negotiation, each host has a set of local and peer addresses that it can use to exchange packets.

The establishment of a shim6 session is performed by using a four-way handshake as shown in Figure 2.3. This handshake is based on the handshake used by HIP [MNJH08]. It was designed [NB09] to protect against replay attacks, to en-

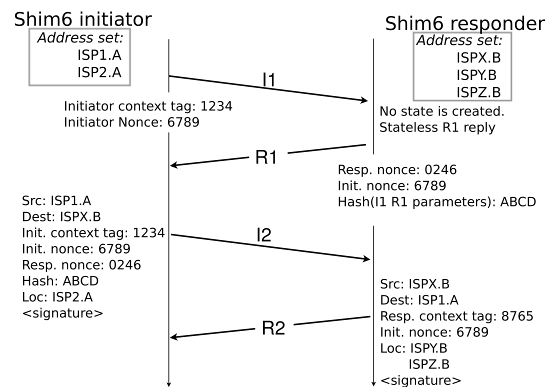

Figure 2.3: Shim6 session establishment

sure that all announced addresses belong to the same peer, and to protect against Denial-of-Service (DoS) attacks. More details about how and why the messages are exchanged this manner are discussed in [NB09, Section 7].

Changing the current path: Typically a path change is triggered when the associated REAP protocol has detected a failure and found an alternate working locator pair. More generally, any appropriately interfaced entity (an application interface for example [KBSS11]) could trigger a path change. Changing the path in the course of a communication is made possible by rewriting the address pair in use. Obviously one particular path is the one corresponding to the ULIDs. In this case the ULIDs and the locators are identical and no rewriting is needed. In all other cases, rewriting is needed and an extension header is added to the outgoing packets. The extension header contains a *context tag* used to identify the flow at the receiver, so that locators can be replaced by the correct ULIDs in the receiver.

The rewriting function of shim6 is located in a new IP-sublayer in the networking stack, as shown in Figure 2.2. Anything located above the shim6 layer sees stable addresses (ULIDs). This includes parts of the IP layer such as IPsec or fragmentation, so that those functions can operate on stable ULIDs, even though shim6 may have had to rewrite the packet header. Conversely, the forwarding functionality of the IP layer must be located below the shim6 layer, so that the locators chosen by shim6 are correctly used to select a path. The effect of address rewriting over the chosen path is illustrated in Figure 2.1.

Locator update: This is useful if a new locator appears after the initial exchange, that is, after the set of locators has been announced by each peer. This could happen should another Ethernet or WiFi interface become operational. If a locator appears or disappears on a host, it is possible to tell the peer about an updated locator set, so that changes in available paths are taken into account. These locator updates are useful in some IP mobility scenarios as we will show in Section 2.8.

The *LinShim6* negotiation involves cryptographic mechanisms. These mechanisms have been carefully designed [NB09] to prevent an attacker from injecting fake addresses, and thus use this attack vector as a basis for new types of attacks. We summarise the critical parts in the next subsection.

2.2.1 Securing locator sets

A key problem faced by host-based techniques that rely on multiple locators is that a receiver must be able to verify the origin of a packet that uses a new locator. RFC4218 [NL05] describes the threats that must be considered while designing any IPv6 multihoming solution in detail. The way shim6 responds to those attacks is described in [NB09, Section 16]. While the solution to many of the threats resides in using well-known protection mechanisms, one particular type of attack, namely address injection, is addressed by a new mechanism that is worth describing here. Address injection consists of an attacker presenting a modified address set to one of the communicating hosts (either by sending fake announcements or modifying existing packets).

The first option proposed by [NB09] to solve this issue is to use Cryptographically Generated Addresses (CGAs) [Aur05]. As explained in Section 1.3.4 this method relies on the use of a signature to prove that all signed addresses have been generated by the same entity. For a given public/private key pair, the private key is used to sign the locator set, while the public key is hashed so as to generate the 64 low order bits of the ULID. Consequently, the security is dependent on an attacker not being able to find a hash collision with a self-generated public key. The time needed to find a collision when the sec parameter is as low as 1 makes such an attack infeasible in short timescales. Over time, when Moore's law does eventually make such an attack practical, or for servers that keep a stable address over time, the attack complexity can be increased further by incrementing the *security parameter* (sec) on the host that generates the signature [Aur05]. With each increase in the *security parameter*, the complexity required to generate a collision will increase by 2^{16*sec} iterations. This increases the cost of address generation, and thus of brute-force attacks, while keeping the cost of address verification constant.

The second option is to bind all addresses together, without using a signature. This type of address is called Hash Based Addresses (HBA) [Bag09]. The 64 low order bits of each address is the result of a hash computation over all the prefixes of the set. An attacker who wishes to inject his own address into the locator set would need to find an input to the hash function that produces, at least, the locator used for forwarding as part of the generated locator set. Since this is made easier by the short length of the hash, HBA uses the same *security parameter* as CGA to tune the cryptographical strength of the locator set.

HBA is computationally cheaper than CGA, but it also has less flexibility. Its main drawback is that the addition of a new address in a locator set requires regenerating the whole set. This is where CGA-compatible HBA addresses are useful. In that case the hash input includes both a public key and the set of prefixes. This

is initially seen by the peer as normal HBA addresses, but if a new prefix must be added afterwards, it can be signed with the public key.

With both these mechanims in place, shim6, as part of the initial context establishment, verifies that the host claiming to be representing `ISP2.A` (for example, see Figure 2.3) can be cryptographically tied to that locator (using either the CGA or HBA mechanisms). While an attacker can generate a new address from a subnet prefix and a public key, this attacker cannot impersonate another hosts address. This is, of course, based on the premise that it is currently beyond the capability of an attacker to harness enough computing power to generate a collision in either the HBA or CGA hash functions.

2.2.2 Failure detection and recovery

REAP is responsible for suggesting to shim6 when to change the current path, as well as for finding an alternate path when the current one becomes unavailable. REAP is closely tied to shim6 because it uses its state to monitor the active flows. REAP can be divided in two main features: *Flow monitoring* and *path exploration*.

Flow monitoring is started immediately after the shim6 initial locator discovery. It is designed in such a way as to minimise the amount of active probing. The main mechanism that allows for reaching that goal is called Forced Bidirectional Detection (FBD). The communication is forced to be bidirectional in the sense that if an end-host receives Upper Layer Protocol (ULP) data, but does not send anything, then control packets (keepalives) are automatically generated. Given this, it can be concluded that a failure has occurred if a host is sending ULP packets without receiving back any data or keepalives[3]. A host decides that a failure has occurred if its *Send Timer* expires. The default expiry time $Tsend$ is defined as $15s$ in [AvB09]. That timer is reset whenever a packet enters the networking stack. In addition to the *Send timer*, a host maintains a *Keepalive Timer*, that sends a keepalive packet on expiry. This is to ensure that the peer does not think that a failure occurred when in fact the application just stopped sending data. The requirement for a *Keepalive Timer* is to have an expiry time that verifies $Tka + \textit{one-way delay} < TSend$. [AvB09] recommends to set Tka as one third of $Tsend$.

The second feature of REAP is path exploration. Due to its flow monitoring capability, REAP can react to failures by probing the known paths (address combinations). The probing process allows for finding an alternate working path, for each direction of the communication. It can even result in the use of different paths for each direction, as it is able to detect unidirectional paths.

REAP relies on a state machine that can be in one of three states : `operational`, `exploring` or `inbound ok`. If the communication is not experiencing any problem, the state is `operational`. This means that end hosts receive either data packets or keepalives from each other. Keepalives are sent if a host has not sent

[3]Note that the number of control packets is kept minimal since no keepalive is needed if data exchange is either bidirectional or paused.

any packet during some time defined as *keepalive interval* (default is 3 seconds). If data traffic stops for a while, keepalives are sent every *keepalive interval*, for the *keepalive timeout* duration (default is 15 seconds as per the RFC). Then no more keepalives are sent until data packets are sent again or the context is destroyed.

The second state defined in [AvB09] is `exploring`. A context reaches that state if a failure has been detected by expiration of the *send timer*. This timer is started when sending a data packet and only if it was not already running. It is reset upon reception of any packet from the peer. Because REAP ensures that the peer will reply with either data or keepalives, not receiving anything from him means that a failure occurred. There are additional ways to detect failures such as indications from upper layers, lower layers or ICMP error messages [AvB09]. Some of these indications may be faster than the timer expiry, but they are not always available.

The third state is named `inbound ok`. A host is in this state if it is receiving packets (either data, keepalives or probes) from its peer, but there is indication that the peer doesn't receive anything. A host may reach the `inbound ok` state from `operational` if it receives an exploring probe from its peer, or from `exploring` if it receives anything from its peer.

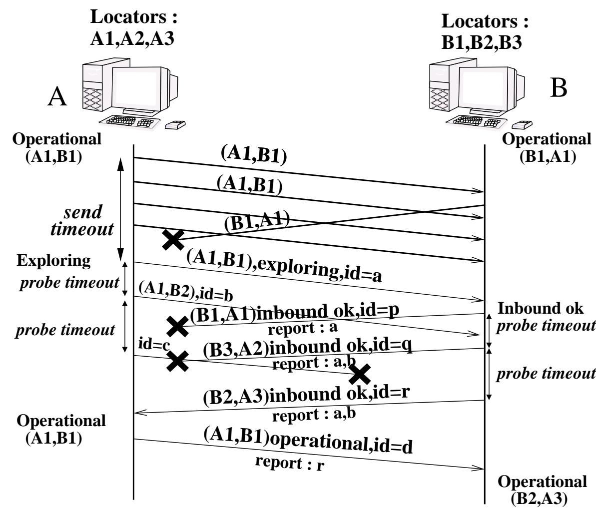

Figure 2.4: Example of failure detection and recovery

A probe contains the state of the sender, a nonce used as identifier and a number of *reports*. A *report* is defined as a summary of a sent or received probe. According to [AvB09] a report contains the source and destination addresses, the nonce and an option field (currently unused). As we will see in the following example, reports of received probes are necessary to learn a new operational path.

Figure 2.4 shows the case of the failure of the path from locator `B1` to locator `A1`. {`A1,A2,A3`} and {`B1,B2,B3`} are the locators assigned to A and B respectively. The first few arrows show the exchange of data packets using locators `A1`

and B1, when the first answer from B is lost. This packet loss triggers a *send timer* expiry inside host A. The consequence is that A switches to the exploring state and starts sending probes. The first one, a (with locators (A1,B1)), goes through, and B learns that its packets no longer reach host A, with the effect of B going to the inbound ok state. After that, both hosts are sending probes along every known path. In our example, all probes from B preceding r are lost. Upon reception, host A reads the reports and learns that probes a and b were successful, that is, locator pairs A1,B1 and A1,B2 are eligible as current locators. The first one is chosen, and a final probe is sent to host B with this new locator pair to announce the switch to the operational state. Because this final probe is sent using a working locator pair, it reaches host B. B learns that its only successful probe has been the one with *id=r*. This means that the only working locator pair is B2,A3. Both hosts update their shim6 context, for address mapping inside the shim6 sublayer, and the conversation continues, without upper layers seeing anything else than some delay.

2.3 The LinShim6 implementation

In this section we provide an overview of our *LinShim6* implementation. As will be shown through measurements, it has been carefully designed to allow good perfomance and modularity. While we only provide a high-level description here, full details can be found in our technical report [Bar08]. The current version (0.9.1) of *LinShim6* contains around 30000 lines of code in the user space daemon (around 14000 of them from third-party code: timer management, hash functions, netlink interface, part of the crypto operations). The kernel side of *LinShim6* has 3556 lines only because it reuses a lot of existing kernel functions and moves functionality to user space whenever possible.

2.3.1 The xfrm framework

Xfrm (for *transformer*) is a network programming framework included in the Linux kernel [WPR+04] to permit flexible transformation of packets. The framework obeys a *Serialized Data State* model, as described by Yoshifuji et al. in [YMN+04].

The idea is to be able to modify the path of packets through the networking stack based on some policies. The framework, originally designed to implement IPsec [KME04], has later been used for the Mobile IPv6 implementation [MN04].

A policy contains a *selector*, a *direction*, an *action* and a *template*. The policy is applied to a packet if it matches the *selector* and is flowing in the *direction* of that policy (inbound or outbound). The selector mechanism allows one to use the addresses, ports, address family and protocol number as fields for the matching (see [KS05, sec. 4.4.1] for the precise semantics of a selector). Now let us assume that a packet matches a given policy. In that case the *template* is used to get a description of the tranformations needed for that kind of packet. Let us further

assume that the packet needs AH (Authentication Header) and ESP (Encapsulating Security Payload) transformations [KS05]. Then the corresponding states (one for AH and the other one for ESP) are found and a linked list of `dst` structures is created. A `dst` structure is normally the result of a routing table lookup, and contains information about the outgoing interface as well as a pointer to the function that must be called to send the packet (for example `ip_output` or `ip6_output`). As shown in figure 2.5, those structures may be linked together, so that several output functions are called sequentially.

After the `dst` path has been created, the linked list is cached for that socket, so that additionnal packets will flow through the IPsec layer as if it was part of the standard networking stack.

While outgoing packets are attached to one or several states by using `dst` entries (as shown in Figure 2.5), incoming packets are processed differently. Since those packets already have extension headers (for example AH and ESP), with lookup keys such as the SPI (Security Parameter Index), it is only necessary to lookup the xfrm states according to the information contained inside each extension header.

Xfrm policies and states are created and managed from user space, with a *key manager* (as called inside the kernel) that communicates with the kernel part of xfrm by using the Netlink API [SKKK03].

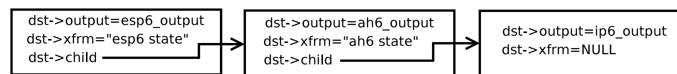

Figure 2.5: Dynamically created path for IPsec packets

2.3.2 LinShim6 0.9.1 : overall architecture

The shim6 mechanism introduces a new sublayer inside the IPv6 layer (below IPsec), similar to AH, ESP or Mobile IPv6. The flexible and modular, yet efficient xfrm framework thus fulfills particularly well the needs of the shim6 transformations: after a shim6 context (negotiated in user space) becomes established, xfrm policies and states are created, so that the packets matching the policies now go through the shim6-handling functions: `shim6_output()` (pointed to by `dst->output`, see Figure 2.5) and `shim6_input()`.

A global view of our design is given in Figure 2.6 (For clarity reasons, the network data flow is not represented and the arrows represent only the control flow). The upper part of the figure runs in user space as a daemon, and currently works with four threads (represented as dashed boxes). Figure 2.7 clarifies the particular ations (described below) undertaken by each of the modules presented in Figure 2.6, based on a common scenario.

One thread is the `telnet server` that provides a Command Line Interface (CLI) to the daemon. The second one is the `timer` thread, that wakes up each

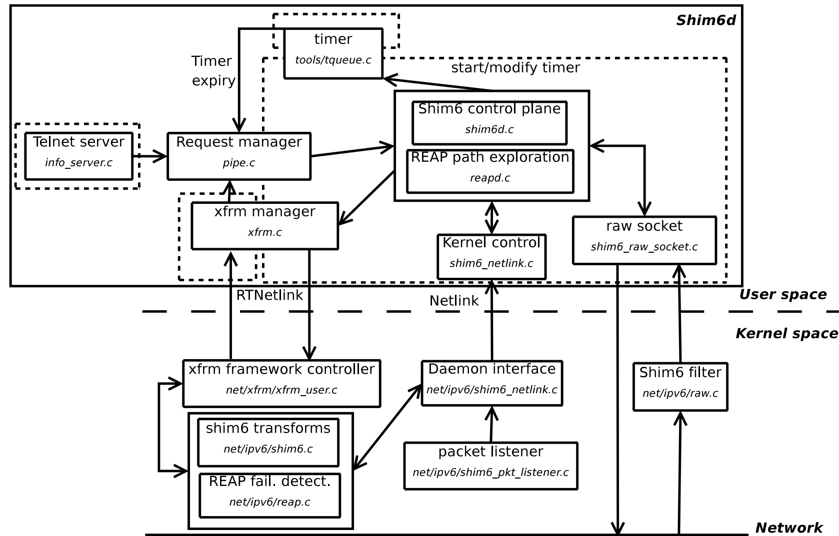

Figure 2.6: Shim6 overall architecture

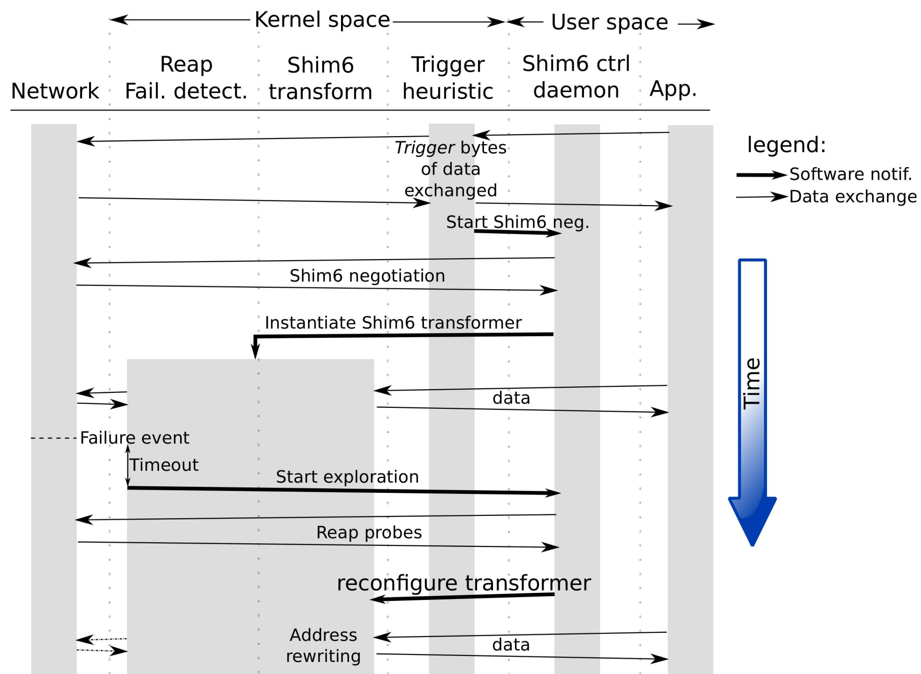

Figure 2.7: *LinShim6* example sequence diagram

time an expiration event happens. The `XFRM manager` listens to messages from the xfrm framework. Finally the main thread listens to messages from the network or the kernel, and reacts appropriately.

One of the major elements of this architecture is the `request manager`. Its role is to simplify concurrency problems caused by external threads[4] (that is, all threads except the main one) wanting to access shim6 data structures (also used by the main thread). For example the `telnet` thread may want to dump the shim6 states, or the `timer` thread may want to access the appropriate context before sending a probe. This led in earlier versions of *LinShim6* to complex mutex schemes, that did not improve the concurrency, since each event was handled by only a few lines of code.

A better scheme for avoiding concurrent access to critical data structures (contexts and hashtables) is to prohibit direct access to those structures from threads other than the main one. Instead, a generic `Request manager` has been written, so that external threads now send a request for service through a pipe. The main thread then performs the service as soon as it is ready. If several requests are sent concurrently, they are queued inside the pipe, hence implementing a message-passing concurrency model. Note that the major benefit of this approach is not an increased efficiency, given that in both cases the problem to solve is to ensure sequential access to ressources that are accessed from different threads. Instead, the benefit is *increased encapsulation*, which in turn improves the overall stability. However, this is efficient only if the message processing time is low for whatever kind of request message (otherwise the response time would increase due to queueing). Fortunately this assumption is true with the shim6 daemon: Each individual action consists only in a single read and/or write to the kernel or the network, with a few memory accesses.

When a new shim6 session starts, packets flow through the kernel and are counted by the `packet listener` module. This module uses the netfilter hooks `NF_IP6_LOCAL_IN` and `NF_IP6_LOCAL_OUT` to detect new exchanges and notify the daemon through the Netlink interface when the implemented heuristic decides that it is worth starting a new shim6 session. Our default heuristic, that we further explain in Section 2.3.3, triggers a context establishment when either 2KB of data have been exchanged or one minute has elapsed. This avoids shim6 establishments for small and short flows. Since shim6 works at the IP layer, if several transport flows are started between two ULIDs, only one network flow is seen by the `packet listener` module.

When the `shim6d` daemon is asked to create a new context, a four way handshake is performed, across the `raw socket`, attached to the shim6 protocol (`IPPROTO_SHIM6`). An important point to note is that the same protocol number is used for control and data plane in the shim6 protocol, which means that the raw socket would normally receive any data message equipped with a shim6 extension header. For efficiency reasons, we prevent this by adding a shim6 filter in-

[4]LinShim6 makes use of POSIX threads.

side the raw socket implementation, as already done for several ICMPv6 messages (ICMPv6 filters may be configured from user space through the `ICMP6_FILTER` socket option).

Now suppose that the shim6 (user space) context becomes established. We need to start the failure detection module, and thus make the packets go through the shim6 transformers, `shim6_output()` and `shim6_input()` (*shim6 transforms* box in Figure 2.6). Actually the transformer only performs address rewriting if ULIDs differ from locators. If not, it simply notifies the REAP `Failure detection` module that a packet has been seen. That module maintains the Keepalive and Send timers [AvB09], and notifies the Shim6d userspace daemon if a failure has been detected (*Netlink* arrow in Figure 2.6). The result is that the `REAP path exploration` module starts sending probes across the `raw socket` until a new operational path has been found.

We decided to split the REAP protocol in two parts, respectively for kernel and user space. Again, this is for efficiency reasons: we try to keep as much as possible the protocols in user space, without sacrificing efficiency. But failure detection needs a timer to be updated for each packet sent or received, and thus *cannot* be implemented in user space.

Finally, when a new path has been found, the `XFRM manager` is notified to update the shim6 xfrm states, so that the `Shim6 transform` module now adds the extension header and rewrites the addresses.

2.3.3 Heuristic for initiating a new shim6 context

When starting a new exchange, it is not necessarily a good idea to immediately negotiate a shim6 context. The obvious case is the one of short flows: for short flows, the shim6 control data may take as much of the communication transmitted bytes as the useful data. Worse, since the flow is short, there are very low chances that the shim6 capability to recover from failures is used at all. In this subsection we examine two heuristics (respectively based on the duration and size of the flows), that may help a shim6 host in deciding whether it is worth negotiating a shim6 context.

Figures 2.8 and 2.9 present the CDF of the duration and size of flows observed on our campus network using NetFlow Version 5 traces, taken on a Cisco Catalyst 6509. Netflow [Cla04] is a tool that monitors network traffic and records *flow-level* information (as opposed to e.g. wireshark that records *packet-level* information) about the observed traffic. Flow-level information is enough however to evaluate the number of flows that would benefit from using shim6, and allows much more efficient capture and analysis. NetFlows were collected at the single Internet connection of the campus. The analysis was performed for every TCP, UDP, GRE or ESP packet sent or received (the flows belonging to those categories represent more than 99.9% of the total collected information). Flows of 0 bytes or 0 packets were ignored. A total of $6.4GB$ of NetFlow log files was collected between 10/27/2007

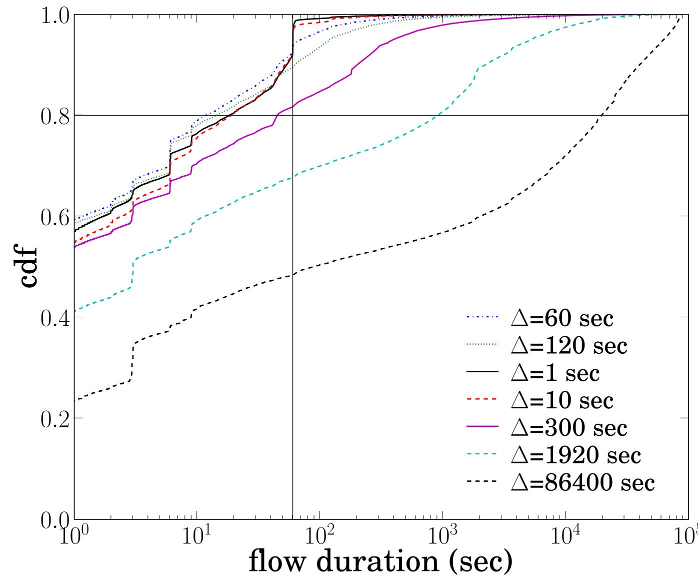

Figure 2.8: Flow-based evaluation: Flow duration

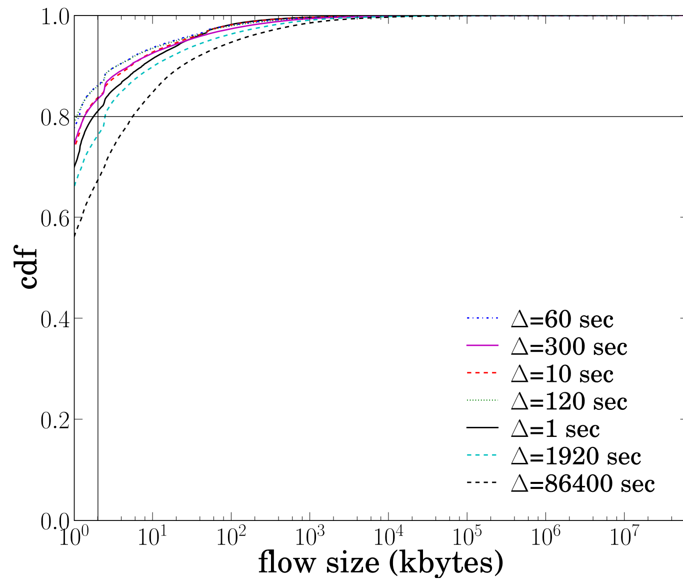

Figure 2.9: Flow-based evaluation: Flow size

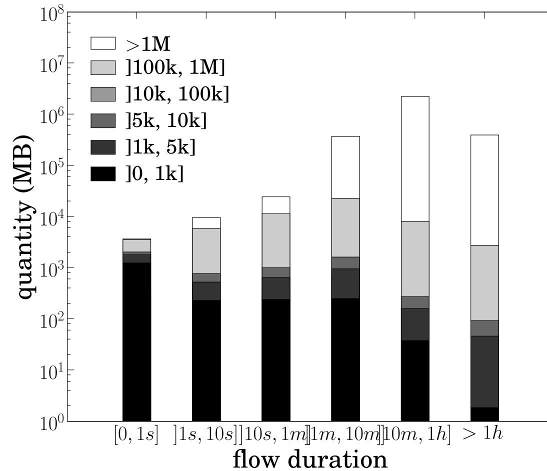

Figure 2.10: Traffic classification

and 11/02/2007[5].

In order to evaluate for what flows to start shim6, we define a flow as follows: a flow is a stream of IP packets with the same source and destination IP addresses, in which packets are never separated by more than Δ seconds, where Δ is a tuning parameter. We consider flows as bi-directional streams of IP packets. Figures 2.8 and 2.9 suggest that it would not be interesting to protect every flow by a shim6 context. Indeed, we can observe that more than 80% of the flows last less than 60 seconds for reasonable Δ values. On the same manner, more than 80% of the flows transmit less than 2KB.

Figure 2.10 shows the kind of flows that carry the majority of bytes in our campus network (Δ is 300 seconds in that figure). That is, for each flow class, sorted by size and duration, the number of bytes pertaining to that class is plotted. While Figures 2.8 and 2.9 show that the majority of the flows are small and short, Figure 2.10 shows that the majority of the bytes are carried by long and heavy flows. This means that a few flows would greatly benefit from shim6 support, with only little overhead. We also observe that if we adequately choose which communication to protect with Shim6, we can both maintain a low global overhead (many small flows will stay unprotected), and enhance the quality and stability of the critical flows. To evaluate the shim6 initialization overhead, we compute it for the realistic case of two peers performing a shim6 negotiation and announcing two CGA locators. This exchange requires the transmission of $1,032$ bytes and represents a 150% overhead for a $2KB$ flow. On the other hand, some flows have a small size, but a long duration. Such flows may need to be protected by shim6, even if the size overhead is high. LinShim6 thus uses a default trigger heuristic that

[5]The NetFlow parameters were as follows: normal aging timeout: 300; fast aging timeout: 60; fast aging packet threshold: 100; long aging timeout: 1920.

starts a negotiation if either a flow is larger than $2KB$, or if it lasts more than one minute. According to the observations from Figure 2.8, this is sufficient to avoid triggering many useless negotiations (actually 80% of the observed flows).

Apart from the size and length considerations, a network administrator may be interested in starting immediately a shim6 negotiation for some flows, or prevent it from being ever triggered for other flows. An example is that DNS requests should not be protected by shim6, while a VPN connection could be protected since its beginning, because it is usually long lived and of high importance.

2.3.4 Incoming packets

As other extension headers, the shim6 extension header is registered in the kernel as a protocol. This allows the standard dispatching function of the Linux kernel (`ip6_input_finish()`) to direct the packet through the correct xfrm state.

The receiving process is illustrated in Figure 2.11 : first the packet is sent to the raw sockets that are listening for that protocol number (left part of the figure). If the next header value in the IPv6 header is `IPPROTO_SHIM6` for example, the packet is delivered to the `raw socket` module of the daemon (if not filtered before).

Next, `ip6_input_finish()` enters a loop that parses each next header and calls the appropriate handler. If a shim6 header is found, the corresponding handler is called and a context tag-based lookup is performed to find an xfrm context.

Packets that do not contain the extension header also need to go through the `shim6_input()` function, since they may be using the ULIDs of an existing shim6 context. This is needed to update the shim6 context timestamp (for garbage collection) and the REAP timers (for failure detection). In that case the standard dispatching function will not send the packet through xfrm, so we do that manually by calling `shim6_input_std()`. Actually in that case we do as if the shim6 header was present, looking at what would be its place regarding extension header order, so as to go through shim6 at the right step (see Nordmark and Bagnulo [NB09, sec. 4.6]). This case is shown in the right part of Figure 2.11.

Note that keepalives and probes are processed by both the `failure detection` (kernel) and `path exploration` (daemon) modules. In that case, the next handler (after function `reap_input()`) is `ipv6_nodata_rcv()` which simply terminates the processing since the next header is `IPPROTO_NONE` (59).

We extended the xfrm lookup functions to support ULID and context tag-based lookups. The current xfrm framework maintains three hashtables : one uses the IPsec Security Parameter Index (SPI) as key [KS05], the second one uses the address pair and the last one uses a request ID (manually configured identifier used by IPsec). Rather than creating new data structures, we use the address pair based lookup for ULIDs, and SPI based lookup to find contexts on the basis of the context tag (the 32 low order bits of the context tag are used for that purpose). This way we can benefit from the performance of hashtable lookups, while keeping the existing data structures.

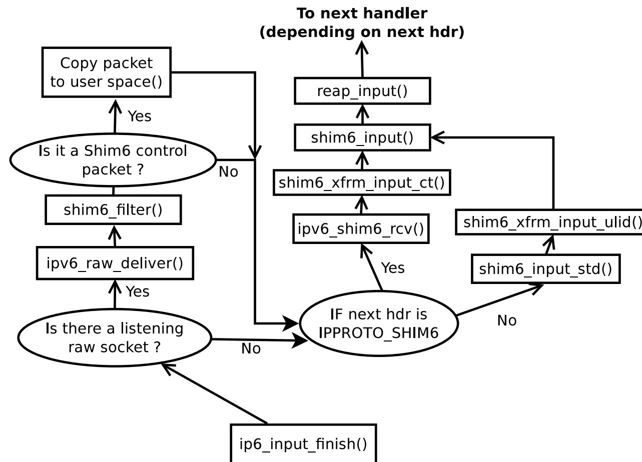

Figure 2.11: Incoming packet flow

2.3.5 Keeping one context for each direction

Since IPsec works on a unidirectional basis, the xfrm framework only supports unidirectional contexts. For this reason, we split the shim6 context into two xfrm contexts. The outbound context stores the peer's context tag and the locator pair (written in each outgoing packet), while the inbound context stores the local context tag and ULID pair.

But this solution is not sufficient for the failure detection module, which really needs a shared data area: when a packet goes out, a timer is started (Send timer). Thus we should maintain a timer structure inside the *outbound* context. But the same timer is stopped when a packet comes in. It means that, when we have one context in hand, we actually need to get the corresponding reverse context also.

We solve this problem by using the private data pointer of an xfrm context. It is private in the sense that its meaning is not known by the xfrm framework and its usage is let to the particular instance of the transformer (shim6). This allows us to do a reverse lookup at context creation only. After that, the shared memory area (only used by REAP) is accessible from both contexts. A reference counter is used to ensure that we free this memory only when the last xfrm context has been destroyed.

2.4 HBA/CGA

CGA and HBA addresses, which are used to secure shim6, are an integral part of the protocol. At the time of writing and to the best of our knowledge, *LinShim6* is the only implementation with full support of HBA and CGA. Supporting these addresses raises operational and performance challenges. From a performance viewpoint, using CGA and HBA addresses may lower the performance of shim6 when

compared to normal IPv6. HBA and CGA operations are the most computationally expensive parts of *LinShim6*. We evaluate here the computational impact of using those addresses, first at generation time (can be done offline), then at signature and generation time (always online).

In the *LinShim6* design, a separate tool allows for CGA/HBA address generation[6]. The tool can be configured to favour either speed of address generation or the strength of generated addresses, incurring extra computational cost for the latter. Also, address generation can optionally be run on a configurable number of parallel processor threads, thus taking advantage of multi-core processors. Much of this address generation code was originally written by DoCoMo [KWRG04] for SEcure Neighbour Discovery [AKZN05], it was integrated into *LinShim6* for the purposes of address generation, and thus it is very similar to the SEND implementation, although our tool features HBA generation as well.

2.4.1 HBA/CGA address generation

As the CGA addresses depend on a public/private key pair, our implementation automatically generates such a key-pair during its installation. This is done so that *LinShim6* will work "out-of-the-box", without complex configuration effort from the user. *LinShim6* configures itself automatically with CGA addresses by using this public/private key-pair. It is, of course, perfectly possible to manually configure several public/private key pairs, and also to define any number of HBA-sets. For *LinShim6*, as CGA addresses can be generated as soon as the host discovers the IPv6 prefix for a network interface, we selected CGA addresses as the default. This implies that CGA addresses are useable on laptops that move regularly, whereas HBA's are not for reasons mentioned previously in Section 2.2.1.

As explained in [Aur05], the cost of the CGA generation is of 2^{16*sec} iterations in the worst case, where sec is the security parameter (a larger sec increases the security but also the time required to generate an address). The same worst case cost applies to HBA generation [Bag09]. The worst case complexity for an attacker to find a matching hash for the address is of the order of $2^{59+16*sec}$ iterations [Aur05], which means that the generator has 2^{59} times less iterations to perform as an advantage in computational cost over the attacker. Consequently, in general, as processing power increases one should consider increasing the value of the security parameter to protect against brute-force attacks.

To evaluate the cost of generating HBA and CGA addresses, we used a Sunblade x6440 equipped with 4 AMD Opteron 8431 processors, each with 6 cores, clocked at 2.4GHz. Figure 2.12 shows the mean time required to generate HBA or CGA addresses, each bar being the mean of 100 trials. Each bar shows the mean generation time of **two** addresses (with different prefixes), in log scale. For this experiment the HBA addresses are CGA-compatible, and both CGA and HBA measurements include the 1024-bit RSA key generation time. The first two sets of

[6]Address generation could potentially be done on a completely separate, more powerful machine

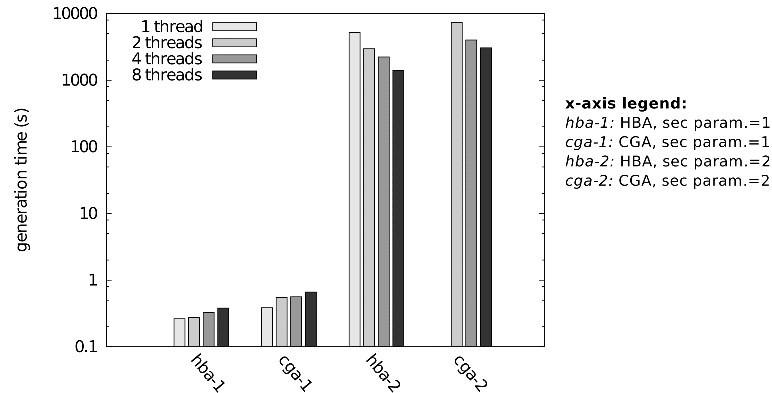

Figure 2.12: HBA/CGA generation time

bars are generated with a security parameter of 1. The other bars were generated with a security parameter of 2. It is worth noting that the standard deviation (not shown in the figure) is very large, because of the brute-force algorithm [Aur05] used in the generation process. For the results presented in Figure 2.12, we observed a standard deviation ranging from 23% to 77% of the reported mean.

The address generation tool [KWRG04] is able to use any number of concurrent threads. This capability was retained and extended in order to support HBA. With the security parameter set to 2, multithreading is necessary in order to obtain a result in a reasonable time. Hence using a security parameter of 1 is the only option on current commodity hardware. When the computational time increases due to a required higher security level, it is clearly beneficial to use as many threads as possible (see the results with 8 threads in the figure, and note that the scale is logarithmic). On the other hand, multithreading gives slightly worse results when the security parameter is 1, because the threading overhead takes a higher proportion of the processing time.

While Figure 2.12 shows the generation time for two addresses, we note that if the number of generated addresses is increased, the CGA generation time increases linearly with the number of addresses. On the other hand, there is a barely perceptible increase in HBA generation time. This is because the expensive part (called *modifier generation* in [Aur05]) is conducted only once for the whole set, in the HBA case.

2.4.2 Address signature and verification

Both operations take place during the initialisation of a shim6 exchange, or when one of the peers announces changes in its locator set. To evaluate the cost of these cryptographical operations we measured the time needed to carry a shim6 negotiation with different security mechanisms. These tests were performed between two hosts on a 100 Mbps Ethernet. The initiator was a Pentium 4 dual core, 2.6GHz with 1GB RAM while the responder was a Pentium 3, 600MHz with 256MB RAM.

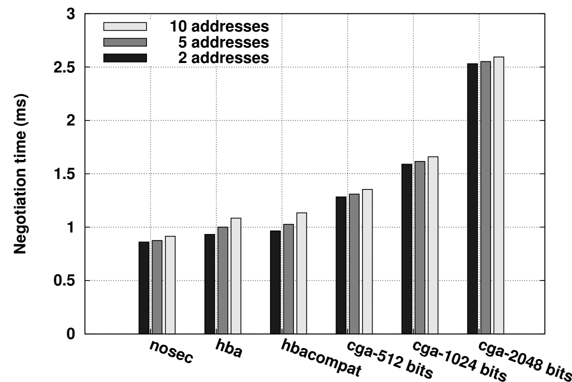

Figure 2.13: HBA/CGA evaluation

The results are reported in Figure 2.13. For each security configuration, the case of each peer announcing 2, 5 or 10 addresses in its locator set is compared. The *negotiation time* is defined as the time elapsed between the transmission of the first I1 message, and the reception of the last negotiation message (R2) (see Figure 2.3). Note that the negotiation time includes two signatures and two verifications, that is, one for each peer. Each bar shows the median negotiation time over 20 consecutive runs.

Looking at the right hand side of Figure 2.13, we note that there is a strong correlation between the length of the RSA key used for signing messages and the negotiation time. Conversely, HBA addresses involve a negotiation time that is almost the same as if no security were used at all. This is because no signature is needed in the case of HBA addresses.

An important consideration, discussed previously, is that HBA addresses require the knowledge of all the prefixes before commencing a shim6 negotiation. This motivates the use of CGA-compatible HBA addresses, defined in [Bag09]. While pure HBA addresses use a random number as input of the SHA-1 hash used during the generation process, CGA-compatible HBA addresses use a public key instead of such a number. However, no signature is needed until the host learns of a new prefix that can be used. At this point a CGA address is generated based on the new prefix, and the key used to generate the previous HBA set. A signed message can be sent to the peer, which will use the already known public key to verify the locator update. In Figure 2.13, the bar labelled *hbacompat* shows the negotiation time needed when an HBA set generated based on a 1024-bit public key is used.

This shows very similar results to the use of pure HBA. The small increase in time is explained by the fact that pure-HBA uses a random number formatted as a 384-bit RSA key, as defined in [Bag09], as opposed to a real 1024-bit key with the CGA-compatible variant. A final observation is that the impact of the number of announced addresses (2, 5 or 10 in the figure) is insignificant compared to the security mechanism used.

From the above observations, we conclude that from a performance point of view, there is a strong argument to be made for using HBA addresses, or even better, CGA-compatible HBA addresses. *LinShim6* allows for the generation of HBA/CGA addresses in advance of their use[7]. Once they are generated, they become active only when configured in the system, either manually or by auto-configuration through the *cgad* daemon. By default, *LinShim6* disables the standard IPv6 auto-configuration mechanism, in order to avoid having both unsecured addresses and HBA/CGA addresses in the system. This mechanism is replaced by the *cgad* daemon, that listens for Router Advertisements, and configures the appropriate addresses when a new prefix is received.

2.5 Improving Shim6 path exploration

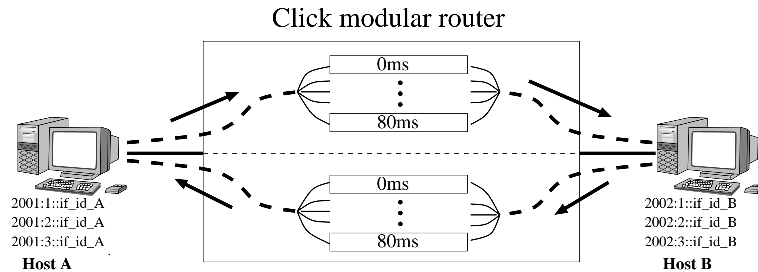

Figure 2.14: Shim6 testbed 1

In this section, we used a testbed composed of three Linux computers. Two of them support shim6, and the third acts as a Click router [KMC+00], used to emulate different paths. The router and one of the end hosts are Pentium II, 300Mhz with 128MB of RAM and 100baseTx-FD Ethernet cards. The other end host is a Pentium Pro 200Mhz with 64MB of RAM and 100baseTx-FD Ethernet cards. Both end hosts run the Linux kernel 2.6.17.11 patched with `shim6/REAP` release 0.4.3. The Click router runs Linux kernel 2.6.16.13 patched with Click release 1.5.0. In order to make measurements faster, the *send timer* has been set to 3 seconds. The setup is shown in Figure 2.14. The router runs the Router Advertisement Daemon[8], that distributes the three prefixes of host A and host B. One Click queue is defined for each possible pair of addresses. Because each shim6 computer receives three prefixes, 9 queues are defined for each direction. Each queue may be configured to be delayed or stopped. Thus we have a total of 18 configurable queues inside the router. This gives the flexibility of simulating unidirectional paths in the Internet, with a configurable delay for each one separately. We may also create a failure in one direction while letting the other direction operational.

[7]By default *LinShim6* will generate CGA addresses on installation

[8]`http://www.litech.org/radvd/`

2.5.1 Validation

To show the benefits of shim6 for a TCP application, we present in Figure 2.15 the effect of a path failure on the throughput of an iperf TCP session. The path is broken approximately 20 seconds after starting the iperf client. The different curves are obtained by artificially adding delay to paths, inside the Click router. This figure shows one of the most important benefits of shim6, that is, transport layer survivability across failures, without any change to TCP. Note that normal TCP/IP is already able to survive if the broken path eventually comes back to life. The difference here is that TCP behaves *as if* the path came back to life, while in fact another path is selected thanks to REAP path exploration. After the recovery, ULIDs are kept constant, while locators are changed, as a result of the path change.

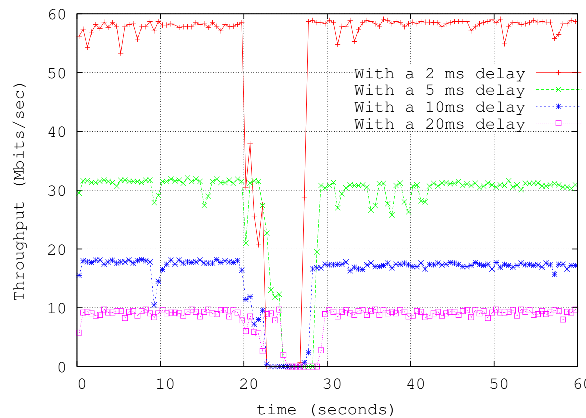

Figure 2.15: Evolution of throughput for an iperf TCP session

The throughput drop of Figure 2.15 represents the full recovery time, including *send timer* expiry (3 seconds)[9]. Now it would be desirable to make this recovery time as short as possible, that is, the throughput drop of Figure 2.15 should be as narrow as possible. This is the subject of the next subsection.

2.5.2 Exploration time

We define the *exploration time* as the duration between leaving and coming back to the REAP `operational` state. This is different from the *detection time*, defined as the interval between the occurrence of a failure, and failure detection by REAP. Finally, the *recovery time* is the sum of the detection and exploration times. The detection time is mainly influenced by the value of the *send timer*. But this timer being started every time a data packet is sent (if not already running), the detection time is also influenced by the frequency of outgoing data packets. For example,

[9]Our lab described in Figure 2.14 uses fixed-length queues regardless of the configured artificial delays. This in turns causes more packet drops for higher artificial delays, hence lower throughput. This however is not related to shim6 as it only enters into action after the failure event.

if one starts an ssh session, then stops activity during some time, keepalives will be sent by shim6 until *keepalive timeout*, but after that no keepalives will be exchanged anymore until ssh becomes active again. In that case the failure will be detected *send timeout* seconds after the first data packet is sent. This is of course a worst case. There is also a best case, which may occur if we can avoid to rely on timers for failure detection (e.g. when the current interface goes down).

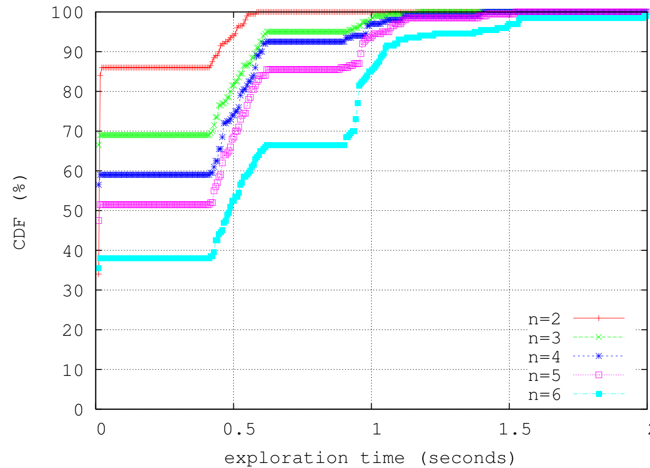

Figure 2.16: CDF of exploration times when n paths are broken

The exploration time depends exponentially on the number of probes sent by each of the peers before finding a successful path for each direction of communication, due to the exponential backoff. It is important here to specify how often a host may send probes, according to [AvB09]. Four initial probes are sent with an interval of 500 ms. Then exponential backoff is started, the interval is doubled each time a probe is sent. When the time interval between probes reaches 60 seconds, exponential backoff is stopped and one probe is sent every 60 seconds. Our implementation introduces an additional 20% jitter to the above intervals, to avoid self-synchronisation [FJ94] (the value of 20% has been fixed arbitrarily). It is also important to note that this version of our implementation selects address pairs by cycling randomly over all possible paths. We present in Section 2.6 another way to cycle over the paths, that tries to probe most distinct paths first.

Figure 2.16 shows the cumulative distribution of exploration times for the testbed described in Section 2.5. Because failure detection requires the existence of a data stream, a UDP client and server have been placed on each host, to establish a UDP bidirectional flow of one small packet every second.

Figure 2.16 has been obtained by measuring exploration times for different values of the number n of broken paths. Broken paths are emulated by tearing down queues inside the Click router. Note that one queue inside the Click router corresponds to a *unidirectional* path from A to B or from B to A, as shown in Figure 2.14. In this experiment we simulate only bidirectional failures, thus we tear

down at least the currently used queues for each simulated failure. Therefore, the minimum value for n is 2. Additional queues (thus additional unidirectional paths) are selected by running through the possible queue combinations, without taking twice the same combination when possible. For each value of n, 200 measurements have been conducted. No artificial delay has been introduced inside queues for that experiment.

The best performance is achieved for $n = 2$ as can be seen in Figure 2.16. We see that if only the current queues are disabled, letting 16 queues enabled, the first probe sent is successful in 86% of the trials. The second probe is always successful. We can conclude this because the initial probes are sent with a 500 ms interval, and 100% of the trials take less than 600 ms (that is, 500 ms plus the 20% jitter).

As we shutdown more and more queues, we decrease the probability for a randomly chosen path to be successful. This gives lower curves as n increases, because each exploration has a higher probability of demanding more probes, thus more time. This is observed in Figure 2.16, where the curves are lower and lower with increasing values of n.

Because the testbed has 18 paths (9 in each direction), we have measured the worst case of breaking 16 paths, letting only one available path for each direction. In that case we obtain a curve with several steps, where each step occurs after one probe interval, due to the exponential backoff. 50% of the explorations lasted less than 7 seconds, 80% less than 17 seconds and 95% lasted less than 31 seconds. This scenario is rather unlikely to happen in practice as in the real world, it can be considered a rare event to have only two operational paths among 18 available.

2.5.3 Paths with different delays

After having studied the impact of the number of broken paths on the exploration time, we now evaluate how well the protocol performs to choose the path with lowest delay. For that experiment, we have compared each queue with an increasing delay in Click. We assigned a delay starting at 0 ms with a 10 ms increment. The last queue (ninth) has a delay of 80 ms. The 9 queues from A to B and from B to A have a symmetric configuration. 500 failures have been simulated. For each failure, we save the path selected after recovery. Figure 2.17 shows a histogram that gives the frequency of selection for each kind of path, sorted by delays. For example, if REAP has selected a path with 0 ms for one direction, and another one with 10 ms for the other direction, we increment each of these classes by one.

The ideal case is plotted as a reference : 50 % of use for each of the two best paths. This is because in the case of the 0 ms path being broken (if it is the current one), an ideal REAP would select the 10 ms path. The next trial would lead to break the 10 ms path and selecting the 0 ms path. Thus, in a perfect world, this experiment would consist of continously jumping between the best path and the second best path.

We can observe that standard REAP gives a uniform distribution, due to the random selection of paths. We have also tried to slightly modify REAP, replac-

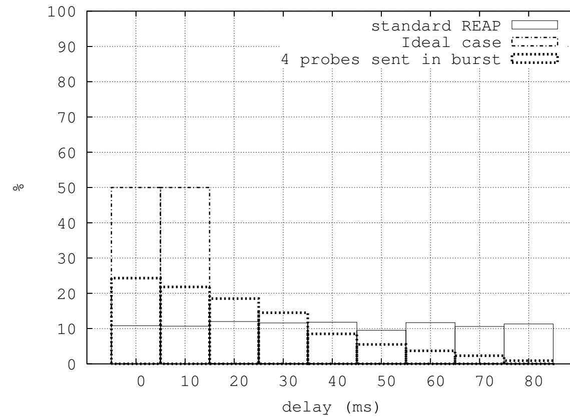

Figure 2.17: Proportions of use for paths with different delays

ing the 500ms initial inter-probe delay with a 0 ms delay. That is, sending four probes in burst. The dashed histogram shows the same measurements made with the modified REAP. While the two best paths have a total selection proportion of 21,5% with legacy REAP, we obtain a proportion of 46,1% when sending 4 probes in burst. This is due to the fact that if probes are sent in burst, the first received answer is taken as the new current path. But that answer is the one corresponding to the path with lowest round trip time among the four tried. If we increase the size of the burst, we will get a higher proportion of selection for the best paths, at the expense of more control packets being sent.

2.6 Improving failure recovery time

In the previous section, we concentrated on the REAP path exploration mechanism. The path exploration is however only a part of the failure recovery mechanism (the first part being failure detection). In this section, we look at how to reduce the overall failure recovery time.

The REAP Failure detection mechanism has been evaluated by simulations in [dBGMS07]. In [dBGMS07], de la Oliva et al. emphasise that the TCP exponential backoff has a negative impact on the recovery time seen by an application. The reason is that after a failure, TCP tries to retransmit until a response is received. The delay between successive retransmissions is exponentially increased. Consequently, when REAP finds a new path, TCP unnecessarily waits for its next retransmission before noticing that the communication path is operational again. [dBGMS07] suggests informing TCP when a new path is found so that it immediately retransmits and recovers. Figure 7 of [dBGMS07] provides simulation results that show the effect of the improvement.

In *LinShim6*, we have added a mechanism that allows for notifications to be issued when any multihoming event occurs. This uses the Linux `netevent` frame-

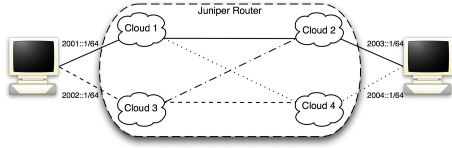

Figure 2.18: Shim6 testbed 2

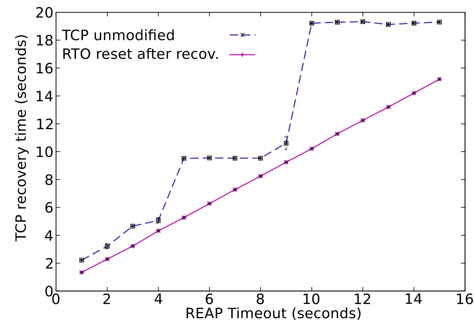

Figure 2.19: Tsend impact on ART

work. Any module in the kernel can register for such notifications, without the knowledge of the shim6 module. This is important for maintaining the layer separation inside the kernel. TCP thus registers for the `PATH_UPDATE` event, and receives a notification when a path has been updated. It reacts by resetting all its RTO (Retransmission TimeOut) timers for the TCP sessions that use that path. *LinShim6* is also modular enough to support external control from informed entities (such as network monitoring daemons), for example to force a change to another path. John Ronan has written such a controlling daemon to take ECN information into account [RM10].

In this section, we use a testbed consisting of a Juniper M10i, as the router, with two Dual Pentium III Blade servers with 512MB of Ram and Gigabit Interfaces (Figure 2.18). Both computers were running the IPerf tool for traffic generation. To simulate link failures, links were switched off in the Juniper router via expect scripts. The gain in recovery time is presented in Figure 2.19. This figure is deliberately very similar to Figure 7 of [dBGMS07]. The goal was to compare the simulation results with the implementation results. We measure the Application Recovery Time (ART), defined in [dBGMS07] as the time elapsed between the last packet reception before a failure, and the first packet received after the recovery. The measurement is repeated for different values of $TSend$ (Failure detection timeout). Each point in the figure is the median of 45 measurements performed in the same conditions. Error bars with percentiles 5 and 95 are also shown, although almost invisible (they appear as small squares) due to the high stability of the results.

With our testbed setup, an Application Recovery Time (ART) cannot be below $Tsend$, because the time of the last packet received is almost equal to the time of the last packet sent (due to the configured packet rate), and the path exploration starts $Tsend$ seconds after the last packet has been sent. In the more general case the lower bound for an ART can be slightly lower as explained in [dBGMS07].

Figure 2.19 confirms the simulation results from [dBGMS07]. We observe an ART that increases linearly with $Tsend$ if the RTO (TCP Retransmission TimeOut) is reset. On the other hand, in the absence of RTO reset, we observe steps in the curve that are due to TCP waiting for its RTO before performing a retransmission.

Regarding the path exploration, our implementation separates the address pairs used for sending probes into two sets, each one randomized. The first set contains all pairs that are completely distinct from the current (stalled) address pair. The second set contains all other address pairs. The first probes sent use address pairs from the first set, in the hope that using an orthogonal path increases the chance our implementation can find a working path on the first attempt. Indeed, in the testbed setup, this proved to be the case. This also confirms what was simulated in [dBGMS07].

While many things are common between our figure and Figure 7 of the aforementioned paper, all our experiments (either with or without RTO reset) reveal a faster ART than the one obtained in [dBGMS07]. One explanation is that while [dBGMS07] sends a probe to the current address pair before actually triggering an exploration, our implementation begins exploration immediately upon expiration of the *Send* timer.

Probing the current locator pair, before commencing the exploration process, is useful when there is some doubt about the failure. For example, a host could receive a spoofed ICMP destination unreachable message, which should trigger a probe on the current pair, but not an exploration. This is so that the host can attempt to detect whether it was a genuine ICMP message or not. In Figure 2.19 and Figure 7 of [dBGMS07], there is a timer expiry that indicates that no traffic has been seen during $Tsend$ seconds. Since REAP ensures (through keepalives) that the Tsend timer expires only when there is truly a network failure, we argue that this is a sufficient condition to immediately start the path exploration, with the benefit of lowering the ART.

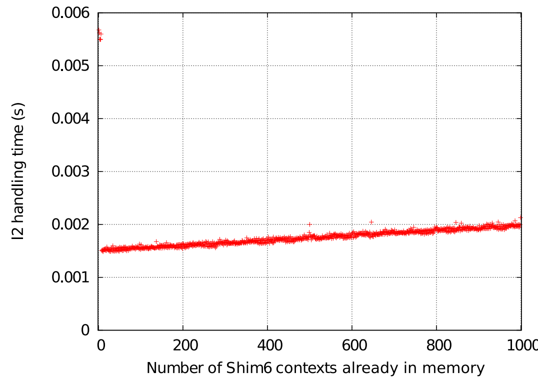

Figure 2.20: I2 generation time under high load

2.7 Cost of state maintenance

Shim6 requires state to be maintained at both the initiator (client) and responder (server). Obviously servers usually manage many connections simultaneously, this would then mean that a server could potentially have to manage many shim6 contexts. In order to reduce the load on a server, it may be preferable to disable the *LinShim6* heuristic. i.e. the context establishment trigger mechanism. That way, the server will never initiate a shim6 negotiation, but only respond to context creation requests from clients. The first step of a shim6 context initialisation would be the sending of an I1 message by the client. The server would reply with an R1 without creating state. Finally the client would send an I2, at which point context state would be created in the server. The I2 message holds the list of locators from the client, secured with a signature that the server is required to verify. If the I2 message is found to be valid, the server would then create a new context, and reply with an R2 message containing it's own signed locator set. As the locators of a server generally do not change all that often, our implementation computes the signatures in advance, in order to spare computing time during the context negotiation.

In our testbed (the same one as used to generate Figure 2.13), we evaluated the I2 processing time, the results of which can be seen in Figure 2.20. Our tests consisted of the following. Every $50ms$, a client initiated a new context negotiation, each client used a different CGA source address, in order to force the creation of a new context in the server. The CGA was generated with a 1024-bit public key. 1000 such contexts have been created, and the I2 processing time measured. The x-axis shows the number of the clients (and hence context creation requests), sorted in chronological order (context 1000 is created $1000 * 50ms = 50s$ after the first one). The figure shows that even when a hosts has 1000 active contexts, the I2 processing time remains at around 2 milliseconds.

Figure 2.21 shows the result of a case study of shim6 context management in our university. The netflow traces of several critical servers in our campus have been analysed (Full IPv4 netflow). Traffic was collected from the 1st to the 7th of

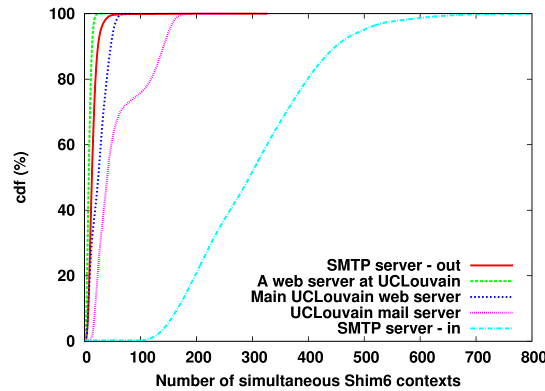

Figure 2.21: Case study of state maintenance on selected servers

August, 2008. In our analysis, we assume that each peer would trigger a shim6 negotiation immediately after the first packet is exchanged and that servers are configured with a garbage collection time of 10 seconds (that is, if no traffic is seen during 10 seconds related to a particular context, then the server decides that it is no longer used and removes it. Peers having more than 10 seconds of idle time then need to renegotiate their context). By comparing with Figure 2.20, we can infer that the I2 processing time (cost of creating a new context) would not exceed $2ms$ for any of those servers. We also observe from Figure 2.21 that even in the worst case where each peer would trigger a shim6 context establishment, the number of concurrent shim6 contexts that need to be maintained is less than 800. Note that in case an administrator wants to reduce the observed number of simultaneous shim6 contexts, he can lower the garbage collection time, in order to more aggressively drop shim6 states. This tuning corresponds to moving state from the server to the network: the more aggressive a server is in dropping contexts, the more often clients will need to refeed the context data through network messages.

Having just explained that servers can avoid unnecessary context creation by simply disabling the shim6 heuristic, and only create contexts upon request from the clients, one simple method that clients could use to reduce their shim6 activity would be to introduce "hints" into the heuristic about whether the peer supports shim6. In particular we know that, currently, the majority of IPv6 addresses correspond to auto-generated MAC-based addresses. Those addresses can easily be detected thanks to their format, i.e. `ff:fe` in the middle of the interface identifier. If the peer uses such an address, most probably it has no support for shim6, because the use of multiple addresses by shim6 requires their format to be either HBA or CGA. Heuristics can be implemented as a kernel module, and a user can define his/her own (or indeed modify the existing one), without having to modify the core implementation. So, for example, a heuristic could be defined to ignore auto-generated addresses, or to limit to some maximum the number of simultaneous shim6 contexts.

	MIPv6	Shim6	MipShim6
Routing Optimisation	V_-	V^+	V^+
Movement Detection	V^+	V_-	V^+
Support of simultaneous movement	V	X	V
Failure detection	X	V	V
Path exploration	X	V	V

V/X: Feature is supported/unsupported
V^+/V_-:V^+ Indicates a more efficient support.

Figure 2.22: Features supported by Mipv6, Shim6 et MipShim6

2.8 Combining Shim6 with Mobile IPv6

We have explained in Section 2.3 that the architecture of *LinShim6* is extensible, thanks to the usage of the modular xfrm framework. In this section we show how *LinShim6* can be combined with Mobile IPv6 to produce a new solution, *MipShim6*, able to handle simultaneously multihoming and mobility. *MipShim6* can be downloaded from `http://inl.info.ucl.ac.be/linshim6`.

Bagnulo et al. already proposed a similar architecture (without testing it, however, as it was not implemented at the time) in [BGMA07]. We implement and evaluate this architecture. We also modify it slightly, removing the Routing Optimisation mode of MIPv6 completely, as we justify how shim6 can achieve the same benefits at a lower cost.

Figure 2.22 summarises the services supported by MIPv6, shim6 and MipShim6, showing the benefit from unifying these technologies. Routing Optimisation (RO) and movement detection are supported by both MIPv6 and shim6, but shim6 (which uses only direct paths) negotiates more efficiently the use of a new path. On the other hand, MIPv6 has received a lot of attention from researchers around movement detection [KMN+07, MN06a, DPN03], and therefore is better at detecting movements compared to shim6, which considers a movement as a failure. The double jumps (where a host and its peer move simultaneously) are only supported by MIPv6 because of the need for a rendezvous point (the home agent), non-existent in shim6. Finally, shim6 is able to monitor a path end-to-end, and try to find another pair of addresses in case of failure of the current path. This allows supporting failures of the home agent, by simply switching to another one.

The architecture we propose is illustrated in Figure 2.23. To ensure the stability of the transport and application layer identifiers, a single address is presented to them. As recommended also by Bagnulo et al. [BGMA07], the address used at this level is a home address, because of its longer lifetime compared to CoAs. We call

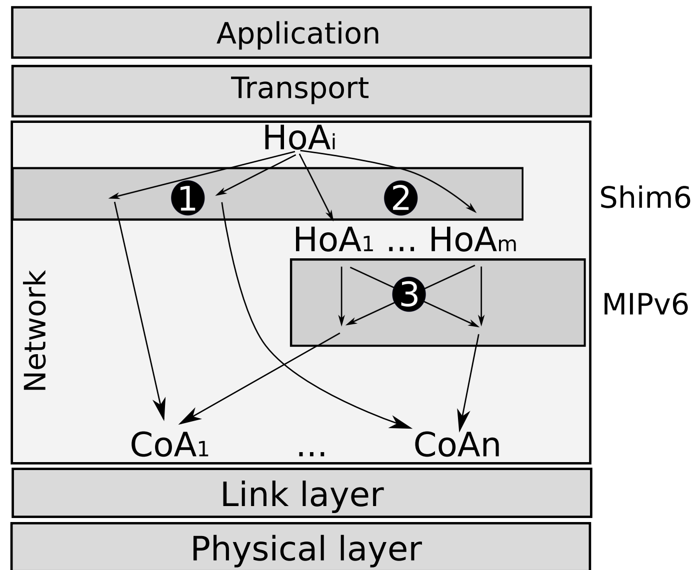

Figure 2.23: Architecture of the MipShim6 stack

this a ULID (Upper Layer IDentifier) to use the shim6 terminology. If a mobile node has multiple home addresses, anyone of them can serve as a ULID.

Any packet leaving the transport layer passes first through the shim6 sublayer, which provides an end-to-end address translation service. The shim6 service is illustrated in Figure 2.23, and has two components, numbered ***1*** and ***2*** in the figure. In the first component, the ULID is rewritten into one of the available Care-of addresses discovered in the visited network. We call this type of translation **Shim6-RO**, because the immediate use of a care-of address without using MIPv6 allows to reach the correspondent node through a direct path, as would the MIPv6 RO were it used.

In the second component, the ULID is rewritten into one of the home addresses. This is useful only when the home address used as ULID has failed and the mobile node has moved simultaneously with the correspondent node (CN), hence needing to pass through the rendezvous point (the home agent) to get back in contact with the CN (see Figure 2.24). In the absence of double jump, only care-of addresses are used and home agent failures do not need to be detected. When doing home address rewriting, two successive transformations take place. First, shim6 replaces the home address used as ULID by the current locator (if necessary). This is the case of the MN in Figure 2.24, which replaces HoA_1 by HoA_2, the home agent HA_1 being down. Then the packet passes through the MIPv6 layer, which encapsulates the packet in order to send it to the home agent. Shim6 address rewriting from one HoA into another one allows supporting both the multihoming of the home network and the failure of a home agent.

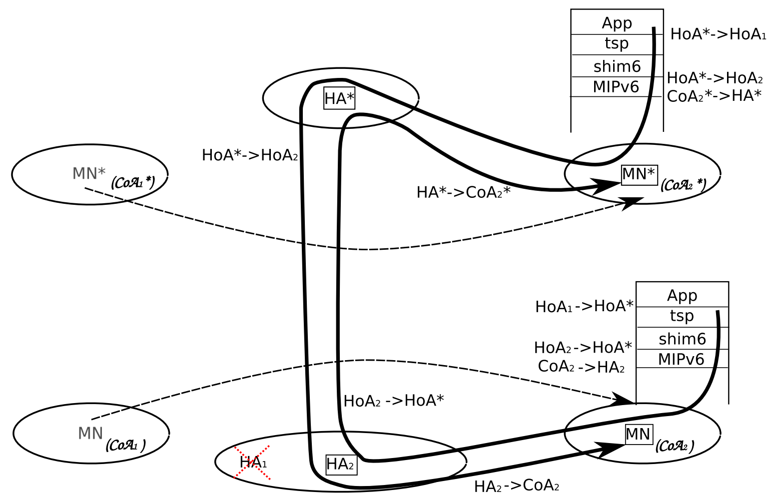

Figure 2.24: Double jump scenario with MipShim6

The third component is the MIPv6 layer. It is mainly responsible for managing double jumps, but also the movement detection and the initialization of a communication context when a shim6 context has not yet been established (when away from home, the only option to start a new exchange is to use the home agent). MIPv6 provides encapsulation for sending packets to the corresponding node through the home agent. As stated in the specification [JPA04], it is possible to maintain multiple Care-of Addresses, although only one of them may be registered at the Home Agent. This allows MIPv6 to try another Care-of address, should the current binding attempt fail.

Shim6 can trigger a locator change when a failure is detected by REAP, or following a movement notification from MIPv6. In that case a REAP exploration is initiated. This involves sending probes on each of the paths, CoAs being preferred as they allow using direct paths to the Correspondent Node (CN). Figure 2.25 shows an example where *MipShim6* switches to Shim6-RO mode after a movement: shim6 waits until the tunnel is established before sending an *Update Request (UR)* to the CN. The remote node however still needs to do a REAP reachability test, before actually using the new path. This is to prevent a malicious node from redirecting the traffic from the correspondent node.

Support for multiple home addresses: Multiple Home Addresses (HoAs) can be given to a mobile node for two main reasons. The first one is that the home network can be multihomed and receive multiple IPv6 prefixes. In this case each prefix is used to generate a new HoA. The second reason is the use of redundant Home Agents. One prefix is assigned to each home agent so that the host can choose (using the shim6 layer) through which of them to send its traffic. This is useful to allow tolerance to Home Agent failures and load distribution across Home Agents.

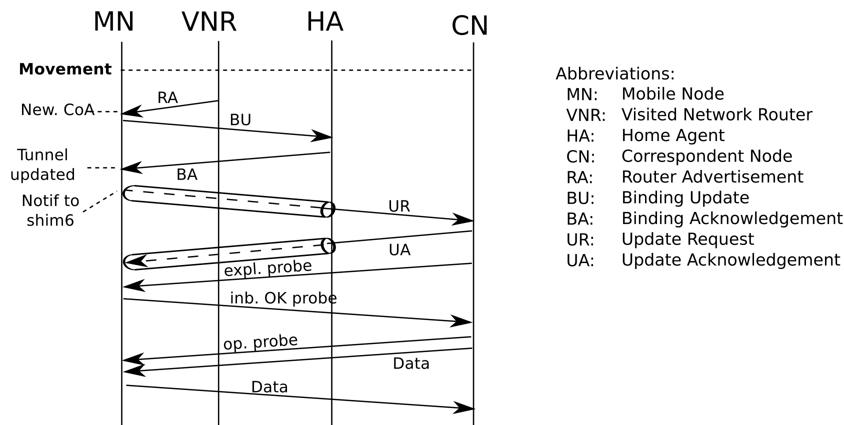

Figure 2.25: Message sequence during a move

Support for multiple Care-of Addresses: The use of multiple CoAs allows supporting multihoming of the visited network. [WDT+09] proposes an extension to support multiple CoAs, but it is not needed thanks to the integrated support of shim6.

2.8.1 Movement with MipShim6

Movement is characterized by the loss of the current CoA and the acquisition of a new CoA. MIPv6 uses generic IPv6 techniques to detect movement and then makes a Binding Update/Binding Ack exchange with the Home Agent when acquiring a new CoA. Our architecture can reuse the movement detection optimisations already developed for MIPv6 [MN02, DPN03].

Depending on whether or not the mobile node is in Shim6-RO mode at the time of movement, we distiguish two cases. If it is not in Shim6-RO mode, the tunnel is simply updated and Shim6 can receive the movement notification, enabling it to trigger an immediate switch to Shim6-RO (as shown in Figure 2.25). In the second case, the mobile node has moved while using the Shim6-RO mode. MIPv6 also updates its tunnel, although it is not used. As in the previous case, it sends a notification to shim6, which may change the current locator.

It is possible that Shim6 detects a movement before MIPv6 or at the same time. This happens when the failure detection timer is set to a value lower than the time required for MIPv6 to accomplish its movement detection and failover. In this case, the movement is actually perceived as a failure by Shim6 which immediately begins to send probes. These will be unsuccessful until MIPv6 has finished updating its tunnel with the Home Agent. Although this case leads to the same result as the second scenario, it is better to avoid it because it generates more control traffic. It is possible to avoid this competition between shim6 and MIPv6 by setting the shim6 failure detection timer to a higher value compared to the average MIPv6 movement recovery time.

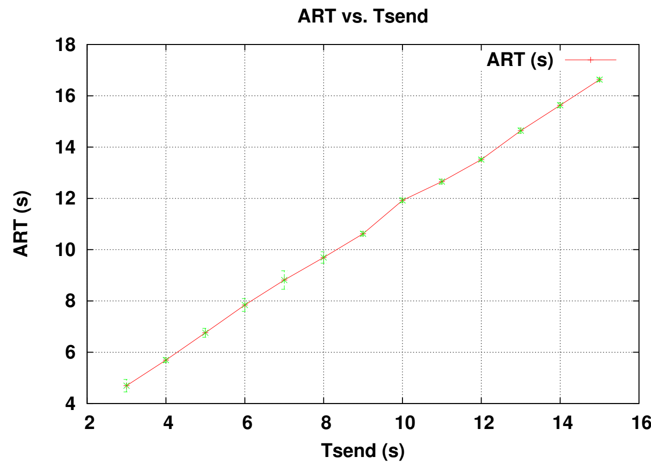

Figure 2.26: Recovery time after a failure of the Home Agent vs TSend timer

2.8.2 Validation

We validated *MipShim6* by running it in two scenarios. In the first scenario, a Mobile Node communicates with a peer in tunnel mode and experiences a failure. Figure 2.26 shows the application recovery time as a function of the shim6 failure detection timer, when a mobile node experiences a failure of the Home Agent. The result is quite similar to what we obtained for pure shim6 in Figure 2.19, where the transport layer is informed about the path change and resets its retransmission timer.

In the second scenario, we verify the correct behaviour of *MipShim6* in case of double jump. This result is shown in Figure 2.27. After an interruption of a few seconds, the TCP exchange recovers. Note that this uses the MIPv6 part of *MipShim6*, given that the Home Agent is needed in this case to act as a rendezvous point. The goodput is highly variable because our lab uses 802.11g wireless connectivity.

More discussion on MipShim6, its implementation and the validation can be found in our *MipShim6* paper [BDMB09].

2.9 Open Issues with shim6 multihoming

From a standardisation viewpoint, the shim6 IETF working group has concluded and several RFC's have been published [NB09, AvB09, Bag09]. Our implementation supports all the important features of shim6. However, there are still several outstanding issues to be solved before there will be a widespread deployment of shim6.

A primary issue is that shim6 requires IPv6. As of this writing, the Internet still mainly uses IPv4, but given the expected exhaustion of the IPv4 address space,

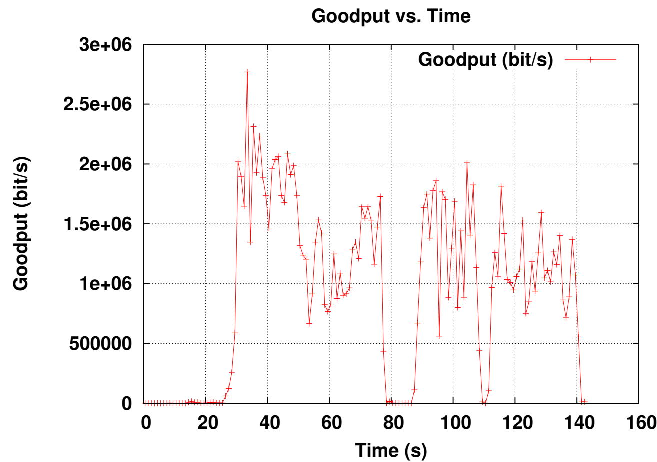

Figure 2.27: TCP goodput before/after a double jump

more and more networks are seriously considering IPv6 and have started deployments [Hus10]. Shim6 could be a very useful feature for multihomed networks. Initially, shim6 could be used for important flows such as VPN, e-commerce or IP telephony servers where rapid recovery from link or interface failures is important. An important advantage of shim6 over other multihoming solutions such as HIP or SCTP, is that shim6 does not require any change to the applications running on hosts. Thus, applications can benefit of shim6 without being aware of it. Simulation studies performed during the early phases of the shim6 development have shown that host-based multihoming techniques such as shim6 allow hosts to use many more paths to send their packets than traditional IPv4 BGP-based multihoming [de 05]. Furthermore, measurements have shown that by using these additional paths, it is possible to achieve much better performance, e.g. lower delays [de 05].

However, there are also some forces against a widespread shim6 deployment. At present, Internet Service Providers are very reluctant to consider it [Sch05]. Their main concern with shim6 is that it allows hosts to influence the path used to send and receive packets towards any multihomed destination. ISP operators have become accustomed to performing traffic engineering by refining their BGP configurations to take into account business policies. Consequently many consider that the deployment of shim6 would limit their traffic engineering capabilities and make the network more difficult to manage [Sch05]. This is not necessarily true. Shim6 provides benefits to both ISPs and their clients. ISPs can benefit from a much more scalable interdomain routing system while clients can benefit from a much larger number of paths providing better performance and more redundancy.

In fact, peer-to-peer applications are also exploiting these alternative paths. Network operators could market shim6 as an added value service to their customers willing to obtain improved performance or reliability. This service could be combined with a path selection service provided by the ISP that allows its clients to

easily determine the best path to reach a given destination. This type of service is already being developed to support peer-to-peer applications [AFS08]. Recently, the IETF has chartered the ALTO working group to work on such a service [SB09] which would be very useful for shim6 to integrate with.

A second issue concerning a widespread deployment of shim6 is that many corporate networks insist on using Provider Independent IP addresses, even for IPv6, instead of Provider Aggregatable addresses [Pal09]. This is because most operators consider that renumbering a network is too complex. Despite a lot of discussions on this topic [CAF10], the IETF does not provide a solution to easily renumber a corporate network. Thanks to DHCP and IPv6's stateless auto-configuration, most hosts can easily change their address, but for servers and routers this remains difficult. For the specific case of shim6, a complete renumbering solution is not necessary. To easily support provider changes, a corporate network could use private addresses internally (e.g. for the routers and the management servers) and simply add the prefixes allocated by their providers to all their routers. Solutions to address this issue have been proposed in [LB09].

2.10 Related Work

Other solutions have been developed to solve the multihoming problem. The SCTP transport protocol [Ste07] was initially designed to support signalling servers in IP telephony environments. It has now been extended to support wider deployment scenarios and is supported by several operating systems. Another example is the Host Identity Protocol (HIP) [MN06b]. HIP has been developed to evaluate the benefits and drawbacks of using a new cryptographical identifier namespace on top of IP. HIP has been extended to support multihoming and mobility [NHVA08] and there are several implementations of HIP available [KVG07]. Compared to these solutions, the main benefit of shim6 is that it does not require any change to the applications. This is very important for a new technique that needs to be incrementally deployed.

Several years ago, based on the recommendation from [MZF07], the Routing Research Group of the Internet Research Task Force (IRTF) was rechartered to consider the evolution of the Internet architecture. Several of the techniques being evaluated within this working group [Atk11, FFML11, Vog08] rely on separating the identifier and locator roles of the IP addresses, as in shim6. Although the details between these protocols and shim6 vary, the experience gained by the implementation and improvements of shim6 will be beneficial for the development of these new protocols.

Two other prototype implementations have been developed respectively by Park et al. [PCC+07] and Ahrenholz and Henderson [AH08], also on the Linux platform. They are mainly user-space implementations with `netfilter` hooks to capture the shim6 packets and process them in user space daemons. In contrast, our implementation uses the xfrm framework and is implemented partially in the

kernel with the non time-sensitive functions in user space. Another important difference is that our implementation completely supports the security mechanisms designed for shim6.

Finally, recently an IETF working group has been created (MPTCP) to design a modified version of TCP, called Multipath TCP [FRHB11], that is able to failover from one path to another, and to spread one single transport flow across several paths. MPTCP will be our main topic starting at Chapter 3, and we propose in Section 4.4.2 an architecture that allows integrating shim6 with MPTCP.

2.11 Conclusions

Multihoming is one of the problems that limits the scalability of the current Internet architecture, because it is currently obtained through injection of additional routes in the BGP system. Shim6 brings a solution to that, by making possible a hierarchical allocation of IPv6 addresses, through increased path management by the end-hosts.

In this chapter, we have studied in details the HBA and CGA mechanisms. Although HBA is computationally cheaper compared to CGA, CGA does offer interesting capabilities in terms of flexibility. For example, an interesting possible extension of our work could be to augment shim6 with IPv4 support, by simply including IPv4 addresses in the CGA signature. The only requirement for this to work would be to use an IPv6 address as ULID, so that the host public key can be encoded in this ULID. This would effectively allow a host to use both IPv4 and IPv6 protocols in the same communication, without the knowledge of the application.

We also analysed in depth the failure recovery capability of shim6, and explained that the associated REAP protocol allows moving the traffic from one path to another. But one could wonder whether it is possible to use several shim6 paths simultaneously. Although this cannot be done autonomously in the shim6 layer, because TCP performance would drop due to the incurred reordering, the concept of *context forking* [NB09] has been proposed in the shim6 specification to allow an upper layer (e.g. a shim6-aware application) to control the paths on its own. This is costly, however, as it requires creating a full shim6 context per path that needs to be handled simultaneously (more memory required). Moreover, each of these contexts must be negotiated separately with the peer (more time required for context establishment). This can be useful in specific cases where an application needs full control of the shim6 context and hence needs to “own” one. However, for simultaneous use of shim6 paths, we will instead propose in Section 4.4.2 a completely *local* interface (that is, without change in the shim6 protocol or additional network exchanges) that enhances shim6 with simultaneous multipath capability, given an appropriately interfaced upper layer can control it (we do this with Multipath TCP).

Finally, we described our MipShim6 proposal, that combines Mobile IPv6 with Shim6. An interesting future work would be to put together MipShim6 with the

Multipath TCP protocol, presented in the next chapter. This would indeed result in a full solution for handling multihoming, mobility and simultaneous use of multiple paths.

Chapter 3

Understanding Multipath TCP

In the previous chapters, we have studied multihoming as seen from the network layer. We will now concentrate on a recently developed protocol, MPTCP, that can handle multiple paths at the transport layer. Before presenting our contributions, we provide a short introduction to regular TCP and to MPTCP as it is defined currently by the IETF [FRHB11].

3.1 Regular TCP

The Transmission Control Protocol (TCP), standardized in 1981 [Pos81b], is used in the vast majority of Internet communications. Together with the User Datagram Protocol (UDP [Pos80]), this transport layer protocol carries 95% of the network traffic on the Internet [LIJM$^+$10].

TCP provides a reliable bytestream abstraction to applications. **Reliable** means that any lost or corrupted data is retransmitted until it is received. A TCP sender knows that a range of bytes have correctly reached the destination thanks to *acknowledgements*. The data bytes are given *sequence numbers* that are referenced in the acknowledgements to indicate the amount of data that has been correctly received. For example, an acknowledgement of 1000 means that the bytes with sequence numbers 0 through 999 have been successfully received (assuming an initial sequence number of 0 in this case). TCP estimates the *Round Trip Time* (RTT) as the time elapsed between the transmission of a segment and the reception of the corresponding acknowledgement. This RTT is used to configure a timer that triggers an automatic retransmission of data when it expires. All of this is done without the knowledge of the application, that only needs to feed a socket with bytes and can be confident that the data will reach the peer, as long as the network path used for the communication is not completely broken.

Bytestream abstraction means that TCP allows the application layer to exchange a flow of bytes with the peer, which requires handling segmentation on behalf of the application because the underlying network layer can only handle packets (see Figure 1.2 in Chapter 1 for a reminder on Internet layering). TCP

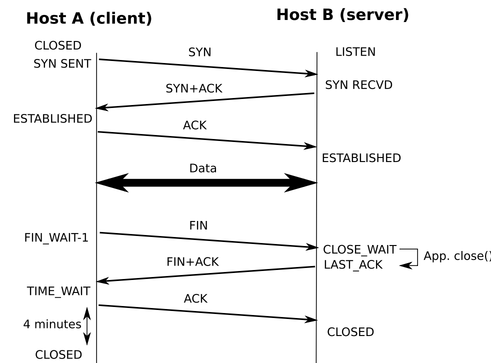

Figure 3.1: Example TCP exchange

can negotiate a *Maximum Segment Size* at the beginning of a connection, and later on tries to fill all segments up to their maximum size (in order to benefit from the best possible control/data ratio in each segment), thanks to the Nagle algorithm [Nag84].

[Pos81b] has been very careful with the establishment and termination phases of a TCP connection, and a full state machine is defined in [Pos81b, Figure 6]. We show a simple example of TCP communication in Figure 3.1 (the state names on the left and right of the picture refer to the state machine in [Pos81b, Figure 6]).

A TCP server waits for incoming connection requests. Until a request is received, it is in `LISTEN` state (The `CLOSED` state refers to the absence of a socket). A connection request segment, in TCP terminology, is called a `SYN`. The server creates no connection specific state until the `SYN` is received (If `SYN` cookies are enabled [Ber], state creation is even delayed until the third message (`ACK`) has been received). For this reason, no data can be sent by the client until the `SYN` has been acknowledged. If after some time the acknowledgement is not received (meaning that either the `SYN` or the `ACK` has been lost in transit through the network), the `SYN` segment is retransmitted. The server neither enters the `ESTABLISHED` state until its own `SYN` has been acknowledged. This is important because the `SYN` segment contains the Initial Sequence Number (ISN), randomized for security reasons. Entering the `ESTABLISHED` state before the reception of the `ACK` would result in potentially sending data that the receiver will not be able to localize in the sequence number space. This initial exchange of three segments is traditionnally called the **three-way handshake**.

A similar handshake happens when terminating the connection. A `FIN` is issued when the application performs a `close()` system call on the TCP socket. The reception of the `FIN` generally triggers a `close()` in the application on the server as well, which allows sending the server `FIN` and the acknowledgement of the client `FIN` in the same segment. When the server receives the `ACK` from the client, it can safely remove any state related to that connection. However, the client has no way to know if his `ACK` has actually reached the peer. If not, it must be

prepared to receive a retransmitted `FIN+ACK`. For this reason, [Pos81b] defines an arbitrary timeout of four minutes, during which the client socket must remain open in order to resend an `ACK` in response to any retransmitted `FIN+ACK`.

[Pos81b] defines some important state variables. We explain some of them here, as we will discuss them later as well in the case of MPTCP.

- $RCV.NXT$: Next expected sequence number, that is, all the bytes until $RCV.NXT - 1$ have been correctly received.

- $RCV.WND$: Receive window. TCP ensures that bytes are transmitted in the correct order, thanks to the sequence number. In case segments arrive out of sequence, they are reordered in a buffer on the receiver. In particular, if a segment is lost, a TCP receiver must be able to store all the data that follows the lost segment until it is retransmitted. This can quickly fill up the receive buffers. Another case where buffering is needed at the receiver is when an application is very slow at reading the incoming data. In those cases where buffering is needed, it is important to prevent the sender from providing data that cannot be stored. The receive window is an indication, included in each segment, of the current available receive buffer. $RCV.WND$ indicates a window in the sequence number space that the sender is allowed to use for sending new data. The exact window is defined as $[RCV.NXT, RCV.NXT + RCV.WND - 1]$.

- $SND.NXT$: Next byte to send. The sequence numbers increase monotonically after each new sent byte. If, for instance, a 1500 bytes segment is to be sent, it is given sequence number $SND.NXT$ and then $SND.NXT$ is incremented by 1500.

- $SND.UNA$: First unacknowledged byte. It stores the highest cumulative acknowledgement received so far. All data with sequence numbers lower than $SND.UNA$ can be discarded from the send buffers, as it has been correctly received by the peer.

- $SND.WND$: This is the sender view of the $RCV.WND$ state variable defined above. From the sender point of view, the allowed window for sending new segments is $[SND.UNA, SND.UNA + SND.WND - 1]$.

Congestion control: We have mentioned above the case of a TCP flow being limited by the receiving application. In that case the receiver advertises a small receive window, to force the sender not to send faster than it can read. It is also possible (and this case is more frequent), that a TCP flow is limited by the network. A flow is network-limited when it uses all the available capacity on the bottleneck link. In such a case, the bottleneck router starts dropping segments, which requires retransmissions, as explained above. If the sending rate is not controlled, however, there is an amplifying congestion effect: The losses induce retransmissions, *in*

addition to the new data being sent in parallel at the same rate. Hence the overall transmission rate is effectively increased, while a drop normally means that the sender is already transmitting too fast.

To control congestion, the solution is to maintain a separate window called the *congestion window* (cwnd) [Ste97]. A sender can send a new segment only if it fits in *both* its send window and congestion window. The intuition is that it can send only if both the network and the receiver are able to handle the new data. The congestion control always happens in two steps. The first step, **slow start**, is used when the congestion level is unknown and a first probing is needed. A more accurate term for it however would be "fast start", as the sending rate is increased exponentially until the first loss event. At that moment, another variable, the *slow start threshold*, is set to one half of the congestion window ($ssthresh = cwnd/2$). $ssthresh$ determines when the sender must be more conservative in increasing its congestion window. This more conservative, second step is called **congestion avoidance**. In the first implementation of congestion avoidance (BSD4.3, Tahoe), the increase was linear, again until a loss happened. There are other congestion avoidance mechanisms in modern TCP implementations such as TCP Vegas [BOP94], CUBIC [HRX08] or Illinois [LBS08], which will not be discussed here.

3.2 Multipath protocols

MPTCP is the most recent but not the first effort to handle simultaneous use of multiple paths at the transport layer. Previous multipath efforts can be classified based on the directions taken by the authors. One obvious direction is the chosen transport protocol: SCTP or TCP. We remark that in many aspects the conclusions for one or the other of those protocols are very similar, because the main problems encountered when designing a multipath protocol are not related to the specificities of SCTP or TCP. Both protocols need to solve reordering problems when data is spread across multiple paths, and adjust their buffer allocation. SCTP-based [IAS06, ASL04, LWZ08] multipath approaches justify their protocol choice by SCTP's built-in ability to define multiple streams [Ste07], which can be more easily turned into concurrent subflows. But the TCP adopters [MK01, HS02, ROA05, ZLK04] have shown that it is equally feasible to turn TCP into a multipath protocol. The reason is that while SCTP provides a socket interface for controlling the subflows, such an interface is not needed when only one flow is presented to the application and the spreading of data across subflows is done internally. In the end, TCP was chosen to implement multipath capabilities because, in contrast to SCTP, it is widely deployed in today's Internet.

Another important direction is the choice of the sequence space. Some proposals use a single sequence number space [MK01, ROA05, IAS06]. This choice may lead to significant reordering of sequence numbers at the receiver. Since reordering is normally considered as a failure indication, new loss detection heuristics are needed to distinguish between normal multipath reordering and failures. To get rid

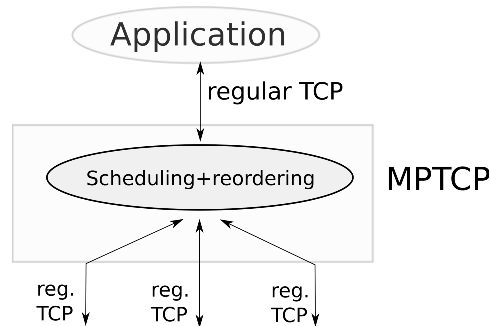

Figure 3.2: MPTCP is transparent to both the network and the applications

of that problem, MPTCP defines a dual sequence number space, where one space is subflow specific and identifies bytes within a subflow. [FRHB11] defines a Data Sequence Number (DSN) space, which takes care of reordering at the aggregate connection level, but these DSNs do not have any impact on retransmission decisions. The closest related work from that point of view is [HS02].

A third design choice is the way to deal with shared bottlenecks. There is a fairness problem when several multipath flows share a bottleneck when employing regular TCP congestion control per subflow, because they would get a higher proportion of the available capacity compared to standard TCP flows. Zhang et al. solve that problem by trying to avoid establishing several subflows across the same bottleneck, thanks to an external tool [ZLK04]. Other approaches simply ignore the problem. In MPTCP, the congestion control algorithm is coupled across all subflows, so as to ensure fairness without needing to detect shared bottlenecks [RHW11, WRGH11]. A good, detailed overview of multipath transport approaches can be found in [Ong09].

3.3 Starting a new MPTCP session

One of the main design goals behind MPTCP was to be completely transparent to both the application and the network. This is illustrated in Figure 3.2. The application opens a regular TCP socket, which initially starts one regular TCP subflow. Up to that point, there is no major difference between TCP and MPTCP. But if both endpoints support Multipath TCP, additional subflows can be initiated by either host. Outgoing data is then scheduled according to some implementation dependent policy. Incoming data from all TCP subflows is reordered to maintain the in-order bytestream abstraction of TCP, as seen by the application. Scheduling and reordering operations are represented by the ellipse in the center of Figure 3.2.

Subflow establishment is shown in Figure 3.3. It is slightly simplified for the sake of describing the main idea, and the full version will be detailed in Section 3.6, in Figure 3.7. `Host A` wants to contact `Host B`. From the DNS, it learns that `Host B` can be reached through address `B.1`. Because MPTCP must

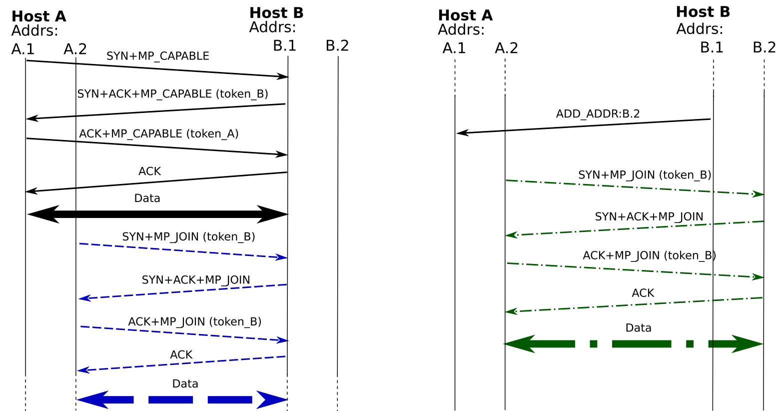

Figure 3.3: MPTCP initial exchange and subflow establishment

be transparent to the network, any new TCP subflow (including the first one) must be established using a three-way handshake[1]. It is augmented with MPTCP-specific TCP options, that need to be understood by the end-system stacks only (not by the network). Thanks to the MP_CAPABLE option inserted in the SYN and SYN+ACK, both Host A and Host B can know that the other end supports MPTCP. MP_CAPABLE is also present in the third message of the three-way handshake to allow the server to defer the state creation until the end of the handshake, and make use of SYN cookies [Ber]. If nothing more is done, MPTCP runs exactly like TCP.

At any moment either host can try to establish a new subflow. Again, this is done through a regular TCP three-way handshake. The option used for additional subflows must be different, however. The peer must be able to understand that the new subflow must be attached to an existing MPTCP connection, and this is made possible by the **token**. During the first three-way handshake both Host A and Host B choose a token to identify the new connection locally. The token is announced to the peer in the MP_CAPABLE option. Since Host A has two addresses, it can establish a new subflow using addresses $< A.2, B.1 >$, and join it to the correct context in Host B by attaching token_B to the MP_JOIN option. Note that token_A does not need to be included in the SYN+ACK because Host A has state for that subflow already, which is not the case of host B. Similarly, Host B could establish the subflow $< A.1, B.2 >$. But neither Host A nor Host B can establish the subflow $< A.2, B.2 >$ since none of them knows the second address of the remote host. This address pair precisely corresponds to the most distinct path (in most network configurations). Thus a new MPTCP option is needed: ADD_ADDRESS. Its use is shown in the right part of Figure 3.3. After Host B has announced that it can be reached at address B.2 as well, Host A

[1] The final ACK makes it actually a four-way handshake, but it is seen by the network as a regular TCP ACK, independent from the establishment. The reason for using a four-way handshake is explained in Section 3.6

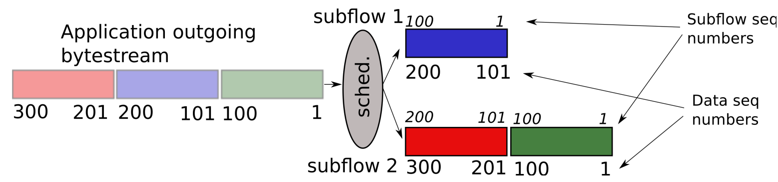

Figure 3.4: MPTCP Data Sequence Numbers (DSNs)

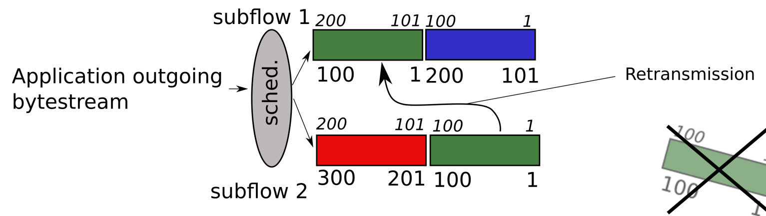

Figure 3.5: MPTCP retransmission

has the opportunity to establish one more subflow.

To close this section, we note that MPTCP subflows can be established over IPv4 or IPv6. A single connection can make use of both underlying networks. This is a very interesting property of MPTCP, as it facilitates the transition from IPv4 to IPv6 and allows hosts having both kinds of addresses to use two different underlying network protocols simultaneously.

3.4 Exchanging data across multiple flows

At the sender, an implementation specific scheduler (represented as a ellipses in Figures 3.4 and 3.5) decides over which subfow to send any byte of data (we describe our design of such a scheduler in the next chapter). As a result, the initial ordering of the application bytestream is lost, and holes appear in the sequence numbers of the bytestream (e.g. subflow 2 in Figure 3.4). This is not desirable as some network devices (e.g. TCP normalizers [HPK01]) analyse the TCP sequence numbers and block a TCP flow when holes appear in the sequence numbering. This motivated the definition of a **second sequence number space**. One sequence number space is related to subflow sequence numbers. Each subflow maintains its own subflow sequence number space, and writes sequence numbers in the same field as regular TCP. Again, the MPTCP information is stored in options, and a DSS (Data Sequence Signal) option has been defined by [FRHB11] to carry the data sequence numbers. Subflow sequence numbers and data sequence numbers are illustrated in Figure 3.4.

Retransmissions: When a loss is detected (loss detection is performed on each subflow individually), it is not necessarily the best choice to retransmit on the same

subflow. For instance, if the loss happened on a high delay path, it may be wise to retransmit the segment on another subflow. This can be done naturally with MPTCP, by simply remapping the data sequence numbers to new subflow sequence numbers, as shown in Figure 3.5. However, the original segment must still be retransmitted on its initial subflow to avoid confusing middleboxes that check for payload consistency upon retransmissions.

Acknowledgements: Due to the subflows behaving like regular TCP connections on the wire, the subflow sequence numbers are acknowledged normally on each subflow. In theory this should be sufficient, because the sender can infer the acknowledged data sequence numbers from the received subflow acknowledgement. However, two practical problems exist with that solution:

- Various kinds of middleboxes have been developed to improve the behaviour of TCP, and are grouped under the term *Performance Enhancing Proxies* (PEPs) [BKG+01]. Some of them acknowledge TCP data before it is actually acknowledged by the peer. If the data happens to be lost after it has been acknowledged by the PEP, it is the responsability of the device to retransmit it. However if the path has failed (that is, any retransmission from the PEP fails), or if the receiver is mobile, so that the PEP retransmissions cannot reach it anymore, the other subflows cannot be used to retransmit the lost data, as the sender has released the corresponding memory.
- In regular TCP, the receive window is defined as follows [Pos81b]:

  $$[RCV.NXT, RCV.NXT + RCV.WND]$$

  $RCV.NXT$ is determined by the `ACK` field in the TCP segment. However, in MPTCP there is no subflow-specific receive window (for reasons that we describe in Section 4.3.5). The advertised receive window is related to the data sequence number space, and is redefined as follows:

  $$[DATA.RCV.NXT, DATA.RCV.NXT + DATA.RCV.WND]$$

  That is, the receive window is no longer a fraction of the subflow sequence number space, but instead a fraction of the data sequence number space. If the data acknowledgement is inferred from the subflow acknowledgement, it is not cumulative, and hence does not reflect $DATA.RCV.NXT$ (which is the next data sequence number expected by the receiver).

To solve the above problems, the protocol specification has defined a *data acknowledgement* option, that is included in the same option as the data sequence number. It solves the first problem because the cumulative Data Acknowledgement can be sent on any subflow, and truly reflects the state of the receiver, even in the presence of PEPs. It also solves the second problem because it explicitly defines the left edge of the data level receive window. Finally, it even simplifies

implementations, by removing the need to infer the data acknowledgement from the subflow-level acknowledgement[2].

Congestion Control: Congestion control algorithms for regular TCP try to use their fair share of the available capacity. Two TCP subflows belonging to a logical connection would then use twice their fair share. Moreover, [WRGH11] describes a scenario (with multiple bottlenecks) where even an aggressive MPTCP would fail to obtain the best possible bandwidth. The authors explain that it is desirable, in multipath scenarios, to use only the less congested paths instead of spreading the traffic equally among the available paths. Starting from an existing theoretical solution [KV05, HSH+06], [WRGH11] develops an algorithm, and adapts it to fulfill the following two goals: *"A multipath flow should give a connection at least as much throughput as it would get with single-path TCP on the best of its paths. This ensures there is an incentive for deploying multipath."* and *"a multipath flow should take no more capacity on any path or collection of paths than if it was a single-path TCP flow using the best of those paths. This guarantees that it will not unduly harm other flows at a bottleneck link, no matter what combination of paths passes through that link."*

The resulting congestion control algorithm is as follows:

- For each non-duplicate ack on subflow i, increase the congestion window of the subflow i by $min(\alpha * bytes_acked * MSS_i/cwnd_{tot}, bytes_acked * MSS_i/cwnd_i)$ (where $cwnd_{tot}$ is the total congestion window of all the subflows and $\alpha = cwnd_{tot} \frac{\max_i(\frac{cwnd_i}{RTT_i^2})}{(\sum_i \frac{cwnd_i}{RTT_i})^2}$). This formula assumes that the congestion window is measured in bytes and $bytes_acked$ is the number of bytes acknowledged by the received ack segment.
- Upon detection of a loss on subflow i, decrease the subflow congestion window by $cwnd_i/2$.

This congestion control algorithm has been proposed by the same authors in an IETF draft [RHW11]. The MPTCP specification [FRHB11] does not mandate the use of that algorithm, however, and emphasises that congestion control is separate from the main specification, to leave space for future definition of other congestion controllers. As of this writing, this is the only congestion control scheme that has been adapted to MPTCP.

3.5 Terminating an MPTCP connection

In the previous sections, we described the motivations that drove the definition of data sequence numbers and data acknowledgements. In this section, we explain

[2] Our implementation initially worked without data acknowledgements, but this forced us to maintain a list of acknowledged fragments in the data sequence number space, just like the SACK implementation in TCP

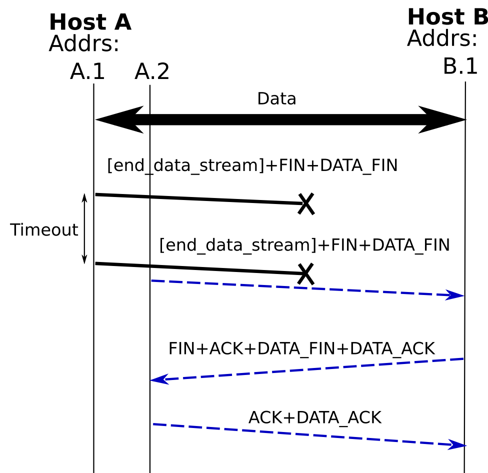

Figure 3.6: MPTCP example connection termination

why MPTCP also includes a **Data FIN** [FRHB11].

One could assume that, to close an MPTCP connection, it is sufficient to send a regular `FIN` on all subflows. This works indeed if all subflows are operational, but will fail if any of the subflows is broken. In regular TCP, if the flow is broken, the connection cannot be closed gracefully anyway, but MPTCP would not be affected by such a failure, as every useful information about the connection can be moved to another working subflow.

The semantics of the `Data FIN` is the same as the semantics of the `FIN`, at the data level. A regular `FIN` does not close a connection, it closes a subflow. A `Data FIN` can be sent on *any* subflow. It is acknowledged with a `Data ACK`, and occupies one byte in the data sequence number space.

An example connection termination is shown in Figure 3.6. We deliberately show a particular case, where subflow $< A.1, B.1 >$ fails, to illustrate the behaviour of MPCTP in that case. `[end_data_stream]` represents the last block of application data. Since the application has issued a `close()` system call after this last block, MPTCP appends a `Data FIN` to the transmission queue. Subflow $< A.1, B.1 >$ is selected by the scheduler. Since the `Data FIN` is present, MPTCP knows that it can as well close the subflow, and also sets the `FIN` flag. Unfortunately this termination segment is lost, and after a timeout it is retransmitted. Thanks to MPTCP's ability to retransmit segments on other subflows (see Section 3.4), the lost segment is also retransmitted on subflow $< A.2, B.1 >$. The retransmission successfully reaches the peer, and the final graceful shutdown can happen on this second subflow. From the application viewpoint, the stack can return that the connection has been correctly closed in the outgoing direction. From the viewpoint of MPTCP, the failed subflow will continue to retransmit a `FIN` segment, with exponential backoff, and finally terminate after a timeout. This is done without the knowledge of the application. Note that [FRHB11] proposes to reduce

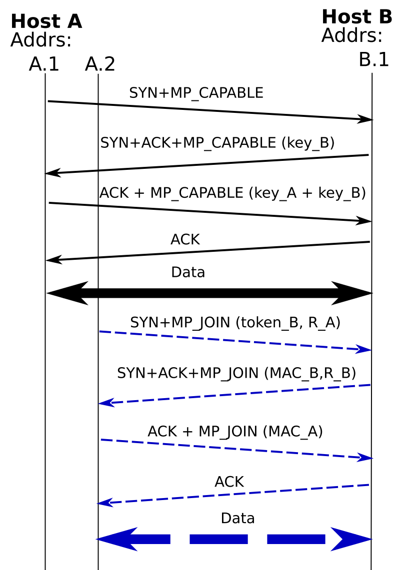

Figure 3.7: MPTCP authentication

the timeout value when the subflow is not needed anymore for the connection.

3.6 Security mechanisms for MPTCP

The security mechanisms for MPTCP have been subject to a long debate in the IETF, and consensus has been reached only recently. For that reason they are not yet included in our Linux MPTCP implementation. In this section we give an overview of the problems and solutions as defined in [FRHB11].

A threat analysis for MPTCP is detailed in [Bag11]. It states two important issues about MPTCP:

- An important design goal of MPTCP is to be no worse than TCP from a security point of view. For example, TCP being vulnerable to Man-in-the-Middle (MiTM) attacks, it is explicitly not a goal for MPTCP to be protected against those attacks.

- The main new threat introduced by MPTCP is related to locator agility. An attacker could send an `ADD_ADDRESS` with his own address and divert part of the data stream, if not all, to himself. Or he could send an `ADD_ADDRESS` with the address of a target, so that it can force a server to flood a chosen victim.

From the above, it comes that the most critical part in MPTCP, from a security point of view, is the subflow establishment. [Bag11] emphasises that a tradeoff is needed between the level of protection we want, and the complexity of the resulting security mechanisms. The currently adopted tradeoff, as defined in [FRHB11]

is that the protection of new subflows is based on the premise that the initial three-way handshake cannot be intercepted by an attacker. Other security mechanisms (e.g. using a Diffie-Hellman exchange) were considered too computationally expensive, and hence no protection is given against the time-shifted attack described in [Bag11] (where an attacker is on-path at the beginning of the communication and then moves away but can still hijack the communication). The protocol could be extended to support other security mechanisms in the future.

Based on the assumption that an attacker cannot see the initial handshake, a key is exchanged in clear text between the peers. `Host A` generates `key_A`, `Host B` generates `key_B`. `Key_A` is sent only in the third segment of the handshake, to allow the utilisation of SYN cookies on the server. Note that the token, introduced in Section 3.3 (Figure 3.3), does not appear anymore in the initial exchange of Figure 3.7. This is because the token is derived from the key: $token_A = hash(key_A)$ and $token_B = hash(key_B)$.

The key is then used to authenticate further subflow establishments, thanks to a Hash-based Message Authentication Code (HMAC), whith the SHA1 [EH06] hash function. Due to the fact that the addresses and ports may be changed on the path to the destination (e.g. by a NAPT device [SE01]), they cannot be included in the HMAC authentication. Instead, [FRHB11] specifies that a pair of random numbers (R_A and R_B), one generated locally and one generated by the peer, must be used as the message to be hashed. `Host A` authenticates itself by showing that it is able to provide the right keyed hash of $R_A + R_B$, using key_A and key_B. Similarly, `Host B` shows that it is able to generate the correct HMAC based on $R_B + R_A$ and using key_B and key_A. Since only `Host A` and `Host B` know the keys, only them can create new subflows with each other (again, given the assumption that the attacker cannot see the initial exchange and that the random keys are long enough to prevent brute force attacks). Moreover, to protect against replay attacks, the numbers R_A and R_B are *nonces*, meaning that they are used only once.

Figure 3.7 shows that the handshakes are actually four-way handshakes, for both the initial and additional subflows. In fact the regular TCP three-way handshake is maintained, but MPTCP forces the server to send an acknowledgement after the three-way handshake to ensure that the final MPTCP security data is received. For the initial exchange, the data is key_A and key_B. For the additional subflows, it is MAC_A. This acknowledgement can and should be used to carry application data, if available.

3.7 Conclusions

In this chapter, after a quick introduction to regular TCP, we have presented Multipath TCP, as defined in [FRHB11]. The main goal of Multipath TCP is to enhance TCP in such a way that two or more paths can be used simultaneously. This has many advantages, including better resource utilisation, better throughput obtained thanks to the resulting pooling of network resources, and smoother reaction to fail-

ures. One important design decision was to make MPTCP *readily deployable*, that is, transparent to applications and to a maximum of network middleboxes. An interesting novelty of MPTCP was the proposal for the *Coupled Congestion Control*. Established from theoretical models, this new algorithm allows moving traffic away from congested paths, while being fair to regular TCP flows in shared bottlenecks. We concluded with a presentation of the security mechanisms currently defined for MPTCP. They are only intended to be no worse than regular TCP. In that matter, MPTCP is not completely successful in its current specification, however, as the so called *time shifted attack [Bag11]* is still possible with MPTCP, while it is not with regular TCP. The other attacks presented in [Bag11] are covered, though.

The next chapter presents our modular implementation of Multipath TCP, in the Linux kernel. We then evaluate the protocol, by using our implementation, in Chapter 5.

Chapter 4

Linux-MPTCP: A modular MPTCP implementation

4.1 Introduction

The Multipath TCP protocol [FRHB11] is a major TCP extension that allows for simultaneous use of multiple paths, while being transparent to the applications, fair to regular TCP flows [RHW11] and deployable in the current Internet. The MPTCP design goals and the protocol architecture that allow reaching them are described in [FRH$^+$11]. Besides the protocol architecture, a number of non-trivial design choices need to be made in order to extend an existing TCP implementation to support Multipath TCP. The objective of this chapter is to achieve a future-proof, yet realistic implementation architecture for MPTCP. In particular we will illustrate in Section 4.4 how MPTCP can be set to take benefit from path management techniques different from the default one, defined in the current specification.

The proposed architecture is expected to be applicable regardless of the Operating System (although the MPTCP implementation described here is done in Linux). Another goal is to achieve the greatest level of modularity without impacting efficiency, hence allowing other multipath protocols to nicely coexist in the same stack.

This chapter is based on the code that we implemented in our Multipath TCP-aware Linux kernel (the version covered here is 0.6) which is available from the address `http://inl.info.ucl.ac.be/mptcp`. We also list configuration guidelines that have proven to be useful in practice. The current version of Linux MPTCP contains around 10000 lines of code, currently without user space program (that is, all the processing takes place in the kernel).

During our work on implementing Multipath TCP, we evaluated other designs. Some of them are not used anymore in our implementation. However, we explain in [BPB11a] the reason why these particular designs have not been considered further, and why some of them could be reconsidered in the future, when more experience is gained.

This chapter is structured as follows. First we propose an architecture that allows supporting MPTCP in a protocol stack residing in an operating system. Then we consider a range of problems that must be solved by an MPTCP stack (compared to a regular TCP stack). In Section 4.5, we give recommendations on how a system administrator could correctly configure an MPTCP-enabled host. Finally, we discuss future work, in particular in the area of MPTCP optimisation.

4.1.1 Terminology

In addition to the concepts introduced in the previous chapter, we define here the following terms, useful to understand the Linux MPTCP implementation. The main ones are illustrated in Figure 4.1.

- **Meta-socket:** A socket structure used to reorder incoming data at the connection level and schedule outgoing data to subflows.
- **Master subsocket:** The socket structure that is visible from the application. If regular TCP is in use, this is the only active socket structure. If MPTCP is used, this is the socket corresponding to the first subflow (hence the name *sub*socket).
- **Slave subsocket:** Any socket created by the kernel to provide an additional subflow. Those sockets are not visible to the application (unless a specific API [SF11] is used). The meta-socket, master and slave subsockets are explained in more detail in Section 4.2.2.
- **Endpoint ID:** Endpoint identifier. It is the tuple that identifies a particular subflow, hence a particular subsocket: (`saddr, sport, daddr, dport`).
- **Fendpoint ID:** First Endpoint identifier. It is the endpoint identifier of the Master subsocket.
- **Connection ID** or **token:** It is a locally unique number, defined in [FRHB11, Section 2], that allows finding a connection during the establishment of new subflows.
- **local_addr_table:** A table of local addresses. It stores, on a per-connection basis, the set of local addresses that an MPTCP connection can use for its subflows.
- **remote_addr_table:** A table of remote addresses. It stores, per connection, the set of remote addresses that an MPTCP connection has learnt from its peer, either through the `ADD_ADDRESS` MPTCP option, or through spontaneous `SYN`s sent by the peer using new addresses.

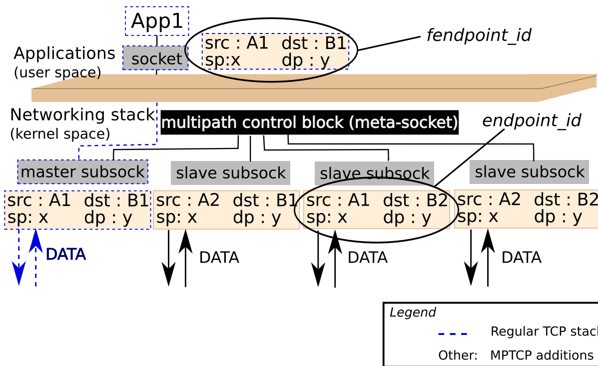

Figure 4.1: Overview of the multipath architecture

4.2 An architecture for Multipath transport

Section 4 of the MPTCP architecture document [FRH+11] describes the functional decomposition of MPTCP. It lists four entities, namely Path Management, Packet Scheduling, Subflow Interface and Congestion Control. These entities can be further grouped based on the layer at which they operate:

- Transport layer: this includes Packet Scheduling, Subflow Interface and Congestion Control, and is grouped under the term “Multipath Transport (MT)”. From an implementation point of view, they all involve modifications to TCP.

- Transport layer and below[1]: path management. Path management can be done in the transport layer, as is the case of the built-in Path Manager (PM) described in the MPTCP architecture document [FRHB11]. That PM discovers paths through the exchange of TCP options of type `ADD_ADDR` or the reception of a `SYN` on a new address pair, and defines a path as an endpoint ID `(saddr, sport, daddr, dport)`. But, more generally, a PM could be any module able to expose multiple paths to MPTCP, located either in kernel or user space, and acting on any OSI layer (e.g. a bonding driver that would expose its multiple links to the Multipath Transport).

Because of the fundamental independence of path management compared to the three other entities, we draw a clear line between both, and define a simple interface that allows MPTCP to benefit easily from any appropriately interfaced multipath technology. In this Section, we stick to describing how the functional elements of MPTCP are defined, using the built-in Path Manager described in [FRHB11], and we leave for Section 4.4 the description of other Path Managers. We describe in the first subsection the precise roles of the Multipath Transport and the Path Manager. Then we detail how they are interfaced with each other.

[1]The exact constraint is that the path management function must “see” packets marked by the Multipath Transport, and hence be located below or inside the transport layer.

4.2.1 MPTCP architecture

Although, when using the built-in PM, MPTCP is fully contained in the transport layer, it can still be organized as a Path Manager and a Multipath Transport Layer (each with its own control and data plane) as shown in Figure 4.2. The Path Manager announces to the Multipath Transport which paths can be used through path indices for an MPTCP connection, identified by the fendpoint ID (first endpoint ID). The fendpoint ID is the tuple `(saddr, sport, daddr, dport)` seen by the application that uniquely identifies the MPTCP connection (an alternative way to identify the MPTCP connection is the connection ID, which is a token as described in [FRHB11, Section 2]). The Path Manager maintains the mapping between the path_index and an endpoint ID. The endpoint ID is the tuple `(saddr, sport, daddr, dport)` that is to be used for the corresponding path index.

Note that the fendpoint ID itself represents a path and is thus a particular endpoint ID. By convention, we always represent the fendpoint ID as path index 1. As explained in [FRH+11, Section 5.6], it is not yet clear how an implementation should behave in the event of a failure of the first subflow. We expect, however, that the Master subsocket should be kept in use as an interface with the application, even if no data is transmitted anymore over it. It also allows the fendpoint ID to remain meaningful throughout the life of the connection. This behaviour has yet to be tested and refined with Linux MPTCP.

Figure 4.2 shows an example sequence of MT-PM interactions happening at the beginning of an exchange. When the MT starts a new connection (through an application `connect()` or `accept()`), it can request the PM to be updated about possible alternative paths for this new connection (step 0 in Figure 4.2). The PM can also spontaneously update the MT at any time (normally when the path set changes). This is step 1 in Figure 4.2. In the example, 4 paths can be used, hence 3 new ones. Based on the update, the MT can decide whether to establish new subflows, and how many of them. Here, the MT decides to establish one subflow only, and sends a request for endpoint ID to the PM. This is step 2. In step 3, the answer is given: `<A2,B2,0,pB2>`. The source port is unspecified to allow the MT ensure the unicity of the new endpoint ID, thanks to the `new_port()` primitive (present in regular TCP as well). Note that messages 1,2,3 need not be real messages and can be function calls instead (as is the case in Linux MPTCP, where an `ADD_ADDR` option causes the Path Manager to directly call the subsocket creation function).

The following options, described in [FRHB11], are managed by the Multipath Transport:

- `MULTIPATH CAPABLE` (`MP_CAPABLE`): Tells the peer that we support MPTCP and announces our local token.

- `MP_JOIN/MP_AUTH`: Initiates a new subflow (MP_AUTH is not yet part of our Linux implementation at the moment)

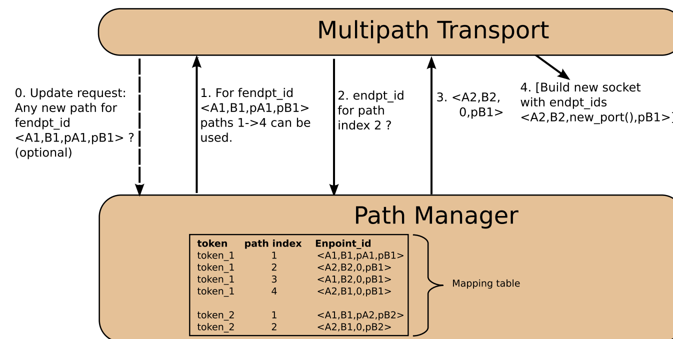

Figure 4.2: Functional separation of MPTCP in the transport layer

- `DATA SEQUENCE NUMBER` (`DSN_MAP`): Identifies the position of a set of bytes in the meta-flow.
- `DATA_ACK`: Acknowledge data at the connection level (subflow level acknowledgements are contained in the normal TCP header).
- `DATA FIN` (`DFIN`): Terminates a connection.
- `MP_PRIO`: Asks the peer to revise the backup status of the subflow on which the option is sent. Although the option is sent by the Multipath Transport (because this allows using the TCP option space), it may be triggered by the Path Manager. This option is not yet supported by our MPTCP implementation.
- `MP_FAIL`: Checksum failed at connection-level. Currently the Linux implementation does not implement the checksum in option `DSN_MAP`, and hence does not implement the `MP_FAIL` option.

The Path Manager applies a particular technology to give the MT the possibility to use several paths. The built-in MPTCP Path Manager uses multiple IPv4/v6 addresses[2] as its means to influence the forwarding of packets through the Internet.

When the MT starts a new connection, it chooses a token that will be used to identify the connection. This is necessary to allow future subflow-establishment SYNs (that is, containing the `MP_JOIN` option) to be attached to the correct connection.

An example mapping table is shown in Figure 4.2. In that example, two MPTCP connections are active. One is identified by `token_1`, the other one with `token_2`. As per [FRHB11], the tokens must be unique locally.

Since the endpoint identifier may change from one subflow to another, the attachment of incoming new subflows (identified by a `SYN + MP_JOIN` option) to the right connection is achieved thanks to the locally unique token.

[2] v6 support is a contribution by Jaakko Korkeaniemi, HIIT

The following options (defined in [FRHB11]) are included in the built-in Path Manager:

- Add Address (`ADD_ADDR`): Announces a new address we own
- Remove Address (`REMOVE_ADDR`): Withdraws a previously announced address

Those options form the built-in MPTCP Path Manager, based on declaring IP addresses, and carry control information in TCP options.

4.2.2 Structure of the Multipath Transport

Our Multipath Transport module handles three kinds of sockets. The relations between them are illustrated in Figure 4.1, page 73. We define them here and use this notation throughout this chapter:

- **Master subsocket:** This is the first socket in use when a connection (TCP or MPTCP) starts. It is also the only one in use if we need to fall back to regular TCP. This socket is initiated by the application through the `socket()` system call. Immediately after a new master subsocket is created, MPTCP capability is enabled by the creation of the meta-socket.
- **Meta-socket:** It holds the multipath control block, and acts as the connection level socket. As data source, it holds the main send buffer. As data sink, it holds the connection-level receive queue and out-of-order queue (used for reordering). We represent it as a normal (extended) socket structure in Linux MPTCP because this allows reusing much of the existing TCP code with few modifications. In particular, the regular socket structure already holds pointers to `SND.UNA`, `SND.NXT`, `SND.WND`, `RCV.NXT`, `RCV.WND` (as defined in [Pos81b]). It also holds all the necessary queues for sending/receiving data.
- **Slave subsocket:** Any subflow created by MPTCP, in addition to the first one (the master subsocket is always considered as a subflow even though it may be in failed state at some point in the communication). The slave subsockets are created by the kernel (not visible from the application) The master subsocket and the slave subsockets together form the pool of available subflows that the MPTCP Packet Scheduler (called from the meta-socket) can use to send packets.

4.2.3 Structure of the Path Manager

In contrast to the Multipath Transport, which is more complex and divided in subentities (namely Packet Scheduler, Subflow Interface and Congestion Control, see Section 4.2), the Path Manager just maintains the mapping table and updates the

event	action
master_sk bound: This is triggered upon either a `bind()` system call, a `connect()`, or when a new server-side socket becomes established.	Discovers the set of local addresses. This set is then maintained in the local_addr_table structure.
`ADD_ADDR` or `SYN + MP_JOIN` received on new address	Updates remote_addr_table correspondingly.
local/remote_addr_table updated	Updates the `mapping_table` structure by adding any new address combinations, or removing the ones that have disappeared. Each address pair is given a path index.
Mapping_table updated	Sends a notification to the Multipath Transport (Figure 4.2, msg 1).
Endpoint ID request received from MT (Figure 4.2, msg 2)	Retrieves the endpoint IDs for the corresponding path index from the mapping table and returns them to the MT (Figure 4.2, msg 3).

Table 4.1: (event,action) pairs implemented in the built-in PM

Multipath Transport when the mapping table changes. The mapping table has been described above (Figure 4.2). We detail in Table 4.1 the set of `(event,action)` pairs that are implemented in the Linux MPTCP built-in Path Manager. For reference, we discuss an earlier architecture for the path management in [BPB11a, Appendix 1].

In the example of Figure 4.2, we show the beginning of an MPTCP connection, where the Path Manager tells the Multipath Transport about the number of useable paths. When the MT asks for the endpoint ID of path index 2, the PM answers with $< A2, B2, 0, pB1 >$. The zero value for the source port indicates that the Path Manager does not mandate any source port (it could if it had some intelligence about an ECMP hash function for example), and the source port is just chosen by the MT in that case (using the regular TCP source port selection algorithm).

4.3 MPTCP challenges for the OS

MPTCP is a major modification to the TCP stack. We have described above an architecture that separates Multipath Transport from path management. Path Management can be implemented rather simply. But Multipath Transport involves a set of new challenges, that do not exist in regular TCP. We first describe how an MPTCP client or server can start a new connection, or a new subflow within a connection. Then we propose techniques (a concrete implementation of which is in Linux MPTCP) to efficiently implement data reception (at the data sink) and data

sending (at the data source).

4.3.1 Charging the application for its CPU cycles

As this chapter is about implementation, it is important not only to ensure that MPTCP is fast, but also that it is fair to other applications that share the same CPU. Otherwise one could have an extremely fast file transfer, while the rest of the system is just hanging. CPU fairness is ensured by the scheduler of the Operating System when CPU cycles are consumed in user space. But inside the kernel, we can choose to run code in “user context”, that is, in a mode where each CPU cycle is charged to a particular application. Or we can (and must in some cases) run code in “software interrupt context”, that is, interrupting everything else until the task has finished. In Linux, the arrival of a new packet on a NIC triggers a hardware interrupt, which in turn schedules a software interrupt that will pull the packet from the NIC and perform the initial processing. The challenge is to stop the processing of the incoming packet in software interrupt as soon as it can be attached to a socket, and wake up the application. With TCP, an additional constraint is that incoming data should be acknowledged as soon as possible, which requires reordering. Van Jacobson has proposed a solution for this [Jac93]: If an application is waiting on a `recv()` system call, incoming packets can be placed into a special queue (called prequeue in Linux) and the application is woken up. Reordering and acknowledgement are then performed in user context. The execution path for outgoing packets is less critical from that point of view, because the vast majority of processing can be done very easily in user context.

In this chapter, when discussing CPU fairness, we will use the following terms:

- **User context:** Execution environment that is under control of the OS scheduler. CPU cycles are charged to the associated application, which allows to ensure fairness with other applications.
- **Software Interrupt context:** Execution environment that runs with a higher priority than any process. Although it is impossible to completely avoid running code in software interrupt context, it is important to minimize the amount of code running in such a context. Note that, here, “software” does not imply a relation with any application, but instead corresponds to kernel code that is programmed for urgent execution, usually after a hardware interrupt or timer expiry.
- **VJ prequeues:** This refers to Van Jacobson prequeues, explained in [Jac93].

4.3.2 At connection/subflow establishment

As described in [FRHB11], the establishment of an MPTCP connection is quite simple, being just a regular three-way exchange with additional options. As explained in Section 4.2.2 this is done in the master subsocket. Currently Linux

MPTCP attaches a meta-socket to a socket as soon as it is created, that is, upon a `socket()` system call (client side), or when a server side socket enters the `ESTABLISHED` state. An alternative solution is described in [BPB11a, Appendix 3].

An implementation can choose the best moment, maybe depending on the OS, to instantiate the meta-socket. However, if this meta-socket is needed to accept new subflows (as in Linux MPTCP), it should be attached at the latest when the `MP_CAPABLE` option is received. Otherwise incoming new subflow requests (`SYN + MP_JOIN`) may be lost, requiring retransmissions by the peer and delaying the subflow establishment.

The establishment of subflows, on the other hand, is more tricky. The problem is that new `SYN`s (with the `MP_JOIN` option) must be accepted by a socket (the meta-socket in the proposed design) as if it was in `LISTEN` state, while its state is actually `ESTABLISHED`. The following are common properties with a `LISTEN` socket:

- Temporary structure: Between the reception of the `SYN` and the final `ACK`, a mini-socket is used as a temporary structure.
- Queue of connection requests: The meta-socket, like a `LISTEN` socket, maintains a list of pending connection requests. There are two such lists. One contains mini-sockets, because the final `ACK` has not yet been received. The second list contains sockets in the `ESTABLISHED` state that have not yet been accepted. "Accepted" means, for regular TCP, returned to the application as a result of an `accept()` system call. For MPTCP it means that the new subflow has been integrated in the set of active subflows.

We can list the following differences with a normal `LISTEN` socket.

- Socket lookup for a `SYN`: When a `SYN` is received, the corresponding socket (in `LISTEN` state) is found by using the endpoint ID. This is not possible with MPTCP, since we can receive a `SYN` on any endpoint ID. Instead, the token must be used to retrieve the meta-socket to which the `SYN` must be attached. A new hashtable must be defined, with tokens as keys.
- Lookup for connection request: In regular TCP, this lookup is quite similar to the previous one (in Linux at least). The 5-tuple is used, first to find the `LISTEN` socket, next to retrieve the corresponding mini-socket, stored in a private hashtable inside the `LISTEN` socket. With MPTCP, we cannot do that, because there is no way to retrieve the meta-socket from the final `ACK` segment. The 5-tuple can be anything, and the token was only present in the `SYN` segments in the first versions of the MPTCP draft. Our Linux MPTCP implementation uses a global hashtable for pending connection requests, where the key is the 5-tuple of the connection request[3].

[3] In the latest version of the MPCTP specification as of this writing [FRHB11], the token has

An implementation must carefully check the presence of the MP_JOIN option in incoming SYNs before performing the usual socket lookup. If it is present, only the token-based lookup must be done. If this lookup does not return a meta-socket, the SYN must be discarded. Failing to do that could lead to mistakenly attach the incoming SYN to a LISTEN socket instead of attaching it to a meta-socket.

Whenever a new path is created at either the client or server side, the corresponding slave subsocket is prepended to a linked list inside the meta-socket, so that it can be accessed by the scheduler to decide where to transmit data. From this linked list, only the slave subsockets that are in ESTABLISHED state can be selected by the scheduler for data transmission.

4.3.3 Locking strategy

In Figure 4.1, we have shown how the meta-socket, master subsocket and slave subsocket relate to each other to form an MPTCP connection. Access to those structures may happen in software interrupt context or user context, or simultaneously in both, on different CPU cores. A careful locking strategy is needed to avoid corrupting the data structures.

The removal of the data structures is another critical point. In particular, an implementation must ensure that the meta-socket remains accessible until the last subsocket disappears, even if any subsocket needs to wait for a timeout.

In this section we will describe the regular TCP locking strategy as it is used in the Linux kernel, before to explain how we extend it to make it safe in a multipath use.

Existing locking strategy for regular TCP: When it comes to locking, it is crucial to understand the difference between the user context and the software interrupt context. We described them succinctly in section 4.3.1. We now explain how they interact from a locking point of view.

- **User context:** Code running in user context can sleep. Locking is ensured typically through mutexes when a code region must be protected from simultaneous accesses that run both in user context (e.g. two processes). If a mutex is locked, the corresponding code simply *sleeps* (calls the scheduler) until the mutex is released.

- **Software Interrupt context:** It is not possible to sleep in software interrupt context, hence mutexes are not an option. Instead, so-called *spinlocks* are used. They simply actively wait in a loop until the lock is released. Because

been added in the final ack to support SYN cookies. The side effect is that it facilitates the lookup described in this paragraph, allowing the global connection request hashtable to be replaced with a local one (specific to each multipath control block). This will make the design closer to that of a regular LISTEN socket, although the lookup must still be done based on the token instead of the 5-tuple.

```
func lock_sock(sk):
    disable_interrupts_local_CPU();
    spin_lock(sk);
    sk->sk_owned_by_user = 1;
    spin_unlock(sk);
    mutex_acquire(sk->sk_mutex);
    enable_interrupts_local_CPU();
    return;
```

Figure 4.3: User context socket locking

of this active loop, the kernel always tries to minimize the amount of time spent in a spinlock.

It is also possible that a piece of code be run from either user context or software interrupt context. An example is the TCP packet transmission mechanism. Packets are transmitted from user context if the transmission is a result of a `sendmsg()` system call. They are transmitted from software interrupt context if a received acknowledgement opened space in the congestion window, allowing to send more segments from the send buffer. In that case both the user context and the software interrupt context need to use a spinlock. Moreover the user context must disable the software interrupts on the local CPU core, otherwise a deadlock situation is possible. The deadlock would happen if a user context grabs a spinlock, and is later interrupted by a software interrupt. The interrupt having higher priority, the user context has no chance to run until the software interrupt is done. On the other hand if the software interrupt tries to grab the lock, it loops forever. Disabling software interrupts (on the local CPU) during the whole locking period is the only solution to prevent this kind of deadlock. Obviously this gives an additional motivation for shortest possible spinlock periods.

Back to TCP, there is a simple way to manage locking in the above example of TCP transmission.

- When sending from user context, disable interrupts and lock the socket.
- When sending from software interrupt context, only lock the socket.

Unfortunately the above mechanism does not respect the requirement to hold a spinlock for very short periods. For that reason the Linux kernel applies the pseudo-code shown in Figure 4.3 for locking in user context. The idea is that instead of holding the spinlock for the whole locking period, it is only used to set a flag. If packet reception happens (in software interrupt context), the flag causes the receive function to enqueue the newly received segment in the backlog instead of processing it. In that case, the segment processing is delayed until the user context releases the lock, as shown in Figure 4.4. Finally, according to Figure 4.4, one could think that the socket spinlock is still held for a long time if the backlog

```
func release_sock(sk):
    mutex_release(sk->sk_mutex);
    disable_interrupts_local_CPU();
    spin_lock(sk);
    if not empty(sk->backlog_queue):
        process_backlog(sk->backlog_queue);
    sk->sk_owned_by_user = 0;
    if is_process_sleeping():
        wake_up_process();
    spin_unlock(sk);
    enable_interrupts_local_CPU();
    return;
```

Figure 4.4: Releasing a socket locked with *lock_sock()* (user context)

contains several segments. The kernel solves this by re-enabling software interrupts after each segment processed from the backlog queue. Moreover, the spinlock is not held during the segment processing itself, because the `sock_owned_by_user` flag already prevents new incoming segments from being processed immediately. The final result is that **Linux ensures that software interrupts are not blocked for more than the time needed to process *one* segment.**

MPTCP general locking policy: Given that MPTCP uses several subflows in parallel, it would be natural to think of running each subflow on a separate core when possible. However, the data converges to a single process in the end, and if it were spread among several CPU local caches, some efficiency would be lost (as data would need to be moved between the caches). For this reason, **we use a single lock for the whole MPTCP connection[4].**

A lock is associated to a socket structure. In MPTCP, we could potentially make use of $n + 1$ locks if n flows are available, as the meta-socket itself uses a socket structure. Given the above design decision to use a single socket lock, a natural choice would be to use the meta-socket lock for any packet processing activity, on any subsocket. This would complicate the fallback procedure to regular TCP, which is needed when the peer does not support MPTCP or the network drops some options. When performing a fallback, the meta-socket is destroyed and only the master-socket lock can be used. This motivates our second design choice: **Socket locking for MPTCP is performed by using only the master socket lock**. Note that the network path corresponding to the master socket could fail at some point, but in that case only the path stops being used. The socket structure remains available for locking and providing the interface with the application.

The locking procedure is the same as in regular TCP (see Figure 4.3). Regard-

[4]However, for the most efficient result, it will be necessary to ensure that all subflows run on the same core, which is not the case yet in Linux MPTCP.

ing the backlog queues, we use the subsocket-specific backlog queues as in regular TCP. Doing so avoids needing to perform a second tuple-based socket lookup when processing the backlog. We modify the algorithm of Figure 4.4 to include an iteration over all backlog queues, so as to ensure that all backlogs are empty after the *release_sock()* call.

Attaching a new subflow to the meta-socket: When we attach a new subflow to the meta-socket, we extend the linked list of available subflows. This requires to grab the master socket lock, so as to avoid corrupting the linked list. Hence, we need a backlog-queue mechanism for the processing of incoming new subflows (`SYN+JOIN`s). We use the meta-socket backlog queue for that. If a `SYN+JOIN` is received and the socket is locked by the userspace (`sock_owned_by_user` is 1), the segment is enqueued in the meta-socket backlog queue and processed during the *release_sock()* operation. Likewise, the final `ACK` of the three-way handshake for a new subflow may be queued in the meta-socket backlog, causing the full socket to be created and attached to the meta-socket in the *release_sock()* function.

Reference counting: Reference counting is the mechanism that ensures the clean removal of data structures from the Linux kernel. Each user increases the reference count, before using the structure and decreases the counter when it is done with its task. This guarantees that the structure is properly released (avoiding memory leaks), and prevents the release from happening when a user still holds a reference to it (avoiding pointer faults). In the case of sockets structures, a *sock_hold()* function is called to increase the reference count. *sock_put()* is used to decrease it, and releases the structure if the counter reaches 0.

For MPTCP, as opposed to the locking mechanism that uses only the master socket lock, we use the reference counts from all the subsockets. This is needed because a slave subsocket does not necessarily disappear simultaneously with the master subsocket. However, we note that the master subsocket and the meta-socket are tied together, so we use the same reference count for them. We define a reference count policy that guarantees the integrity of the pointers. The following *sock_hold()/sock_put()* pairs are applied:

- meta-socket allocation/destruction: *sock_hold/put(master_sk)*. This ensures that the master socket does not disappear before the meta-socket. The reverse is possible however, and happens in case of fallback to regular TCP.

- new slave subsocket/release of slave subsocket: *sock_hold/put(master_sk)*. This ensures that the master and meta-sockets do not disappear until the last slave subsocket has been released.

- packet reception: *sock_hold/put(**sub_sk**)*, resp. at the beginning or end of TCP reception procedure. This behaviour is unchanged compared to regular TCP. This is safe because the reference count increase prevents the release of

the subsocket. In turn, the presence of the subsocket prevents decrementing the master socket reference count.

- timer started/stopped: *sock_hold/put(**sub_sk**)*. This is also the same behaviour as regular TCP.

The meta-socket is released either during the fallback operation or at the end of the master socket release.

4.3.4 Subflow management

Further research is needed to define the appropriate heuristics to solve problems with subflow management. Initial thoughts are provided in Section 4.6.

Currently, in a Linux MPTCP client, the Multipath Transport tries to open all subflows advertised by the Path Manager. On the other hand, the server only accepts new subflows, but does not try to establish new ones. The rationale for this is that the client is the connection initiator. New subflows are only established if the initiator requests them. This is subject to change in future releases of our MPTCP implementation.

4.3.5 At the data sink

There is a symmetry between the behaviour of the data source and the data sink. Yet, the specific requirements are different. The data sink is described in this section while the data source is described in the next section.

Receive buffer tuning

MPTCP needs that the receive buffer be larger than the sum of the buffers required by the individual subflows. The reason for this and proper values for the buffer are explained in [FRH+11, Section 5.3]. Not following this could result in the MPTCP throughput being capped at the bandwidth of the slowest subflow.

An interesting way to dynamically tune the receive buffer according to the bandwidth/delay product (BDP) of a path, for regular TCP, is described in [FF01] and implemented in recent Linux kernels. It uses the `COPIED_SEQ` sequence variable (sequence number of the next byte to copy to the application buffer) to count, every RTT, the number of bytes received during that RTT. This number of bytes is precisely the BDP. The tuning algorithm is conservative, in that it never shrinks a previously increased receive buffer. The accuracy of this receive buffer tuning is directly dependent on the accuracy of the RTT estimation. Unfortunately, the data sink does not have a reliable estimate of the SRTT. To solve this, [FF01] proposes two techniques:

1. Using the timestamp option (quite accurate).

2. Computing the time needed to receive one `RCV.WND` [Pos81b] worth of data. It is less precise and Linux considers as outliers such RTT estimates that are more than 8 times larger than the last estimate.

The receive window advertised by MPTCP is shared by all subflows. Hence, no per-subflow information can be deduced from it, and the second technique from [FF01] cannot be used[5].

As mentioned in [FRH$^+$11], the connection-level receive buffer that is allocated should be $2 * \sum_i BW_i * RTT_{max}$, where BW_i is the bandwidth seen by subflow i and RTT_{max} is the maximum RTT estimated among all the subflows. We achieve this in Linux MPTCP by slightly modifying the first tuning algorithm from [FF01], and disabling the second one. The modification consists in counting on each subflow, every RTT_{max}, the number of bytes received during that time on this subflow. Per subflow, this provides its contribution to the total receive buffer of the connection.

Receive window handling: The original design of the MPTCP protocol used subflow-specific receive windows. This looks interesting indeed as a way for the receiver to perform ingress load balancing, and it is also close to regular TCP behaviour, hence simplifying implementations. However, this can potentially create deadlock scenarios and we detected them while developing our implementation. Consider two hosts A and B, connected through paths 1 and 2 (with data flowing from A to B, see Figure 4.5). They can initially use both paths to exchange data. If path 1 suddenly fails, the last congestion window sent on path 1 must be retransmitted on path 2, but this only happens after a TCP RTO (Retransmission Timeout). Until the expiration of the timer on path 1, path 2 has enough time to transmit new data, progressively saturating the receive buffer in host B, because no data can be delivered to the application until the lost data from path 1 has been retransmitted. If path 2 is fast enough, host A will eventually receive a zero-window on path 2. This is a deadlock situation: host A needs to retransmit lost data to unblock the communications, but both its paths are closed, one fails and the receive window of the other one is zero.

Deadlock solution: The problem of the above deadlock is that subflow-specific receive windows cannot provide any information about the shared receive buffer. This is solved by changing the receive window semantic, so that it is understood as a *connection-level* receive window, that is, it reflects the size of the *shared receive buffer* and its left-edge is given by the `Data ACK` instead of the subflow-level acknowledgement. If we consider again the deadlock scenario described above, where one congestion window worth of data must be retransmitted on subflow 2, this will be possible now, as the receiver will not send a zero-window. If its receive

[5] More precisely, the time needed to receive one `RCV.WND` in MPTCP cannot exceed the maximum RTT on the set of paths used to transmit that amount of data. This could be used to compute an upper bound on the receive buffer, however the result is correct only if *all* paths are used in the measurement period.

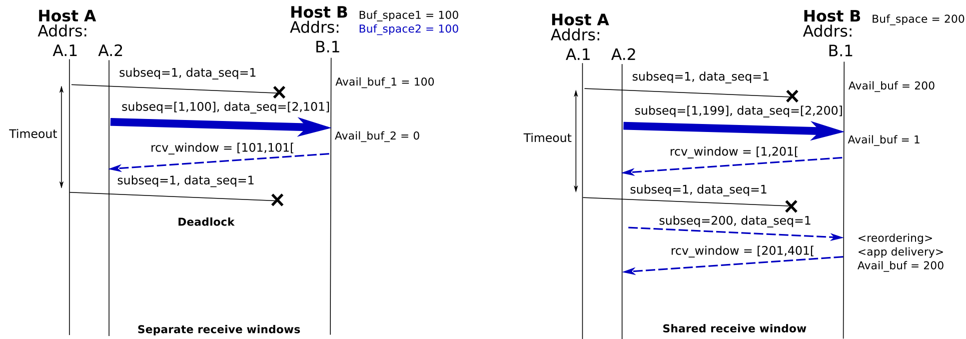

Figure 4.5: A deadlock may happen if MPTCP uses separate receive windows

buffer comes to saturation, it will only prevent the transmission of any *new* data, still accepting lost data, since the left-edge of the connection-level receive window does not move forward until the lost data has been retransmitted. This change is now part of the main protocol specification [FRHB11].

Receive queue management

As advised in [FRHB11, Section 3.3.1], *"subflow-level processing should be undertaken separately from that at connection-level"*. This also has the side-effect of allowing much code reuse from the regular TCP stack.

A regular TCP stack (in Linux at least) maintains a receive queue (for storing incoming segments until the application asks for them) and an out-of-order queue (to allow reordering).

In Linux MPTCP, the subflow-level receive-queue is not used[6]. Incoming segments are reordered at the subflow-level, just as if they were plain TCP data. But once the data is in-order at the subflow level, it can be immediately handed to MPTCP (See [FRH$^+$11, Figure 7]) for connection-level reordering. The role of the subflow-level receive queue is now taken by the MPTCP-level receive queue. In order to maximize the CPU cycles spent in user context (see Section 4.3.1), VJ prequeues can be used just as in regular TCP. We have recently added support for them, and VJ prequeues will be included in the upcoming version 0.7 of Linux MPTCP. Once it is ensured that the data is processed in user context whenever possible, one should also take care of handling efficiently the MPTCP out-of-order queue. Whereas regular TCP handles small queues (the source of the reordering being the network and packet losses), MPTCP may need to handle very large reordering queues (the source of reordering being the use of multiple independent flows). To handle this, we have implemented several mechanisms:

- *Segment aggregation*: Whenever several segments are contiguous in the reordering queue at the connection level, but not yet ready for inclusion in

[6] An alternative design, where the subflow-level receive queue is kept active and the MPTCP receive queue is not used, is discussed in [BPB11a, Appendix 4].

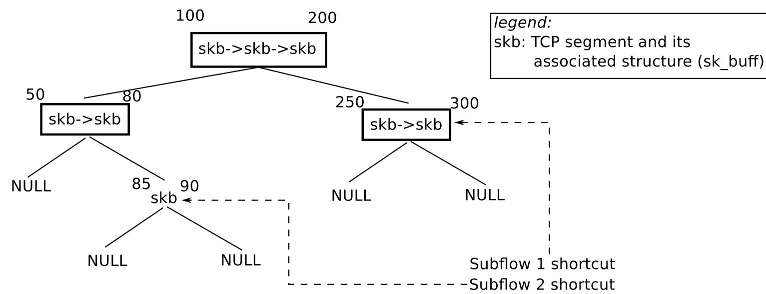

Figure 4.6: Structures used to optimise segment reordering at the receiver

the receive queue, we aggregate them in a *node*. A *node* contains a list of contiguous segments, but requires only one iteration when searching the out-of-order queue for a particular sequence number. The memory required to allocate a new node is taken from a cache to reduce the allocation time.

- *Shortcut pointers*: It is often the case that more than one segment is sent over the same subflow. The reason is that when a subflow becomes available for sending data, the scheduler feeds it with as many contiguous segments as it is able to send. We optimise for this by maintaining, in each subflow, a shortcut to the slot in the out-of-order queue where the next segment is expected to arrive. Upon arrival of a new segment, the search starts from the shortcut slot. This improvement is very interesting as it provides a direct hit for more than 80% of the received segments (more details on the evaluation can be found in Section 5.2).

- *Binary Search Tree (BST)*: To efficiently handle the case where the shortcut does not work, we have implemented a Binary Search Tree, as a replacement for the traditional out-of-order queue. While apparently better than the queue because of its expected logarithmic lookup time (which actually depends on the extent to which the tree is balanced), it does not necessarily improve the receiver performance because other operations are more expensive. In particular, getting a pointer to a neighbour requires dereferencing several node pointers. For instance, in a BST, the closest neighbour of a node n on the left is the right-most child of its left child. If n has no left child, then the closest left neighbour is the closest ancestor whose child in the direction of n is a right child. In a simple queue, the left neighbour is one pointer away. Initial experiments tend to show that while the BST does bring benefits over a regular queue, simply adding shortcut pointers (without changing the whole structure) gives even better results. More details are presented in Section 5.2.

The three optimisation mechanisms are illustrated together in Figure 4.6, although they can be used separately. The bolded boxes show *segment aggregation*, where several contiguous segments (skbs) are stored in a node as a linked list to reduce the size of the data structure. The nodes are not associated to a particular

subflow (except for the shortcuts) because this structure is used at the connection level, where the data is not attached anymore to a particular subflow and uses only the connection-level data sequence numbers. The dashed arrows show shortcut pointers. In the picture, we can see that subflow 1 currently expects to receive the segment with data sequence number 300, while subflow 2 expects the data sequence number 90. The main structure illustrates the binary search tree.

Finally, regular TCP bypasses the receive queue when a segment arrives in order, and copies the content directly to user space. This saves receive queue processing time. Similarly, when MPTCP receives data in order from the connection level point of view, we copy directly data from user space to kernel space.

To summarise, receive-side processing can be optimised at several levels. Because MPTCP requires much more reordering compared to regular TCP, it is even more important to ensure that (i) this processing happens in user context and (ii) it requires a minimal amount of iterations. (i) is ensured by Van Jacobson prequeues, but also by the backlog (segments received when the socket is locked are always processed in user context by design). (ii) requires MPTCP-specific modifications that we presented here and that will be evaluated in Section 5.2. Lastly, when reordering is not needed, the processing can be made even faster by bypassing the connection-level receive queue, just like regular TCP already does.

Scheduling Data ACKs

As specified in [FRHB11, Section 3.3.2], `Data ACKs` not only help the sender in having a consistent view of which data has been correctly received at the connection level. They are also used as the left edge of the advertised receive window.

In regular TCP, if a receive buffer becomes full, the receiver announces a zero receive window. When finally some bytes are delivered to the application, freeing space in the receive buffer, a duplicate `ACK` is sent to act as a window update, so that the sender knows it can transmit again. Likewise, when the MPTCP shared receive buffer becomes full, a zero window is advertised. When some bytes are delivered to the application, a duplicate `Data ACK` must be sent to act as a window update. Such an important `Data ACK` should be sent on all subflows, to maximize the probability that at least one of them reaches the peer. If, however, all `Data ACKs` are lost, there is no other option than relying on the window probes periodically sent by the data source, as in regular TCP.

In theory a `Data ACK` can be sent on any subflow, or even on all subflows, simultaneously. As of version 0.5, Linux MPTCP simply adds the `Data ACK` option to all outgoing segments (regardless of whether it is data or a pure `ACK`). There is thus no particular `Data ACK` scheduling policy. The only exception is for a window update that follows a zero-window. In this case, the behaviour is as described in the previous paragraph.

4.3.6 At the data source

In this section we discuss the same topics as in the previous section, in the case of a data sender. The sender does not have the same view of the exchange, because one has information that the other can only estimate. Also, the data source sends data and receives acknowledgements, while the data sink does the reverse. This results in a different set of problems to be dealt with by the data source.

Send buffer tuning

As explained in [FRH$^+$11, Section 5.3], the send buffer should have the same size as the receive buffer. At the sender, we don't have the RTT estimation problem described in Section 4.3.5, because we can reuse the built-in TCP SRTT (smoothed RTT). Moreover, the sender has the congestion window, which is itself an estimate of the BDP, and is used in Linux to tune the send buffer of regular TCP. Unfortunately, we cannot use the congestion window with MPTCP, because the buffer equation does not involve the product $BW_i * delay_i$ for the subflows (which is what the congestion window estimates), but it involves $BW_i * delay_{max}$, where $delay_max$ is the maximum observed delay across all subflows.

An obvious way to compute the contribution of each subflow to the send buffer would be: $2 * (cwnd_i/SRTT_i) * SRTT_{max}$. However, some care is needed because of the variability of the SRTT (measurements show that, even smoothed, the SRTT is not quite stable). Currently Linux MPTCP estimates the bandwidth periodically by checking the sequence number progress. This however introduces new mechanisms in the kernel.

Send queue management

As Multipath TCP involves the use of several TCP subflows, a scheduler must be added to decide where to send each byte of data. We have evaluated two possible places for the Linux MPTCP scheduler. One option is to schedule data as soon as it arrives from the application buffer. This option, consisting in *pushing* data to subflows as soon as it is available, was implemented in older versions of Linux MPTCP and is now abandoned (it is described in [BPB11a, Appendix 5]). Another option is to store all data centrally in the Multipath Transport, inside a shared send buffer (see Figure 4.7). Scheduling is then done at transmission time, whenever any subflow becomes ready to send more data (usually due to acknowledgements having opened space in the congestion window). In that scenario, the subflows *pull* segments from the shared send queue whenever they are ready. Note that several subflows can become ready simultaneously, if an acknowledgement advertises a new receive window that opens more space in the shared send window. For that reason, when a subflow pulls data, the Packet Scheduler runs and other subflows may be fed at the same time. This approach, similar to the one proposed in [HS02], has several advantages:

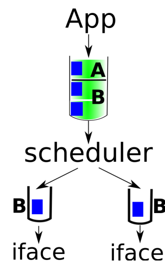

Figure 4.7: Send queue configuration

- Each subflow can easily fill its pipe (as long as there is data to pull from the shared send buffer, and the scheduler is not applying a policy that restricts the subflow).
- If a subflow fails, it will no longer receive acknowledgements, and hence will naturally stop pulling data from the shared send buffer. This removes the need for an explicit "failed state", to ensure that a failed subflow does not receive data (as opposed to e.g. SCTP-CMT, that needs an explicit marking of failed subflows by design, because it uses a single sequence number space [BDI+11]).
- Similarly, when a failed subflow becomes active again, the pending segments of its congestion window are finally acknowledged, allowing it to pull again data from the shared send buffer. Note that in such a case, the acknowledged data is normally just dropped by the receiver, because the corresponding segments have been retransmitted on another subflow during the failure time.

Despite the adoption of that approach in Linux MPTCP, there are still two drawbacks:

- There is one single queue, in the Multipath Transport, from which all subflows pull segments. In Linux, queue processing is optimised for handling segments, not bytes. This implies that the shared send queue must contain pre-built segments, hence requiring the *same* MSS to be used for all subflows. We note however that today, the most frequently negotiated MSS is around 1380 bytes [BPB11b], so this approach sounds reasonable. Should this requirement become too constraining in the future, a more flexible approach could be devised (e.g. supporting a few Maximum Segment Sizes).
- Because the subflows pull data whenever they get new free space in their congestion window, the Packet Scheduler must run at that time. But that time most often corresponds to the reception of an acknowledgement, which happens in software interrupt context (see Section 4.3.1). This is both unfair to other system processes, and slightly inefficient for high speed flows. The problem is that the packet scheduler performs more operations than the

> usual "copy packet to NIC". One way to solve this problem would be to have a small subflow-specific send queue, which would actually lead to a hybrid architecture between the pull approach (described here) and the push approach (described in [BPB11a, Appendix 5]). Doing that would require solving non-trivial implementation problems, though, and requires further study.

As shown in Figure 4.7, a segment first enters the shared send queue. Then, when reaching the bottom of that queue, it is pulled by some subflow. In Linux MPTCP, the segment data is kept in the shared send queue (B portion of the queue) until it is acknowledged. The A portion of the shared send queue contains data that has never been transmitted on any subflow. Any pull operation takes data from the bottom of the A "sub-queue". When a subflow pulls a segment, it actually only copies the control structure (`struct sk_buff`) (which Linux calls packet cloning) and increments its reference count. The pulling operation is a bit special in that it can result in sending a segment over a different subflow than the one which initiated the pull. This is because an acknowledgement received on any subflow can unblock all subflows, given the receive window is considered at the connection level. Hence, the Packet Scheduler is run as part of the pull, which can result in selecting any subflow. In most cases, though, the subflow which originated the pull will get fresh data, given it has space for that in the congestion window. Note that the subflows have no A portion in Figure 4.7, because they immediately send the data they pull.

Note on the send window: A subflow can be stopped from transmitting by the congestion window, but also by the send window (that is, the receive window announced by the peer). Given that the receive window has a connection level meaning, a `Data ACK` arriving on one subflow could unblock another subflow. Implementations should be aware of this to avoid stalling part of the subflows in such situations. In the case of Linux MPTCP, that follows the above architecture, this is ensured by running the Packet Scheduler at each pull operation.

Scheduling data

As several subflows may be used to transmit data, MPTCP must select a subflow to send each byte of data. First, we need to know which subflows are available for sending data. The mechanism that controls this is the congestion controller, which maintains a per-subflow congestion window. The aim of a multipath congestion controller is to move data away from congested links, and ensure fairness when there is a shared bottleneck. The handling of the congestion window is explained in [RHW11, WRGH11]. Its implementation in Linux MPTCP has been contributed by Christoph Paasch, and the implementation is documented in [BPB11b, BPB11a].

Whenever the Congestion Controller (described above) allows new data for at least one subflow, the Packet Scheduler executes. When only one subflow is available the Packet Scheduler just decides which packet to pick from the A section of the shared send buffer (see Figure 4.7). Currently Linux MPTCP picks the bottom most segment. If more than one subflow is available, there are three decisions to take:

- *Which of the subflows to feed with fresh data:* As the only Packet Scheduler currently supported in Linux MPTCP aims at filling all pipes, it always feeds data to all subflows as long as there is data to send. Other schedulers would be possible. One example would be to keep a path for backup only. In that case the path would be used only when all other paths fail. This makes sense if a link is very expensive compared to others (e.g. 3G vs WiFi). Another possible scheduler would be to use a path in *overflow mode*. In that configuration the scheduler would use the path only when all others already have a full congestion window worth of data in flight, and cannot accept more data until an `ACK` has been received.

- *In what order to feed selected subflows:* when several subflows become available simultaneously, they are fed by order of time-distance to the client. We define the time-distance as the time needed for the packet to reach the peer if transmitted on a particular subflow. This time depends on the RTT, bandwidth and queue size (in bytes), as follows:

  $$time_distance_i = queue_size_i/bw_i + RTT_i$$

  Given that with the architecture described in Section 4.3.6, the subflow-specific queue size cannot exceed a congestion window, the $time_distance$ becomes $time_distance_i \cong RTT_i$. This scheduling policy favours fast subflows for application-limited communications (where all subflows need not be used). However, for network-limited communications, this scheduling policy has little effect because all subflows will be used at some point, even the slow ones, to try minimizing the connection-level completion time.

- *How much data to allocate to a single subflow:* this question concerns the granularity of the allocation. Using large allocation units allows for better support of TCP Segmentation Offload (TSO). TSO allows the system to aggregate several times the MSS into one single segment, sparing memory and CPU cycles, by leaving the fragmentation task to the NIC. However, this is only possible if the large single segment is made of contiguous data, at the subflow level and the connection level. On the other hand, using small allocation units allows more evenly using the subflows for low-traffic applications (such applications could end up using only one of the subflows with large allocation units). Our implementation currently allocates on a per-MSS basis as TSO is not supported yet.

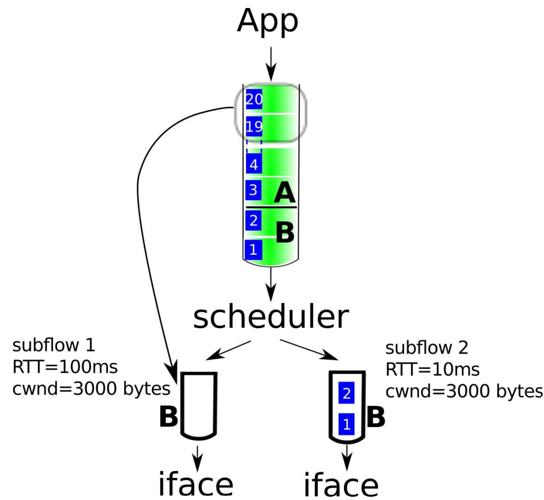

Figure 4.8: Send queue configuration

Note about segment aggregation: When scheduling data to subflows, an implementation must be careful that if two segments are contiguous at the subflow-level, but non-contiguous at the connection level, they cannot be aggregated into one. As the Linux kernel merges segments when it is under memory pressure, it could easily decide to merge non-contiguous MPTCP segments, simply because they look contiguous from the subflow viewpoint. This must be avoided, because the DATA_SEQ mapping option would lose its meaning in such a case, corrupting the bytestream. Our implementation does that by doing an additional check on the data sequence number before to merge a segment with another one.

Note about the shared send queue: Our scheduler currently takes the bottommost segment from the shared send queue, whenever it is called. A possible improvement would be to intelligently choose *which* segment to allocate, from the shared send queue. We show in Figure 4.8 an example case where this would be useful. In this example, two subflows are used. Subflow 1, on the left, has an estimated RTT of $100ms$, while subflow 2, on the right, has an RTT of $10ms$. They both have a current congestion window of 3000 bytes. As the congestion window approximates the Bandwidth-Delay Product (BDP), we can evaluate BW_1 and BW_2 resp. to $30000 bytes/s$ and $300000 bytes/s$. Figure 4.8 shows that subflow 1 is asking for new data to the scheduler, because its congestion window has been fully acknowledged. On the other hand, the faster subflow 2 is not available currently.

In such a situation, our current scheduler would allocate segments 3 and 4 (assuming a MSS of 1500 bytes) to subflow 1, and would have received the corresponding acknowledgements $100ms$ later. This is clearly suboptimal, as by waiting a maximum of $10ms$, subflow 2 would have been able to transmit, allowing

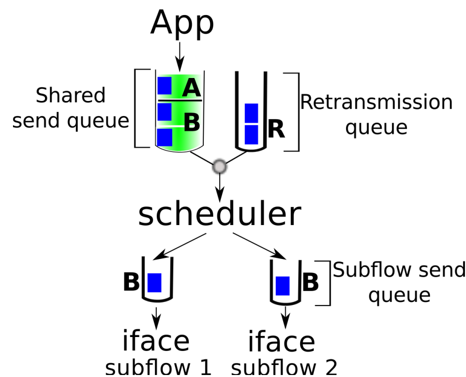

Figure 4.9: Retransmission mechanism

the data to be acknowledged within $20ms$ instead of $100ms$. Continuing this reasoning, we see that as many as 18 segments (with a MSS of 1500 bytes) would be faster acknowledged if sent over subflow 2. The modified scheduler would then take directly segments 19, 20 and feed them to subflow 1. Doing so would reduce the connection-level reordering at the receiver, hence the required amount of receive buffer and the burstiness of packet delivery to the application.

Handling retransmissions

Data retransmission is more complicated in MPTCP compared to regular TCP, because a lost segment may be retransmitted on any subflow. Figure 4.7 was slightly simplified because it did not take retransmissions into account. The full transmission mechanism is illustrated in Figure 4.9. A specific queue is used for retransmissions. It is located above the scheduler because these segments can be retransmitted on any subflow. Since retransmissions can block the progression of the data transfer, the scheduler does not take data from the shared send queue until the shared retransmission queue is empty.

The retransmission mechanism is best explained through an example. Consider a host with two subflows (Figure 4.9). Subflow 1 receives an acknowledgement. It updates its congestion window and asks the scheduler for new data. Further consider that the retransmission queue is empty at this point, so that the scheduler allocates segment S from the shared send queue to subflow 1. The segment is *cloned*, that is, it remains in the shared send queue (until it is acknowledged at the connection level), and a new structure (say, S_1) pointing to the segment data is stored in the send queue of subflow 1. S_1 is given subflow sequence numbers from the sequence number space of subflow 1. A bitmap in S (`path_bitmap(`S`)`) is updated to remember that S has currently been cloned only on path 1 (and hence can still be cloned to path 2 later if needed).

Assume that S_1 is never acknowledged. After a subflow-level timeout, it is retransmitted. But **since we consider a timeout as an indication of potential fail-**

```
func timeout(subflow sf):
    for each segment S in unacked_segments(sf):
        for each path i in available_paths():
            if i is not in path_bitmap(S):
                /* There is at least one possible alternative
                 * subflow for retransmission */
                enqueue_segment(clone(S), retransmit_queue);
                break;
    schedule();
    return;
```

Figure 4.10: Retransmission algorithm

ure, we decide that it is worth trying to retransmit as well on another subflow[7]. Our retransmission algorithm is detailed in Figure 4.10. In our example, the result of the algorithm is that subflow 2 is found to have never been allocated segment S, so S is appended to the retransmit queue. The scheduler is immediately run. In our case the only allocation option is subflow 2. However, **if several subflows are eligible for retransmission (i.e. have never been allocated the segment before), the scheduler decides which one will receive the next retransmission**. This is important as it allows using the same scheduling policy as for normal transmissions.

Now, consider that subflow 2 (our only option) is currently not accepting data, because it has already sent one full congestion window and is waiting for the acknowledgements. In that case the scheduler returns immediately. But as soon as an acknowledgement opens new space in the congestion window of subflow 2, the subflow initiates a *pull* operation. The scheduler then takes data from the retransmission queue, and only when it is empty does it take new data from the send queue.

In the general case, a segment is initially allocated to one subflow, judged the best one by the scheduler. If the subflow experiences a timeout before the segment has been acknowledged, MPTCP tries to transmit on one more subflow. In the worst case (where the endpoint becomes fully disconnected from the Internet), the unacknowledged segments are finally sent over all available subflows, with exponential backoff as in regular TCP. This situation is resolved by either a time out at the connection level or the recovery of any of the subflows. The full MPTCP transmission mechanism is summmarized in Table 4.2.

[7]In contrast, fast retransmits are only done on the same subflow as they indicate a single loss and do not necessarily imply a reduction of the path quality.

event	action
Segment acknowledged at the subflow level	Remove the reference to the segment from the subflow-level queue
Segment acknowledged at connection level	Remove the reference to the segment from the connection-level queue
Timeout (subflow-level)	Push the segment to the best running subflow (according to the Packet Scheduler). If no subflow is available, push it to the retransmit queue.
3 duplicate ACKs	Retransmit segment only on local subflow
Ready to put new data on the wire (normally triggered by an incoming ack)	If the retransmit queue is not empty, first pull from there. Otherwise, take new segment(s) from the connection level send queue (A portion).

Table 4.2: (event,action) pairs implemented in the Multipath Transport queue management

Related work: The most important retransmission mechanism for transport layer multipath that has been proposed before was for SCTP-CMT [IAS07]. This paper emphasises that poor retransmission choices may significantly increase the flow completion time due to a problem that the authors call *receive buffer blocking*. *Receive buffer blocking* happens when a fast sender is slowed down by a small receive window, that the receiver is forced to advertise because its receive buffer comes to saturation. A receive buffer can easily be saturated when one subflow experiences a time out, requiring from the receiver to store all the data coming from other subflows until the lost segments are finally retransmitted. [IAS07] explains that the only way to mitigate this problem is to intelligently choose the subflow used to retransmit. They evaluate five possible retransmission policies, that we will compare with our unified approach (The quoted text in front of the policy name is the definition of the policy according to [IAS07]):

- **RTX-SAME**: *"Once a new data chunk is scheduled and sent to a destination, all retransmissions of the chunk are sent to the same destination"*. In MPTCP, even if it is decided to retransmit on another subflow, a segment is *always* retransmitted *as well* on the initial subflow, with exponential backoff. This is actually a requirement from [FRHB11], that comes from the constraint that *once subflow sequence numbers are assigned to a segment and sent to the network, they cannot be re-assigned to other data*[8]. This constraint is not present in SCTP-CMT because it does not use subflow-level sequence numbers.

[8] The rationale for this is that middleboxes could replay old segments, confusing the receiver with different data attached to the same sequence numbers

- **RTX-ASAP**: *"A retransmission of a data chunk is sent to any destination for which the sender has cwnd space available at the time of retransmission. If multiple destinations have available cwnd space, one is chosen randomly."* In MPTCP, the scheduler is run to decide where to send the data. But we have explained previously that only flows with available $cwnd$ space are eligible for selection by our scheduler. Hence, in some way, RTX-ASAP is also applied by our schedulder. However, RTX-ASAP uses random selection as tie-break. We use shortest delivery time (estimated through the RTT and queue size as explained in the previous subsection).

- **RTX-CWND**: *"A retransmission is sent to the destination for which the sender has the largest cwnd. A tie is broken by random selection."* This approach is quite different from what we do, because it may accept to delay the retransmission if that allows sending on a subflow with higher congestion window. We discuss hereafter a possible improvement to our mechanism, that would also delay the transmission in the hope that the segment finally reaches the peer faster.

- **RTX-SSTHRESH**: *"A retransmission is sent to the destination for which the sender has the largest ssthresh. A tie is broken by random selection."* This is very similar to the previous policy, and gives indeed similar results according to [IAS07].

- **RTX-LOSSRATE**: *"A retransmission is sent to the destination with the lowest loss rate path. If multiple destinations have the same loss rate, one is selected randomly."* This policy is not explicitly included in our Linux MPTCP. However, paths with low loss rates have a much higher probability to be chosen by our retransmission mechanism, because the coupled congestion control favours paths with low loss rates.

Discussion: According to the simulations results of [IAS07]. The best results are obtained from the policies RTX-CWND and RTX-SSTHRESH. Those two policies are precisely the most different ones compared to our implementation. However, we note that our retransmission policy is not any of the other three policies considered by [IAS07], but instead a *combination* of them. Finally, our retransmission mechanism could be further improved as described in Figure 4.8. That modification would have in common with RTX-CWND and RTX-SSTHRESH that the scheduler does not necessarily allocate a segment to the first available subflow, but instead takes into account the time needed for that segment to reach the peer.

4.3.7 At connection/subflow termination

In Linux MPTCP, subflows are terminated only when the whole connection terminates, because the heuristic for terminating subflows (without closing the connection) is not yet mature, as explained in Section 4.3.4.

At connection termination, an implementation must ensure that all subflows plus the meta-socket are cleanly removed. **The obvious choice to propagate the `close()` system call on all subflows does not work.** The problem is that a `close()` on a subflow appends a `FIN` at the end of the send queue. If we transpose this to the meta-socket, we would append a `Data FIN` on the shared send queue (see Section 4.3.6). That operation results in the shared send queue not accepting any more data from the application, which is correct. But it also results in the subflow-specific queues not accepting any more data from the shared send queue. The shared send queue may however still be full of segments, which will never be sent because all subflows are closed.

Inferred implementation rule: Upon a `close()` system call, an implementation must refrain from sending a `FIN` on all subflows, unless the implementation uses an architecture with no connection-level send queue (like the one described in [BPB11a, Appendix 5]). Even in that case, it makes sense to keep all subflows open until the last byte is sent, to allow retransmission on any path, should any one of them fail.

Currently, upon a `close()` system call, Linux MPTCP appends a `Data FIN` to the connection-level send queue. Only when that `Data FIN` reaches the bottom of the send queue is the regular `FIN` sent on all subflows (which requires that the retransmission queue be empty as well).

Note: In the Linux MPTCP behaviour described above, a connection could still stall near its end if one path fails while transmitting its last congestion window of data (because the maximum size of the subflow-specific send queue is `cwnd`). A way to avoid this has been proposed: instead of sending the FIN together with the `Data FIN`, send the `Data FIN` alone and wait for the corresponding `Data ACK` to trigger a `FIN` on all subflows. This however prolongs the duration of the overall connection termination by one RTT.

4.4 Implementing alternative Path Managers

In Section 4.2, the Path Manager is defined as an entity that maintains a (path_index, endpoint ID) mapping. This is enough in the case of the built-in Path Manager, because the segments are associated to a path within the socket itself, thanks to its endpoint ID. However, it is expected that other Path Managers may need to apply a particular action, on a per-packet basis, to associate them with a path. Example actions could be writing a number in a field of the segment (which we call *colouring* a packet for clarity) or choosing a different gateway than the default one in the routing table. In an earlier version of Linux MPTCP, based on a Shim6 Path Manager, the action was used and consisted in rewriting the addresses of the packets.

path index	Action (Write x in DSCP)
1	red (write 1)
2	blue (write 2)
3	green (write 3)

Table 4.3: Example mapping table for a colouring PM

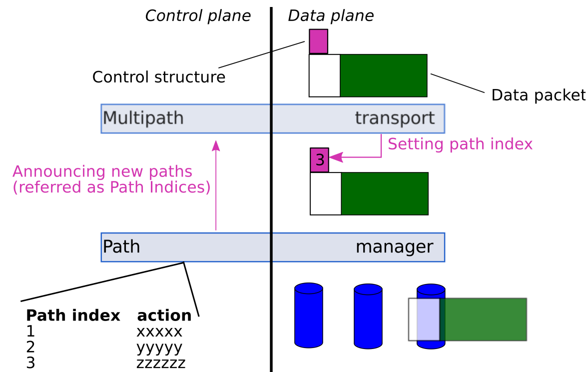

Figure 4.11: Extended Path Manager with per-packet actions

To reflect the need for a per-packet action, the PM mapping table (an example of which is given in Figure 4.2) needs to be extended with an action field. We show in Table 4.3 an example mapping table for a Path Manager based on writing a value into a field of the packets. Additionnally, while the built-in Path Manager is active only in the control plane, the extended Path Manager (with per-packet action ability) needs also be active in the data plane, as shown in Figure 4.11.

The Path Manager being now extended with an *action* field, we can illustrate the modularity of the architecture by defining other Path Managers. Path Managers are not attached to a particular layer. The only requirement for them is to be located below the Multipath Transport (MT) in the networking stack, so that outgoing packets can be marked by the MT before being handled by the Path Manager. The actual layer in which the Path Manager is located determines the scope of the paths that are managed. For example a link layer Path Manager handles links directly connected to the host. On the other hand, a network layer Path Manager could influence part or all of the end-to-end path. We provide a few examples of alternative Path Managers in the followings subsections (of them, only the Shim6-based Path Manager has been implemented as of this writing).

4.4.1 Next-hop selection

If a host has several interfaces, or one interface with several gateways on the same link, it can happen that multiple routes are available to reach the same destination. Currently, such cases usually result in the host using one default route, and keep

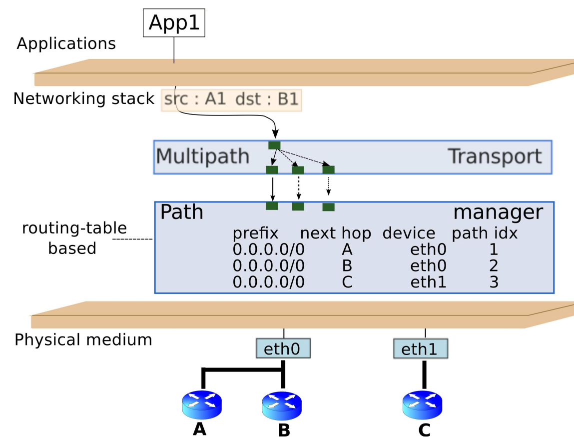

Figure 4.12: Path management with next-hop selection

the other ones for backup connectivity. Solutions exist (such as the *iproute2* package in Linux) to allow the end-host to perform load balancing on a per-transport flow basis, but per-packet load balancing is not recommended due to the classical reordering problem in TCP. The Path Manager abstraction would allow announcing the MT that multiple paths are available, while hiding the fact that those paths have actually multiple next-hops. The Path Manager based on next-hop selection is shown in Figure 4.12. Note that it is possible, in Figure 4.12, that the `eth1` interface uses its own address. This will simply be reflected in the source address of the third subsocket (that is, the subsocket with path index 3), because the PM can tell what addresses to use for a particular subflow (see Figure 4.2).

4.4.2 Shim6-based Path Manager

In the previous chapters, we have presented our work on the shim6 protocol. In fact, shim6 can very well provide a path management service to MPTCP. As currently defined [NB09], shim6 is a sub-layer of the IPv6 layer, and is completely invisible to the upper layers. It is able to detect failures on a path, identified by an address pair, and switch a flow to another path, while ensuring transport layer survivability of the connections. Such survivability is obtained by rewriting the address fields of a packet when the address seen by TCP is not the one that Shim6 wants to use (because it is known to have failed for example). Shim6 can be used almost as is for playing the role of a Path Manager. In that case, its failure detection capability is not needed anymore, since the Multipath Transport layer sees failures on a link as an infinite level of congestion, and it is assumed to be able to react accordingly. Shim6 contains all the necessary features to act as a Path Manager: It discovers the available paths by exchanging its address set with its peer. It is able to direct packets to any of the available paths by rewriting addresses. For experimentation, we have modified our LinShim6 implementation of shim6 to enable its use as a Path Manager. It can be downloaded from

https://scm.info.ucl.ac.be/trac/mptcp/wiki/mptcp_shim6.

We expect that the shim6 based Path Manager could be useful especially when a user wants to use shim6 anyway (i.e. to provide a path management service to MPTCP, *and* a regular shim6 service to other transport protocols). In other cases, the MPTCP built-in Path Manager remains the best choice, as it supports IPv6 together with IPv4, even in the same communication. A more detailed comparison between MPTCP and shim6 is given in the conclusion of this thesis.

4.4.3 Link aggregation

Link aggregation provides at layer 2 a function similar to the layer 3 next-hop selection. Here, only one route is seen in layer 3, but several physical links exist, although they appear as one logical link from layer 3 upward. Again, the problem of TCP reordering currently makes it necessary to ensure that any transport flow is always carried over the same link. Adding the Path Manager interface to a link aggregation mechanism would allow bringing knowledge of the multiple available links to the MT (Multipath Transport), and so perform packet based load balancing. This kind of Path Managers can reveal to be very useful in some data-centre configurations, where several Gigabit interfaces use link aggregation. MPTCP could be easily made aware of the multiple physical links, in order to use them to their full capacity.

4.4.4 Remotely controlled Path Managers

All the above end-host mechanisms have an equivalent in the network. Some of them are even more frequently deployed in networks than in end-hosts. For example, possible "network-based Path Managers" are ECMP [TH00], Multipath routing [MFB$^+$11] or proxy-shim6 [Bag08]. It would thus be interesting to remotely control them. Whereas host-based Path Managers read the Path Index in the control structure associated with a packet, network-based Path Managers will read the Path Index in the packet itself. Whereas host-based Path Managers send notifications about path properties to registered entities inside the local system, network-based Path-Managers send their notifications to registered hosts inside the network. This introduces the need for a means to perform the announcements from the network to the hosts. This mean could be ICMP, DHCP or another protocol. Alternatively, a static version of remotely controlled Path Managers would imply local configuration in the hosts, rather than using a network protocol.

Remotely controlled Path Managers include a host part and a network part. The host part is actually a special kind of host-based Path Manager, with the same interface as any other one. But the mechanism embedded in that Path Manager only consists of reading path information from either a configuration file or some network protocol and forwarding that information in the form of Path Indices to the MT. In the data plane, the local Path Manager writes the Path Index into some field of the packet (for example DSCP in IPv4 or the flow label in IPv6). The remote

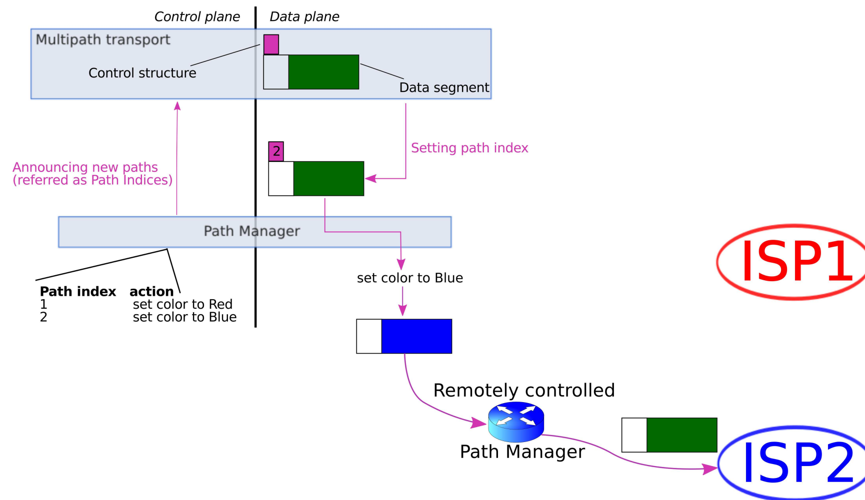

Figure 4.13: Remotely controlled Path Manager

Path Manager then reads the Path Index of incoming packets, and translates that number into an action, in exactly the same way as local Path Managers. Figure 4.13 illustrates the operation of a remotely controlled Path Manager. Inside the host, the Path Index makes the Path Manager set a colour for the packet. That is, some field holds a Path Selector value (the colour) which is used by the remote Path Manager, and can differ from the one used inside the host. The packet remains coloured until it reaches the remote Path Manager. Then the colour is removed and translated into an action (in Figure 4.13, direct packet to ISP2).

Related work: Another way to remotely control a Path Manager, using DHCP, has been proposed in [WR11]. The idea is to use only the MPTCP built-in Path Manager (using multiple IP addresses), and configure DHCP to announce one address per path. A small extension to DHCP allows exchanging information about the multiple paths between the DHCP server and client. By combining the protocol modification from [WR11] with the above Path Manager (using colours) it would be possible to increase its benefits by widening its applications and removing the requirement for several IP addresses. For example, the DHCP message `mp-proxy-avail` defined in [WR11] can be used to allow the DHCP server to announce how many paths the remote Path Manager supports (hiding the precise nature of the PM, e.g. proxy-shim6, multiple next-hops or another one). The second message from [WR11], `mp-range`, could be used to ask what Path Selector to use for a particular path index (currently it is used to ask what *address* to use). Note that the address is a particular case of path selector. Other path selectors could be ports or special values to be written in a particular field of packets. By generalizing that way, we could benefit from multipath even when one public address only is available, and no NAT is in place (hence no option to use multiple private

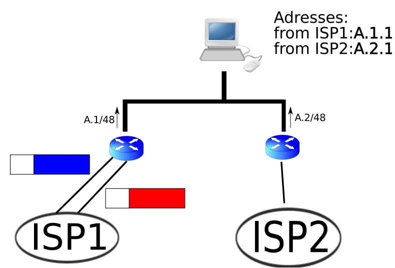

Figure 4.14: Example instantiation of cascaded Path Managers

addresses).

4.4.5 Combining Path Managers

The previous subsection presented several examples of Path Managers, each with different capabilities and scopes. The Path Manager abstraction enables different path selection mechanisms, while presenting a stable interface to the Multipath Transport. We note also that several path selection mechanisms may coexist in the network. It would thus be useful to benefit from all of them, in order to optimise the path offer for the MT. We explore here how future versions of MPTCP could support the simultaneous use of several Path Managers. A detailed evaluation of implementation details is left for future work, however.

When several Path Managers are used simultaneously, we propose that each one be given a depth attribute. The depth attribute reflects the layering requirement of each PM. Outgoing packets first flow through the PM of depth 0, then the PM of depth 1 and so on. We must also adapt the PM interface to support cascaded PMs. Two new rules are introduced:

- The MT can only listen to the Path Manager of depth 0
- A Path Manager of depth i listens to events from a Path Manager of depth i+1.

To illustrate a useful case of cascaded Path Managers, we use the scenario described in Figure 4.14. In that figure, a host is located in an IPv6 network that is dual-homed. It receives one address from ISP1 and another one from ISP2. We assume that it uses the built-in MPTCP PM (using the ADD_ADDRESS TCP option) to manage its two addresses. But ISP1 has itself two major upstream providers, and offers its clients the option of choosing what upstream provider is selected by routers in ISP1, based on the flow label field of the IPv6 header.

To map this situation to our architecture, we use two Path Managers. One is the built-in MPTCP Path Manager, with depth 0 (which we expect will be enabled by default in any setup). The built-in MPTCP PM sees two local addresses, giving the possibility to choose between ISP1 and ISP2. The other one is a remotely controlled Path Manager (depth 1), the host part of which writes the flow-label

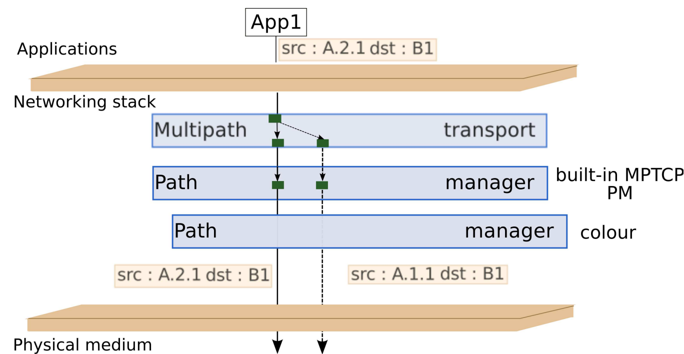

Figure 4.15: Cascaded Path Managers, only the built-in PM is active

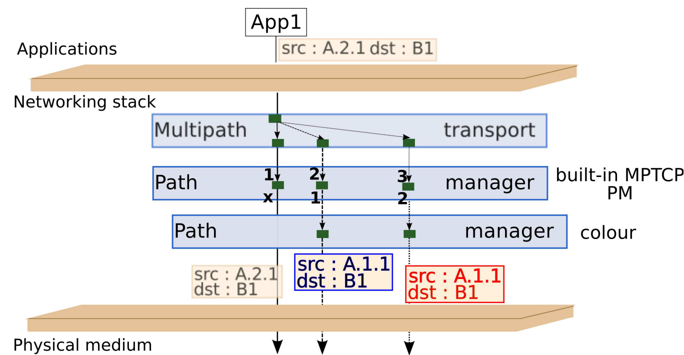

Figure 4.16: Cascaded Path Managers, final setup

field of IPv6 packets. The network part, located on the routers of ISP1, selects the outgoing interface based on the flow-label field, which actually holds a Path Index. Blue packets (e.g. flow label=1) are directed to one upstream provider, while red packets (e.g. flow label=2) are directed to the other one.

The two Path Managers are layered as shown in Figure 4.15. The built-in PM has Depth 0, while the Colouring Path Manager (that remotely controls ISP1) has Depth 1.

Let's suppose that an application App1 wants to start a communication with address pair $< A.2.1, B1 >$. The built-in PM detects the new flow and exchanges address information with the peer. The colouring PM remains inactive, because it knows only one path for source address $A.2.1$ (thus ISP2). As soon as MPTCP has found that two paths can be used to reach B1 (through ISP1 or ISP2), it tells it to the MT, which immediately starts balancing packets over two paths (see Figure 4.15). But now the Colouring Path Manager perceives a new flow for address pair $< A.1.1, B1 >$, since MPTCP has opened a new subflow with this address pair. The administrator has written a configuration file that tells that two paths are available through flow label tagging if source address prefix $A.1/48$ is used. The Colouring Path Manager announces the two paths upwards. Only the PM with

Source Address Prefix	path index	Action (Write x in flowlabel)
A.1/48	1	1
A.1/48	2	2

Table 4.4: Mapping table for the colouring Path Manager (depth 1)

token	path idx	Endpoint id	Action (rewrite path idx to x)
token_1	1	$< A.1.2, B1, 0, pB1 >$	0
token_1	2	$< A.1.1, B1, 0, pB1 >$	1
token_1	3	$< A.1.1, B1, 0, pB1 >$	2

Table 4.5: Example mapping table for the built-in PM (depth 0)

depth 0 listens to that notification, and learns that it can itself behave as an MT for the Colouring Path Manager.

We emphasise that in the case of Cascaded Path Managers, each path Manager on the path of a packet may need to rewrite the Path Index located in the control structure of the packet, in order to control the following Path Manager. In our example, the Colouring Path Manager has announced no path index for address pair $< A.2.1, B1 >$, and path index 1, 2 for address pair $< A.1.1, B1 >$. From now on, the built-in PM knows that it can not only apply its usual mechanism for its own path selection, but also set a path index 1 or 2 in case it uses $< A1, B1 >$ as addresses for its packets. In order to allow upper layers to make use of this additional path, the built-in PM tells upper layers that one new path is available. The final setup is shown in Figure 4.16 and tables 4.4, 4.5. The Multipath Transport is now able to use all paths without having any idea of the mechanisms (combined in this case) that are used to select the paths.

4.5 Configuring the OS for MPTCP

Previous sections concentrated on our MPTCP implementation. In this section, we gather guidelines that help getting the full potential from MPCTP through appropriate system configuration. By providing configuration guidelines, we also shed light on how difficult it is for a user to get benefits from MPTCP.

4.5.1 Source address based routing

As already pointed out by [BS10], the default behaviour of most operating systems is not appropriate for the use of multiple interfaces. Most operating systems are typically configured to use at most one IP address at a time. It is more and more common to maintain several active links (e.g. using the wired interface as main link, but maintaining a ready-to-use wireless link in the background, to facilitate fallback when the wired link fails). But MPTCP is not about that. MPTCP is about *simultaneously* using several interfaces (when available). It is expected that

one of the mostly used MPTCP configurations will be through two or more NICs, each being assigned a different address. Another possible configuration would be to assign several IP addresses to the same interface, in which case the path diverges later in the network, based on the particular source address that is used in the packet[9].

Usually an operating system has a single default route, with a single source IP address. If the host has several IP addresses and we want to do Multipath TCP, it is necessary to configure source address based routing. This means that based on the source address selected by Multipath TCP, the routing engine consults a different routing table. Each of these routing tables defines a default route to the Internet. This is different from defining several default routes in the same routing table (which is also supported in Linux), because in that case only the first one is used. Any additional default route is considered as a fallback route, used only in case the main one fails.

For instance, consider a host with two interfaces, I1 and I2, both connected to the public Internet and assigned addresses resp. A1 and A2. Such a host needs 3 routing tables. One of them is the classical routing table, present in all systems. This default routing table is used to find a route based on the destination address only, when a segment is issued with the undetermined source address. The undetermined source address is typically used by applications that initiate a TCP `connect()` system call, specifying the destination address but letting the system choose the source address. In that case, after the default routing table has been consulted, an address is assigned to the socket by the system (For IPv6, the RFC3484 source address selection algorithm [Dra03] is applied. For IPv4 no such algorithm is defined and the address selection is operating system dependent).

The additional routing tables are used when the source address is specified. If the source address has no impact on the route that should be chosen, then the default routing table is sufficient. But this is a particular case (e.g., a host connected to one network only, but using two addresses to exploit ECMP paths later in the network). In most cases, a source address is attached to a specific interface, or at least a specific gateway. Both of those cases require defining a separate routing table, one per *(gateway, outgoing interface)* pair.

To select the proper routing table based on the source address, an additional indirection level must be configured. It is called "policy routing" in Linux and is illustrated at the bottom of Figure 4.17.

If only the default routing table were used, only the first default route would be used, regardless of the source address. For example, a packet with source address A2 would leave the host through interface I1, which is incorrect.

[9] Note that this is *source address based routing*, which is different from *source routing*, where the end-host encodes in the packet header the addresses of the intermediate hops that should be used for forwarding.

```
+------------------------------------------------------+
|                    Default Table                     |
+------------------------------------------------------+
| Dst: 0.0.0.0/0  Via: Gateway-IP1 Dev: I1             |
| Dst: 0.0.0.0/0  Via: Gateway-IP2 Dev: I2             |
| Dst: Gateway1-Subnet Dev: I1 Src: A1  Scope: Link    |
| Dst: Gateway2-Subnet Dev: I2 Src: A2  Scope: Link    |
+------------------------------------------------------+

+------------------------------------------------------+
|                       Table 1                        |
+------------------------------------------------------+
| Dst: 0.0.0.0/0  Via: Gateway-IP1 Dev: I1             |
| Dst: Gateway1-Subnet Dev: I1 Src: A1 Scope: Link     |
+------------------------------------------------------+

+------------------------------------------------------+
|                       Table 2                        |
+------------------------------------------------------+
| Dst: 0.0.0.0/0  Via: Gateway-IP2 Dev: I2             |
| Dst: Gateway2-Subnet Dev: I2 Src: A2 Scope: Link     |
+------------------------------------------------------+

+------------------------------------------------------+
|                  Policy Table                        |
+------------------------------------------------------+
|   If src == A1 , Table 1                             |
|   If src == A2 , Table 2                             |
+------------------------------------------------------+
```

Figure 4.17: Example Routing table configuration for Multipath TCP with two interfaces

4.5.2 Buffer configuration

[FRH+11, Section 5.3] describes in details the new, higher buffer requirements of MPTCP. Sections 4.3.5 and 4.3.6 describe how the MPTCP buffers can be tuned dynamically. However, it is important to note that even the best tuning is capped by a maximum configured at the system level.

When using Multipath TCP, the maximum receive and send buffer should be configured to a higher value than for regular TCP. There is no universal guideline on what value is best there. Instead, the most appropriate action for an administrator is probably to roughly estimate the maximum bandwidth and delay that can be observed on a particular connectivity setup, and apply the equation from [FRH+11, Section 5.3], to find a reasonable tradeoff. This exercise could lead an administrator to decide to disable MPTCP on some interfaces, because it allows consuming less memory while still achieving reasonable performance.

4.6 Future work

Although Linux MPTCP is already an operational and efficient prototype, there is still much space for improvements. In this section we assemble a list of future improvements that would make the MPTCP implementation even better.

- Today's host processors have more and more CPU cores. Given Multipath TCP tries to exploit another form of parallelism, there is a challenge in finding how those can work together optimally. An important question is how to work with hardware that behaves intelligently with TCP (e.g. flow to core affinity). This problem is discussed in more detail in [Wat10]. Our current design is optimised for handling per-core connections (a whole connection with all of its subflows runnning on the same core). But for this to work with best efficiency, all subflows should be forced on the same core (to benefit from cache sharing), which is not the case yet. Another completely different design that could be evaluated is to handle per-core subflows, where each subflow is forced on a different core. We expect less benefit from this second option however, as the data must be gathered in the same memory pool in the end (the user-context buffer), which could involve cross-caches copies.

- An evaluation of Linux MPTCP exists (see Chapter 5) and some optimisations have been implemented already, but many optimisations are still possible and should be evaluated. Examples of them include MPTCP fast path (that is, a translation of the existing TCP fast path to MPTCP) or DMA support.

- Currently, support for TCP Segmentation Offload remains a challenge because it modifies the Maximum Segment Size. Linux MPTCP currently works with a single MSS shared by all subflows (see Section 4.3.6). Adding

TSO support to MPTCP is certainly possible, but requires further work. Also, support for Large Receive Offload has not been investigated yet.

- There are ongoing discussions in the IETF on heuristics that would be used to decide when to start new subflows. Those discussions are summarised in the next paragraph, but none of the proposed heuristics has been evaluated yet.

Heuristics for subflow management: An interesting discussion on future possible improvements happened on the IETF MPTCP mailing list [10]. We summarise it here, as it can provide valuable input for future implementation work.

MPTCP is not useful for very short flows, so three questions appear:

- How long is a "too short flow"
- How to predict that a flow will be short ?
- When to decide to add/remove subflows ?

To answer the third question, it has been proposed to use hints from the application. On the other hand the experience shows that socket options are quite often poorly or not used, which motivates the parallel use of a good default heuristic. This default heuristic may be influenced in particular by the particular set of options that are enabled for MPTCP (e.g. an administrator can decide that some security mechanisms for subflow initiation are not needed in his environment, and disable them, which would change the cost of establishing new subflows). The following elements have been proposed to feed the heuristic:

- Check the size of the write operations from the applications. Initiate a new subflow if the write size exceeds some threshold. This information can be taken only as a hint because applications could send big chunks of data split in many small writes. A particular case of checking the size of write operations is when the application uses the `sendfile()` system call. In that situation MPTCP can know precisely how many bytes will be transferred.

- Check if the flow is network limited or application limited. It is network limited if the sender is paced mainly by its congestion window. It is application limited when either the receiver or the sender application is limiting the transmission rate, by respectively using the `read()` and `write()` system calls rarely and/or with small amounts of data. A possible heuristic would be to initiate a new subflow only if a flow is network limited.

- It may be useful to establish new subflows even for application-limited communications, to provide failure survivability. A way to do that would be to

[10] MPTCP mailing list archive, *mptcp-multiaddressed: How long before multipath starts ?*, from January 31st, 2011 to February 2nd, 2011

initiate a new subflow (if not done before by another trigger) after some time has elapsed, regardless of whether the communication is network or application limited.

- Wait until slow start is done before to establish a new subflow. Measurements with Linux MPTCP (see Chapter 5) suggest that the end of the slow start could be a reasonable hint for determining when it is worth starting a new subflow (without increasing the overall completion time). More analysis is needed in that area, however. Also, this should be taken as a hint only if the slow start is actually progressing (otherwise a stalled subflow could prevent the establishment of another one, precisely when a new one would be useful).

- Use information from the application-layer protocol. Some of them (e.g. HTTP) carry flow length information in their headers, which can be used to decide how many subflows are useful.

- Allow the administrator to configure subflow policies on a per-port basis. The host stack could learn as well for what ports MPTCP turns out to be useful.

- Check the underlying medium of each potential subflow. For example, if the initial subflow is started over 3G, and WiFi is available, it probably makes sense to immediately negotiate an additional subflow over WiFi.

It is not only useful to determine when to start new subflows, one should also sometimes decide to abandon some of its subflows. An MPTCP implementation should be able to determine when removing a subflow would increase the aggregate bandwidth. This can happen, for example, when the subflow has a significantly higher delay compared to other subflows, and the maximum buffer size allowed by the administrator has been reached (Linux MPTCP currently has no such heuristic yet).

4.7 Conclusions

In this chapter, we have presented our implementation of Multipath TCP in the Linux kernel. That implementation is the first and most complete available in an operating system. We explained our main design choices, especially the need for buffering data at the connection level in order for the scheduler to make decisions about packet path attributions as close as possible to the actual sending of the packet. We also explained that doing this does unfortunately not come without drawbacks, and that all subflows must use the same MSS.

We emphasised that the size of the buffers required to actually get benefits from MPTCP is higher than the buffer size required by regular TCP, especially when the available paths have very different delays. Such differences could lead an administrator to disable some of the slowest paths.

In section 4.4, we explained in detail our division of MPTCP in two main components, namely Multipath Transport and Path Management. We showed through various examples the benefits that can be obtained from decoupling path management, and illustrated how Path Managers could be combined in order to benefit from multipath at several layers during a single MPTCP connection.

Finally, we described the areas of improvements that could be given to Multipath TCP, to make it even more competitive in today's Internet and on multicore hosts.

Chapter 5

MPTCP evaluation

5.1 Introduction

In this chapter we use our Linux MPTCP implementation to evaluate the protocol behaviour in real-world scenarios. We first look at the performance of our implementation, in Section 5.2. We then show that the coupled congestion control correctly achieves the fairness goals presented in [WRGH11], allowing MPTCP to efficiently use several paths simultaneously, while still being fair to competing TCP flows. We conclude with a presentation of a concrete, promising, but initially not expected application for MPTCP: data-centres.

5.2 MPTCP performance

To evaluate the performance of our implementation, we performed lab measurements in the HEN testbed at University College London (`http://hen.cs.ucl.ac.uk/`). Our testbed is depicted in figure 5.1.

It is composed of three workstations. Two of them act as source and destination while the third one serves as router. The source and destination are equipped with AMD Opteron™ Processor 248 single-core 2.2 GHz CPUs, 2GB RAM and two Intel® 82546GB Gigabit Ethernet controllers. The links and the router are configured to ensure that the router does not cross-route traffic, i.e. the packets

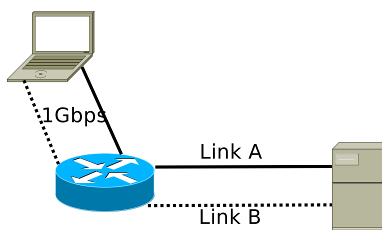

Figure 5.1: Testbed used for the performance evaluation

that arrive from the solid-shaped link in figure 5.1 are always forwarded over the solid-shaped link to reach the destination. This implies that the network has two completely disjoint paths between the source and the destination. We configure the bandwidth on `Link A` and `Link B` by changing their Ethernet configuration.

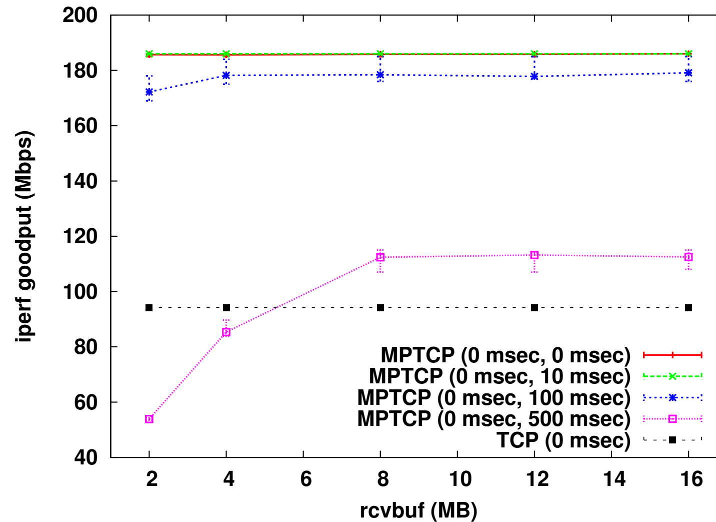

Figure 5.2: Impact of the maximum receive buffer size

As explained in section 4.3.5, one performance issue that affects Multipath TCP is that it may require large receive buffers when subflows have different delays. To evaluate this impact, we configured `Link A` and `Link B` with a bandwidth of 100 Mbps. Figure 5.2 shows the impact of the maximum receive buffer on the performance achieved by MPTCP with different delays. For these measurements, we use two MPTCP subflows. One is routed over `Link A` while the second is routed over `Link B`. The router is configured to insert an additional delay of 0, 10, 100 or 500 milliseconds on `Link B`. No delay is inserted on `Link A`. This allows us to consider an asymmetric scenario where the two subflows are routed over very different paths. Such asymmetric delays force the MPTCP receiver to store many packets in its receive buffer to be able to deliver all the received data in sequence. As a reference, we show in figure 5.2 the throughput achieved by `iperf` with a regular TCP connection over `Link A`. When the two subflows have the same delay, they are able to saturate the two 100 Mbps links with a receive buffer of 2 MBytes or more. When the delay difference is of only 10 millisecond, the goodput measured by `iperf` is not affected. With a delay difference of 100 milliseconds between the two subflows, there is a small performance decrease. The performance is slightly better with 4 MBytes which is the default maximum receive buffer in the Linux kernel. When the delay difference reaches 500 milliseconds, an extreme case that would probably involve satellite links in the current Internet, the goodput achieved by Multipath TCP is much more affected. This is mainly be-

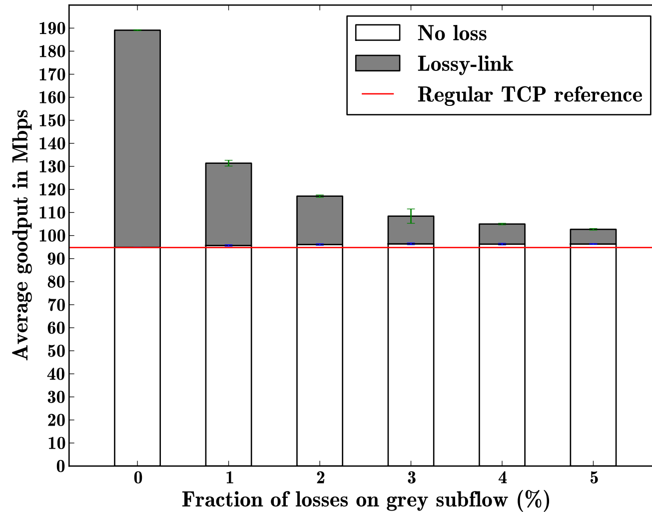

Figure 5.3: Impact of packet losses

cause the router drops packets and these losses cause timeout expirations and force MPTCP to slowly increase its congestion window due to the large round-trip-times.

The losses are a second factor that affects the performance. To evaluate whether losses on one subflow can affect the performance of the other subflow due to head-of-line blocking in the receive buffer, we configured the router to drop a percentage of the packets on `Link B` but no packets on `Link A` and set the delays of these links to 0 milliseconds. Figure 5.3 shows the impact of the packet losses on the performance achieved by the MPTCP connection. The figure shows the two subflows that compose the connection, as stacked bars (10 measurements per bar, 95% confidence intervals). We obtain the contribution of each subflow to the overall goodput by considering the aggregated goodput as measured by the iperf tool, and weighting by the total number of data bytes effectively sent on each of the subflows, without taking retransmissions into account. This number of bytes is measured directly on the packet trace. The subflow shown in the lower bar passes through `Link A` while the subflow shown in the upper bar (in gray) passes through `Link B`. The goodput achieved by a regular TCP connection running the new-Reno congestion control algorithm is shown as a reference (94.8Mbps). When there are no losses, the MPTCP connection is able to saturate the two 100 Mbps links. As expected, the gray subflow that passes through `Link B` is affected by the packet losses and its goodput decreases with the fraction of packet losses. The goodput of the other subflow does not suffer from sharing transmissions with a lossy subflow. With a loss fraction of 5% on the grey subflow, we also remark that the aggregate goodput goes beyond 100Mbps.

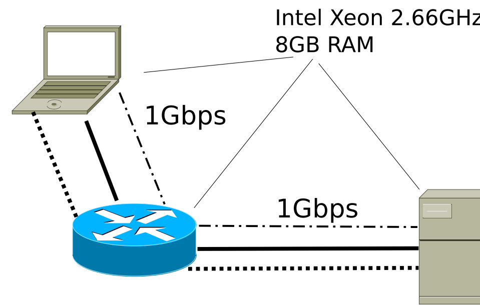

Figure 5.4: Testbed with 3 Gigabit links

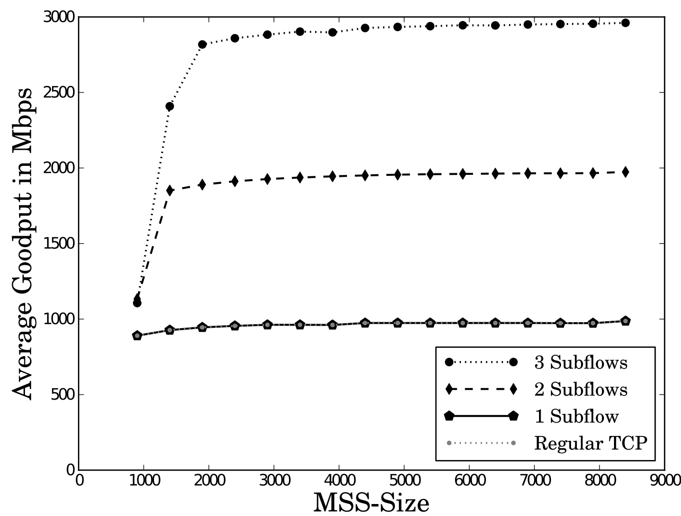

Figure 5.5: Impact of the MSS on the performance

This is better than the throughput achieved when the coupled congestion control is disabled[1], because in that case the higher amount of traffic sent on the lossy link causes timeouts, that in turn saturate the receive window and slow down the whole connection.

The last factor that we analyze is the impact of the Maximum Segment Size (MSS) on the achieved goodput. TCP implementors know that a large MSS enables higher goodput [Bor89] since it reduces the number of interrupts that need to be processed. To evaluate the impact of the MSS, we do not introduce delays nor losses on the router and use either one, two or three Gigabit Ethernet interfaces on the source and the destination (see Figure 5.4). Figure 5.5 shows the goodput in function of the MSS size. With a standard 1400 bytes MSS, MPTCP can fill one 1 Gbps link, and partly use a second or third one. It is able to saturate two Gigabit Ethernet links with an MSS of 3000 bytes, and three Gigabit Ethernet links with an MSS of 4500 bytes. Note that we do not use TSO (TCP Segmentation

[1] This case, not shown in the picture, corresponds to each subflow running new-Reno separately.

Histogram bar	BST	Segment aggregation	Shortcuts
Regular	Disabled	Disabled	Disabled
Tree	**Enabled**	**Enabled**	Disabled
Shortcuts	Disabled	Disabled	**Enabled**
Aggreg	Disabled	**Enabled**	**Enabled**

Table 5.1: Optimisations enabled for Figure 5.6

Offload) [Cur04] for these measurements. With TSO the segment size handled by the system could have grown virtually to 64KB and allowed it to reach the same goodput with a lower CPU utilisation. TSO support will be added later in our MPTCP implementation.

Finally, we have looked at the CPU consumption in the system in the data sink. Most TCP implementations support Van Jacobson's fast path processing [Jac93]. The receiver assumes that data is received in-order and TCP quickly places the data received in-sequence in its receive buffer, at the end of the in-order receive-queue (which the application can read). If the packet cannot be placed in the receive queue, it is then placed in the out-of-order queue, which is an ordered sequence of segments with holes. This queue is searched for an appropriate place starting at the end, because in most cases this queue is used when a packet has been lost and all subsequent packets arrive in sequence until the hole is filled (that is, the missing segment is retransmitted). In the rare case where a segment can be placed neither at the end of the receive queue nor at the end of the out-of-order queue, TCP scans the out-of-order queue to find the exact location of the received data.

With MPTCP the situation is completely different: while subflow sequence numbers are received in-order, data sequence numbers are often out-of-order, forcing receivers to scan the large out-of-order queue. An obvious fix is to use a binary search tree to reduce the out-of-order queue lookup time. This adds complexity to the code, and still takes logarithmic time to place a packet.

To obtain a simple, constant time receive algorithm we take into account the way packets are sent: when a subflow is ready to send data, a batch of segments with contiguous data sequence numbers are allocated by the connection and sent on this subflow. Each subflow then receives ordered segments at the data level as long as the batch size is large. To take advantage of this, we augment each subflow data structure with a pointer to the out-of-order queue position that is expected to receive the next segment to arrive on that subflow. If the pointer is wrong, we revert to scanning the whole out-of-order queue (the implementation details for the binary search tree and the shortcut pointers have been given in Section 4.3.5).

Figure 5.6 compares the CPU load (measured with the *mpstat* Linux utility) for the different receive algorithms[2]. The exact configuration used to generate each of the bars is shown in Table 5.1. TCP (with 2 and 8 connections) is used as a benchmark. The optimising algorithms are evaluated by considering a client

[2]This set of measurements has been performed by Costin Raiciu

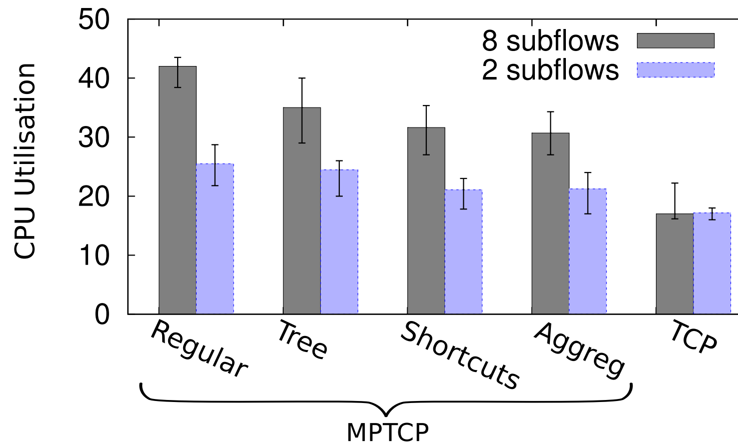

Figure 5.6: Effect of ofo receive algorithms on CPU load

directly connected to a server by using two 1 Gbps links. The client starts a long download and the CPU load of the receiver is measured. With more subflows the number of out-of-order segments increases. For clarity, the results are presented only with 2 subflows, a lower bound to utilise the links, and 8 subflows beyond which results are similar.

The shortcut pointers are particularly efficient as they work for 80% of the received segments. The *Tree* algorithm reduces CPU utilisation, but *Shortcuts* and its variant *Aggreg* help much more. When 8 subflows are used, CPU utilisation drops from 42% to 30%, and when 2 subflows are used it drops from 25% to 20%.

Regarding the repartition of the load between the software interrupts and the user context, we found that the majority of the processing was done in user context for the receiver. For example, with a 1400 bytes MSS and a single Gigabit Ethernet link, with 16 subflows, less than 1% of the receive-side MPTCP processing time was spent in software interrupt context. The bigger amount of processing for user context comes from backlog queue processing: if the meta-socket is locked by the user context, all segments received during that time are enqueued in the backlog queue. They are handled during the `release_sock()` operation, in user context (see Section 4.3.3). Van Jacobson prequeues (which we described in Section 4.3.1) are in use in these measurements, but the backlog queue plays a more significant role in this set of measurements.

The sender spends around 20% of the total MPTCP processing time in software interrupt context under the same conditions. The significantly larger amount of work performed in software interrupt context comes from the fact that the majority of the segments are sent when an incoming acknowledgement opens more space in the congestion or sending window. This event happens in interrupt context and requires (currently) running the scheduler for each segment.

5.3 MPTCP congestion control

The support for Coupled Congestion Control [RHW11, WRGH11] has been added by Christoph Paasch in our MPTCP implementation[3]. To test it, a specific topology has been used in the HEN testbed, as shown in Figure 5.7.

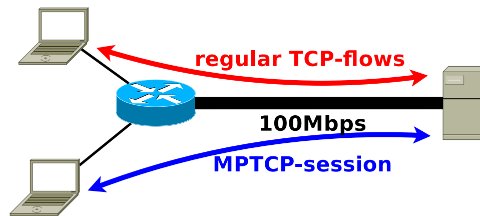

Figure 5.7: Congestion testbed

This topology is composed of four Linux workstations that use Intel® Xeon CPUs (2.66GHz, 8GB RAM) and Intel® 82571EB Gigabit Ethernet Controllers. The two workstations on the left are connected with a Gigabit link to a similar workstation that is configured as a router. The router is connected with a 100 Mbps link to a server. The upper workstation in figure 5.7 uses a standard TCP implementation while the bottom host uses our Multipath TCP implementation.

Detailed simulations performed by Wischik et al. in [WRGH11] show that the coupled congestion control fulfills its goals. In this section we illustrate that our Linux MPTCP with the Coupled Congestion Control achieves the desired effects, even if it is facing additional complexity due to fixed-point calculations compared to the user-space implementation used in [WRGH11]. In the congestion testbed shown in figure 5.7, the coupled congestion control should be fair to TCP. This means that an MPTCP connection should behave like a single TCP connection at the bottleneck. Furthermore, an MPTCP connection that uses several subflows should not slow down regular TCP connections. To measure the fairness of MPTCP, the bandwidth measurement software `iperf`[4] has been used to establish sessions between the hosts on the left and the server on the right part of the figure [5]. Different numbers of regular TCP connections are established from the upper host, as well as different numbers of MPTCP subflows used by the bottom host. Iperf sessions were run for a duration of 10 minutes, to allow the TCP-fairness over the bottleneck link to converge [LLS07]. Each measurement is run five times and the average throughput is reported, as well as the 95% confidence intervals.

Thanks to the flexibility of the Linux congestion control implementation, tests have been performed by using the standard NewReno congestion control scheme with regular TCP and either NewReno or the coupled congestion control scheme with MPTCP.

[3]The evaluation presented in this subsection is the result of a joint work presented in [BPB11b].
[4]http://sourceforge.net/projects/iperf/
[5]This set of measurements has been conducted by Christoph Paasch.

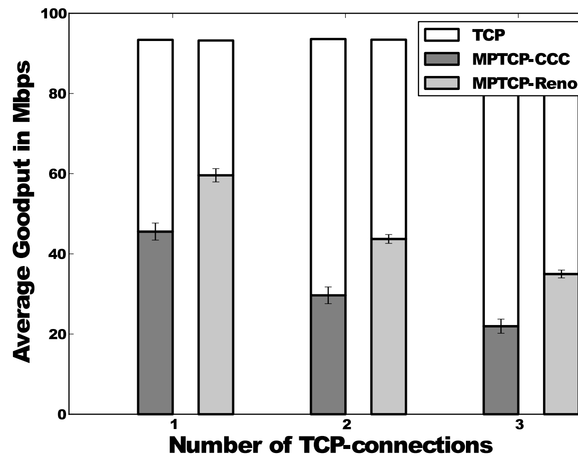

Figure 5.8: Multipath TCP with coupled congestion control behaves like a single TCP connection over a shared bottleneck with respect to regular TCP.

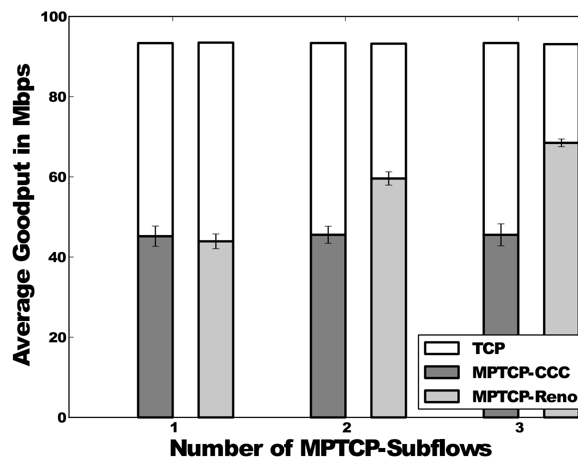

Figure 5.9: With coupled congestion control on the subflows, Multipath TCP is not unfair to TCP when increasing the number of subflows.

The measurements show that MPTCP with the coupled congestion control is fair to regular TCP. Figures 5.8 and 5.9 present the amount of the available bottleneck link capacity (100Mbps) taken by TCP and MPTCP. If several TCP flows coexist (Figure 5.8), they are summed and presented in the same stack bar. When an MPTCP connection with two subflows shares a bottleneck link with a regular TCP connection, the coupled congestion control behaves as if the MPTCP session was just one single TCP connection. However, when Reno congestion control is used on the subflows, MPTCP gets more bandwidth because in that case two TCP subflows are really competing against one regular TCP flow (Figure 5.8).

The second scenario that is evaluated with the coupled congestion control is the impact of the number of MPTCP subflows on the throughput achieved by a single TCP connection over the shared bottleneck. Measurements are performed by using one regular TCP connection running the Reno congestion control scheme and one MPTCP connection using one, two or three subflows. The MPTCP connection uses either the Reno congestion control scheme or the coupled congestion control scheme. Figure 5.9 shows first that when there is a single MPTCP subflow, both Reno and the coupled congestion control scheme are fair as there is no difference between the regular TCP and the MPTCP connection. When there are two subflows in the MPTCP connection, figure 5.9 shows that Reno favours the MPTCP connection over the regular single-flow TCP connection. The unfairness of the Reno congestion control scheme is even more important when the MPTCP connection is composed of three subflows. In contrast, the measurements indicate that the coupled congestion control provides the same fairness when the MPTCP connection is composed of one, two or three subflows.

5.4 MPTCP in the data-centre

In a collaboration with University College London, we have studied the benefits of Multipath TCP in a data-centre environment. We combined simulation results (from UCLondon) with experimental results, using our Linux MPTCP implementation. The results of this work have been published in [RPB+10, RBP+11].

5.4.1 Context

One issue that could slow down the deployment of MPTCP is the requirement to support it at both ends of the communication. Data-centre environments do not suffer from this, because the sender and the receiver are under the same administrative control. Moreover, MPTCP can bring major benefits to such environments as they are already built with multiple paths for both failure tolerance and load sharing.

Today, data-centres can involve hundreds of thousands of servers [Sha08]. Routing traffic between them requires the use of a hierarchical topology of switches and/or routers. To understand this, consider that we want to build a data-centre using 1 Gbps NICs and at most 10 Gbps switch ports. Those values are realis-

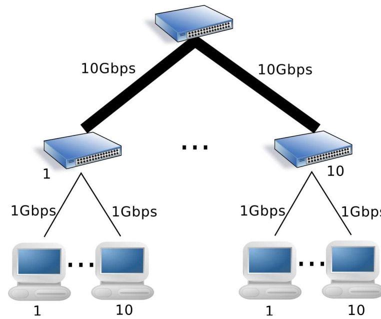

Figure 5.10: Simple data-centre topology

tic with today's hardware, as data-centres often run distributed applications that need fast transfers to and from arbitrary servers in the data-centre. A data-centre topology is often characterized by its bisection bandwidth, defined by [HP06] as follows:*"For a given topology, bisection bandwidth, $BW_{Bisection}$, is calculated by dividing the network into two roughly equal parts - each with half the nodes - and summing the bandwidth of the links crossing the imaginary dividing line. For nonsymmetric topologies, bisection bandwidth is the smallest of all pairs of equal-sized divisions of the network."*. If the worst-case bisection of the network allows one half to communicate at interface speed with the other half, the network is said to have *full bisection bandwidth*. In that case the bisection bandwidth is $iface_speed \times N/2$, where N is the number of hosts. The network shown in Figure 5.10 has full bisection bandwidth (assuming the core switch is powerful enough): N is 100 (note the dots between the two fat links), and the bisection bandwidth is $iface_speed \times N/2 = 50Gbps$.

Full bisection bandwidth topologies are not the norm however, as they require using prohibitively expensive devices for a capacity that is not even necessarily used. For that reason the core links are usually designed to achieve less than full bisection bandwidth. The extent to which such topologies depart from full bisection bandwidth is given by the *oversubcription ratio*: $\frac{BW_{Bisection}}{iface_speed \times N/2}$. For example, we can increase the number of hosts to 200 in figure 5.10 by just connecting ten more hosts per lower-level switch. In that case the bisection bandwidth is still $50Gbps$ but $N/2$ is 100, giving an oversubscription ratio of 1:2. The lower-level switches indeed become potential bottlenecks.

The network of Figure 5.10 is a kind of fat-tree [Lei85] (that is, the links become "fatter" when moving away from the leaves to the root). As the network switches needed to handle the fat links are significantly more expensive than the smaller ones, [AFLV08] has proposed to instantiate a Clos topology [Clo53][6] for data-centres. An example is shown in figure 5.11, where 4-ports switches are used, and only Gigabit Ethernet links are used (meaning that all layers of the topology

[6] Clos proposed, in 1953, a topology for telephone switching neworks, when also facing high price differences between low and high bandwidth switches.

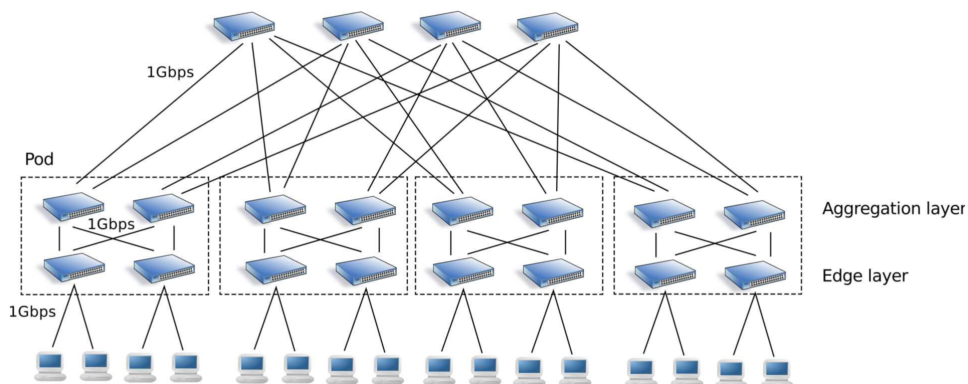

Figure 5.11: Clos-based data-centre topology (Al-Fares et al.)

can use commodity switches). Ironically, Al Fares et al. chose to call their architecture proposal "Fat-Tree". We will not follow that terminology and will instead refer to this structure as a Clos topology, given the main property of the topology is precisely that *there is no fat link* in the proposal from [AFLV08] described in Figure 5.11. An example of *real* Fat Tree is instead Figure 5.10 [Lei85]. The particular small example of Figure 5.11 is obviously not interesting as there are more switches than hosts. However, it scales much better than the hierarchical fat-tree approach and is cheaper. For example a Clos-based data-centre that uses 48-ports switches can support up to 27648 hosts according to [AFLV08], and requires 2880 GigE 48-ports switches. This topology is *rearrangeably non-blocking* [Ben65], that is, it is always possible to find a combination of paths (as this topology is multipath) that allows all hosts to communicate with another host at full interface speed. However, [AFLV08] recognizes that TCP does not allow to *rearrange* the paths easily (otherwise reordering would cause a performance drop). In a separate paper [AFRR+10], the same authors propose the use of a centralized scheduler to rearrange only the big flows (as it would be too costly to rearrange *all* flows centrally). Unfortunately this centralized scheduler suffers from slow reaction time and may need to be run very frequently under some communication patterns [RBP+11]. This is exactly where MPTCP can bring great benefits. MPTCP just uses all of the paths, hence rearranging "automatically", at the time-scale of a Round-Trip-Time. Simulation results have been published in [RBP+11] and show that MPTCP indeed performs much better than regular TCP in such topologies.

Similar to the Clos topology proposed by [AFLV08], VL2 (Virtual Layer 2) is also a Clos topology but it uses fewer faster links near the roots [GHJ+09]. It has also multiple paths and tries to efficiently use them by implementing Valiant Load Balancing [KLS04, ZSM04] and ECMP inside the end-hosts. As ECMP and VLB are blind to actual traffic conditions in the network, there is also much benefit to expect from running MPTCP in such networks, as shown in simulations in [RBP+11].

Finally, another recent data-centre proposal is BCube [GLL+09]. Here the ap-

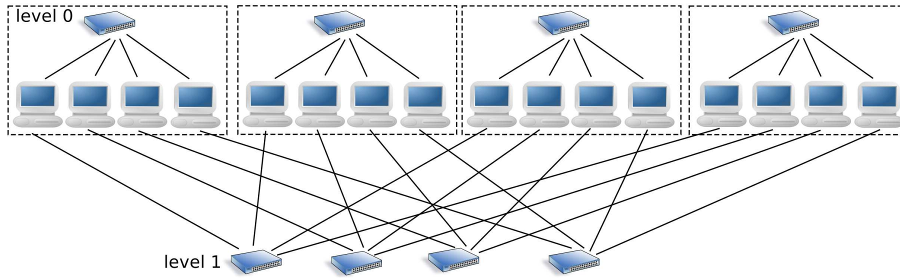

Figure 5.12: BCube data-centre topology ($k = 1, n = 4$)

proach is completely different as the hosts themselves are used to forward traffic. The topology is arranged recursively (see Figure 5.12). The simplest element (level 0) is a set of servers directly connected through a rudimentary switch. Additional server ports can be used to connect to other recursive levels as shown for level 1 in Figure 5.12. In general, a $Bcube_k$ has $k + 1$ recursive levels (0 through k) that use simple n-ports switches to connect together n $Bcube_{k-1}$. Like the other topologies, the Bcube leverages multiple paths between each destination pair. Although a specific routing protocol is proposed to take benefit from this multipath topology, Bcube is forced to assign flows, not packets to any particular path. Again, MPTCP brings the possibility to get finer-grained path allocation. Moreover, given that this topology uses several interfaces per host, MPTCP can even allow bandwidths that are *higher* than the interface rate, a feature that was not possible before. [RBP$^+$11] also presents simulation results that emphasise the benefits of MPTCP in a BCube environment.

5.4.2 Experimental evaluation

To complement the simulation results presented in [RBP$^+$11], we have used our implementation to evaluate the MPTCP behaviour in real-world conditions.

Completion time for short flows: We conducted a first experiment using the HEN testbed presented in Figure 5.1. The question we wanted to answer was: Should MPTCP be enabled for all TCP connections in a data-centre ? We measured the time to setup a connection and transfer a short file. In this measurement the file transfer is initiated by the client immediately after the three-way handshake. TCP can only use one interface; MPTCP can also use the second, but only after the first subflow has negotiated the use of MPTCP and the second subflow has been established. Figure 5.13 shows that TCP is faster for files of less than about 10 packets, but much slower thereafter. This comes from the overhead of establishing an additional subflow, and the fact that this subflow cannot be established before the initial three-way handshake has completed. To avoid penalizing short flows, the code just needs to wait two RTTs after data transmission starts (or until the

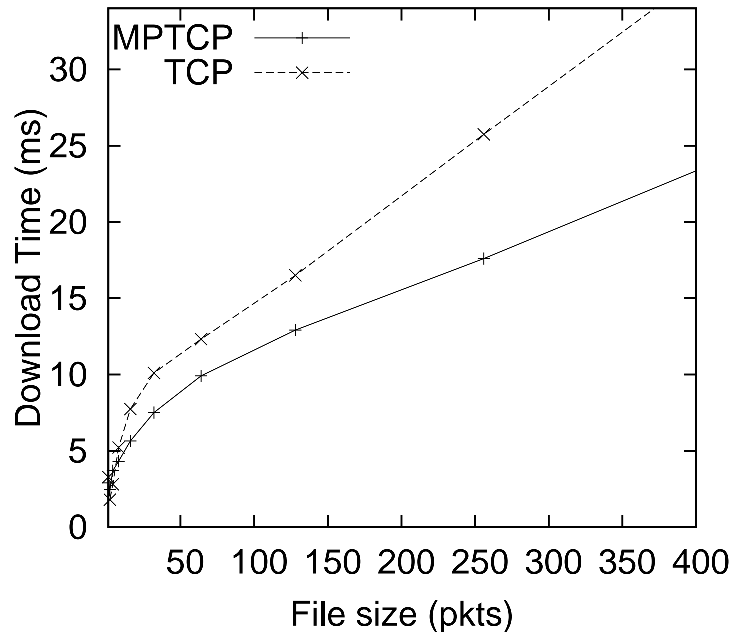

Figure 5.13: Time to transfer short files

window of the first subflow is big enough) and only then start a second subflow.

To achieve such a good result as depicted in Figure 5.13, however, we have needed to revise our implementation of the context estabishment and accept having more code running in sofware interrrupt context compared to the initial version, hence being slightly less fair to other system processes. Our revised version, in the server side, creates an MPTCP meta-socket as soon as the Multipath capable SYN message is received, without waiting until the corresponding application user context is woken up again. This necessarily happens in software interrupt context and is not completely free as it requires atomic memory allocation (as opposed to interruptible user context memory allocation). However, we believe this is acceptable, especially given the benefit it provides, as this is how regular TCP establishment is implemented. We just allocate one more structure (the meta-socket). If the MPTCP structure were not created immediately upon SYN reception, the server could miss the first JOIN request, delaying the use of multiple paths.

MPTCP performance in the Amazon's EC2 cloud: Amazon's EC2 compute cloud[7] allows us to run real-world experiments on a production data-centre. Amazon has several data-centres; their older ones do not appear to have redundant topologies, but their latest data-centres (USEast1c and USEast1d) use a topology that provides many parallel paths between most pairs of virtual machines.

[7] `http://aws.amazon.com/ec2/`

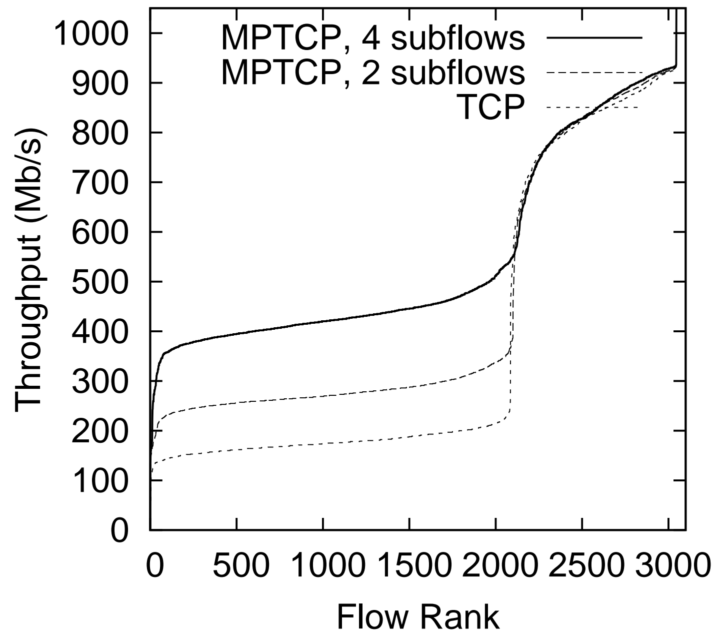

Figure 5.14: 12 hours of throughput, all paths between forty EC2 nodes

We do not know the precise topology of the US East data-centre. The measurements are complicated because each instance is a virtual machine, sharing the physical hardware with other users. Background traffic levels are also unknown to us, and may vary between experiments.

To understand the variability of the environment and the potential for MPTCP to improve performance, our MPTCP-capable Linux kernel has been run on forty EC2 instances, and for 12 hours throughput was sequentially measured with iperf between each pair of hosts, using MPTCP with 2 and 4 subflows and TCP as transport protocols. The resultant dataset totals $3,000$ measurements for each configuration, and samples across both time and topology.

Figure 5.14 shows the results ordered by throughput for each configuration. Traceroute shows that a third of paths have no diversity; of these paths 60% are local to the switch (2 IP hops), while the others have four IP hops. They roughly correspond to the right-hand 35% of the flows in the figure; they achieve high throughput, and their bottleneck is most likely the shared host NIC. MPTCP cannot help these flows; in fact some of these flows show a very slight reduction in throughput; this is likely due to additional system overheads of MPTCP.

The remaining paths are four IP hops, and the number of available paths varies between two (50% of paths), three (25%) up to nine. Traceroute shows all of them implement load balancing across a redundant topology. MPTCP with four subflows achieves three times the throughput of a single-path TCP for almost every path across the entire 12-hour period.

5.5 Conclusions

Multipath TCP is a major extension to TCP that is being developed within the IETF. Its success will depend on the availability of a reference implementation. We have analyzed the performance of our implementation in the HEN testbed, looking at the impact of the delay on the receive buffers and the throughput. We also showed that losses on one subflow had limited impact on the performance of another subflow from the same Multipath TCP connection. From a performance point of view, we have shown that our implementation is able to efficiently utilise Gigabit Ethernet links when using large packets, and identified the next steps for optimisation, based on the current CPU consumption results of our implementation (version 0.6).

Another critical aspect of MPTCP is the coupled congestion control. This has been already developed and evaluated in details by Wischik et al. in [WRGH11]. We showed that Linux MPTCP fulfills the fairness goals described in [RHW11].

Finally, we presented the benefits of MPTCP in a data-centre environment. We explained that current data-centres already leverage multiple paths, and usually attempt to optimise their use by randomisation of flow allocations. We showed that MPTCP can bring high benefits even in existing data-centres (by running our implementation in the Amazon EC2 cloud). Another interesting result is that, even for very short flows, MPTCP can provide lower completion times than regular TCP.

Conclusion

Throughout this thesis, we focused on two IETF proposals that aim at improving the utilisation of multiple paths in the Internet. The first one, shim6, is actually one of the multiple attempts by the IETF to make IPv6 a more successful protocol than IPv4, in particular by enabling effective multihoming in the presence of the new hierarchical address allocation model of IPv6.

The second one is Multipath TCP. This effort is even more ambitious, as it not only tries to improve multihoming, but uses multiple paths *simultaneously*. Furthermore, MPTCP can use both IPv4 and IPv6 in the same connection. Should we have to rank these two protocols based on these sole two paragraphs, we would probably select MPTCP as the clear winner. But MPTCP also has drawbacks compared to shim6. Its broad range of capabilities could become a handicap compared to the very focused shim6 solution.

The main contribution of this thesis is to evaluate the feasibility of these new proposals in real world conditions. For this, we adopted a pragmatic approach, implementing everything including all details. The main benefit of implementation-based evaluations is that this allows to answer important questions like:

- *Does the protocol allow optimisations to run efficiently on high-end servers ?* For shim6, we have proposed a simple way to identify peers that do not support the protocol, based on a simple check of the peer's address. For MPTCP, we have discussed current TCP optimisations and how they can be mapped to a multipath version of TCP.
- *What kind of protocol optimisations can be envisaged and what are the involved tradeoffs ?* For shim6 we have proposed a simple way to find low delay paths, but this is at the cost of more control traffic. We have also proposed that highly loaded servers drop shim6 contexts more aggressively, but again this may generate more shim6 traffic as the clients may attempt to re-create the server state. MPTCP has tradeoffs in the amount of required buffering. More buffering allows getting a higher throughput in the presence of paths with high bandwidth-delay products. But some path configurations may consume a huge amount of memory.
- *Does the wheel really need to be reinvented ?* When a new protocol is developed to answer a particular problem, it is often the case that the protocol

designers reinvent existing functionalities for their new protocol, hence tending to individual monolithic solutions instead of combined protocols. This holds particularly true for shim6 and MIPv6 where several researchers have tried to support mobility with shim6 [RBKY08, DM08] or multihoming with MIPv6 [WDT+09]. But even for the newer MPTCP proposal, several mechanisms are very similar to shim6 (e.g. address knowledge exchange and connection identifier). Although those similar mechanisms for path management solve problems that shim6 cannot solve (in particular IPv4 support is not offered by shim6), shim6 can still be useful as a Path Management service for MPTCP in an IPv6 environment. More generally, we proposed a functional separation of the MPTCP functionality [FRH+11], to allow it running over any path management solution.

- *Are there ambiguities in the protocol specification ?* When evaluating a new protocol, simulations require making simplifying assumptions that may hide some design problems. For instance if the simulations concentrate on congestion control, buffering problems may be ignored. In both shim6 and MPTCP, we have contributed to the improvement of the protocol specifications.

Simulation-based studies are important as well and have fortunately been conducted by others [dBGMS07, RBP+11, WRGH11], so that both shim6 and Multipath TCP benefit now already from a good set of evaluations and consolidations. Obviously MPTCP in particular is still a work in progress and there are several important problems that still need to be solved. To conclude this thesis, we briefly compare the MPTCP and shim6 proposals, and finally summarise the research avenues that remain open.

MPTCP vs Shim6

At first sight, MPTCP could simply replace shim6, its functionality being a superset of shim6. It is capable of exchanging addresses with the peer, and failure recovery is naturally handled by TCP retransmission mechanisms. Even better, the failure recovery time by MPTCP is *always* lower than the shim6 failure recovery time. Indeed, although it would be theoretically possible to configure low enough timeout values for shim6 to beat MPTCP, that would unavoidably imply many useless REAP explorations because of erroneous failure detections.

Trying to find a winner

Beside capabilities that are shared by MPTCP and shim6, MPTCP presents huge advantages, the main one being the *parallel multipath* capability, whereas shim6 has only *sequential multipath* capability. By using multiple paths in parallel, Multipath TCP can just get rid of a REAP-like protocol, as the path exploration is

permanent. Finally, MPTCP is designed to support both IPv4 and IPv6, whereas shim6 is specialized for IPv6. An important consequence is that even if a host has only one IPv4 address and one IPv6 address, it is already multihomed as far as MPTCP is concerned. But shim6 does not lose everywhere. It does present the advantage of being located in the network layer. The layer difference brings the following consequences:

- **Supported protocols:** shim6 can offer multihoming support to any transport layer protocol. MPTCP is limited to offering multihoming support for TCP applications. This is important as from the user perspective, installing MPTCP alone in an IPv6-only environment (this situation is likely to become realistic in the near future) would result in partial failure support. If the user pays for multihoming, he probably expects that even its UDP flows can failover across the providers. If shim6 is not used the non-TCP transport flows are on their own to detect and recover from failures. This can be implemented in the application layer however for UDP flows, because any source address can be used for each datagram. RTP multihoming is also being proposed at the IETF [SKO+11].

- **Amount of state to maintain:** one of the major drivers for the MPTCP design is to require no change to existing applications. But existing applications often open **many** TCP connections to the same destination, in particular web browsers. At the time of designing shim6, people were concerned about the amount of state required by shim6 in high end servers. This motivated our proposal for fast identification of non-shim6 peers and aggressive context removal. But things are much worse with MPTCP: MPTCP multiplies its amount of state by the number of TCP subflows, while shim6 maintains **one** small state per host pair (unless special treatment is requested, like context forking). For each single connection MPTCP maintains a separate connection identifier, it handles the security mechanisms, and creates complete sockets for each of the subflows. Even worse, the individual sockets maintain reordering queues, and a meta-socket (per MPTCP connection) maintains a meta-reordering queue that may grow large, especially in the presence of paths with high-delay-bandwidth products. This is the price to pay for the *parallel multipath* capability. It is important to understand that this is not free in terms of memory consumption, as we have shown in our MPTCP evaluation.

- **Address management:** shim6 is slightly handicapped by the fact that it needs to perform address rewriting, while MPTCP just needs to store the correct addresses in the corresponding sockets, since it handles one socket per path. However this is counter-balanced by the fact that MPTCP needs to run a scheduler at the sending side and an intensive reordering algorithm at the receiving side.

Or are they both losers ?

The title of this section is obviously exagerated. However shim6 and MPTCP share a drawback because of their similar way to handle addresses. In both shim6 and MPTCP, the application uses a stable address, while the actual address in use may change over time. In case of renumbering, it may happen that a host starts using a newly acquired address, while another one still uses the same address at the application layer, although the topological address is now different because of the renumbering. This possibility lead the IETF shim6 Working Group to decide that **shim6 does not support renumbering**, and a shim6 context must be dropped when the ULID is no longer topologically owned by the end-host. This IETF consensus has been reflected in Section 1.5 of [NB09]. To be coherent, one should also assume that **MPTCP does not support renumbering**: it indeed shares the exact same limitation as shim6, given an address can be used at the application layer while being topologically assigned to another host, if no subflow is using the address.

By extension, we can conclude that MPTCP and shim6 do not support mobility, at least if they are used *alone*. Mobility indeed involves regularly losing the ownership of an address, and get another one from another network. Mobility is however possible if the application-layer address (the ULID as per the shim6 terminology) is a stable address belonging to the home network, as is the case for *MipShim6*.

Recently a paper has been published about the use of MPTCP as the main building block of a mobility [RNBH11] architecture. Unfortunately it does not take into account the address ownership problem just mentioned. This problem could however be solved probably by combining MPTCP with Mobile IP, as we did for shim6. Mobile IP would provide a stable, owned address for the application layer, and MPTCP could make use of care-of addresses while moving (or even use several of them simultaneously). An initial architecture that combines MPTCP with Mobile IP has been proposed by Bagnulo et al. in [BEF+10]. In the extreme case, we can imagine that the home network itself could be renumbered, and the home address reassigned. Supporting that case would involve going further into the locator/identifier separation and use non-routeable identifiers in the application layer.

Finally a related problem is the handling of referrals. Both Shim6 and MPTCP present a stable address to the application, while using another one for the routing. A consequence is that the application may advertise that stable address to a third-party, thinking that it works while the corresponding path is actually broken. In both cases the third-party won't be able to reach the referred address, unless it is able to retrieve the corresponding alternate locators thanks to a global mapping system.

Stop fighting and become good friends ?

We have described in Section 4.4.2 our work on combining shim6 with MPTCP. They indeed share some functions regarding path management, that we can avoid duplicating by combining the two protocols:

- **Address exchange:** MPTCP uses TCP options to advertise the local set of addresses to the peer. Shim6 uses the dedicated shim6 protocol. MPTCP and shim6 share the new kinds of attacks that are made possible by host multihoming, and both define their own mechanisms. Multipath TCP is limited however in what it can do, partly by the limited length of TCP options, partly by the time required to perform cryptographic operations. Shim6 on the other hand takes benefit from the long IPv6 addresses and offers a better protection thanks to HBA or CGA addresses. An example of attack that is possible only with MPTCP is when an attacker can listen to the first few packets of a communication. With MPTCP, the attacker will see all the keys and be able to impersonate one or both of the involved hosts. With shim6 this is not possible because a public/private key pair is used to verify the exchanged addresses. One way to enhance MPTCP with shim6-like security would be to integrate the HBA/CGA mechanisms into MPTCP, but such mechanisms would be available only when the initial subflow uses IPv6 addresses (additional subflows can use IPv4 or IPv6, given they are authenticated with the key stored in the initial address).

- **Context identification:** MPTCP uses a token to find the connection corresponding to a new join request (`MP_JOIN`). There is one token per MPTCP connection. With shim6, the equivalent mechanism is a context tag that identifies an association between two hosts. The MPTCP token needs to be used only in the SYN exchange, as further packets are attached to the correct connection thanks to the 5-tuple. Shim6 cannot use the 5-tuple as it is located in the network layer. The context tag is thus included in all packets unless the locators in use are the ULIDs. When shim6 is used as path manager for MPTCP, the MPTCP address exchange options and the token can simply be ignored.

Shim6 seems to be a good candidate for MPTCP path management. However it is not currently designed to support IPv4, which is probably a showstopper, at least until IPv6 becomes largely dominant[8].

Still, shim6 does offer multihoming support for any transport layer protocol, and it can be used for protocols other than TCP. MPTCP could then access both IPv4 and IPv6 networks, and other transport protocols could benefit from multi-

[8] A possible extension to shim6 would be allow IPv4 addresses as alternate locators. The Upper Layer IDentifier must be an IPv6 address, however, as its 128 bits are required by the HBA/CGA security mechanisms. It would also be necessary to define an IPv4 encoding for the shim6 extension header, used to attach a packet to the correct shim6 context.

homing support over the IPv6 topology (IPv4 multihoming support being usually handled by the routing system anyway).

Open perspectives

Our work has opened interesting research opportunities. All of our implementations (*LinShim6*, *MipShim6*, MPTCP over shim6, Linux MPTCP) are publicly available and can certainly be further improved. In particular they can be used to further understand how to improve the shim6 and MPTCP protocols.

In this thesis we have presented a lot of lab measurements. Both shim6 and MPTCP would benefit from experiments across the Internet, in particular to study their deployability, as middleboxes of all kinds are being deployed on the Internet and may filter or transform unknown protocols.

We have mentioned in the previous section the problem of renumbering, faced by both MPTCP and shim6. Currently the most obvious way to solve this problem is to go further into the locator/identifier separation principle: wheras shim6 and MPTCP use a particular locator as identifier, other solutions such as HIP are based on a non-routeable identifier and can thus naturally survive a rehoming event. These approaches are however a greater architectural change compared to the current Internet and their advantages come at the cost of more difficult deployments.

It is interesting to consider how shim6 and MPTCP can be adapted to handle mobility as well. We have proposed a solution for shim6, *MipShim6*, but it has only be shortly evaluated, and the implementation is still a prototype. Likewise, efforts from other researchers lead to an interesting proposal for mobile MPTCP, although we do not support the idea of using an arbitrary address in the application layer. Hence, a modification of MPTCP that integrates Mobile IP would be an interesting research topic.

From an implementation viewpoint, MPTCP still has several important challenges, as mentioned in Section 4.6. In particular it should be optimised for SMP systems and NICs with flow-to-core affinity. The TCP fast-path should be adapted in an equivalent "MPTCP fast path". An efficient algorithm should be devised for handling reordering at the connection level, as the queue sizes involved are much larger compared to regular TCP.

Finally, we have used our MPTCP implementation to evaluate the protocol for the use case of data-centre deployments. MPTCP is a very promising solution, and many other use cases can be evaluated. An example is the case of mobile devices with multiple interfaces, where mobile operators could offer improved service with MPTCP.

Bibliography

[AAK+02] J. Arkko, T. Aura, J. Kempf, V.M. Mantyla, P. Nikander, and M. Roe. Securing IPv6 Neighbor and Router Discovery. In *Workshop on Wireless Security (WiSe)*, 1st, pages 77–86. ACM, September 2002.

[AAN07] APNIC, ARIN, and RIPE NCC. IPv6 Address Allocation and Assignment Policy. ripe-421, November 2007.

[AFLV08] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, pages 63–74. ACM, August 2008.

[AFRR+10] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 7th. USENIX Association, April 2010.

[AFS08] V. Aggarwal, A. Feldmann, and C. Scheideler. Can ISPs and P2P systems co-operate for improved performance? *ACM SIGCOMM Computer Communications Review (CCR)*, 37(3):29–40, 2008.

[AH08] J. Ahrenholz and T. Henderson. shim6 implementation. `http://www.openhip.org/docs/shim6.pdf`, 2008.

[AKP11] A. Achour, B. Kervella, and G. Pujolle. Shim6-based mobility management for multi-homed terminals in heterogeneous environment. In *International Conference on Wireless and Optical Communications Networks (WOCN)*, 8th. IEEE, May 2011.

[AKZN05] J. Arkko, J. Kempf, B. Zill, and P. Nikander. SEcure Neighbor Discovery (SEND). RFC 3971 (Proposed Standard), March 2005.

[ALD+05] J. Abley, K. Lindqvist, E. Davies, B. Black, and V. Gill. IPv4 Multihoming Practices and Limitations. RFC 4116 (Informational), July 2005.

[APB09] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), September 2009.

[ASL04] A. Abd El Al, T. N. Saadawi, and M. J. Lee. LS-SCTP: a bandwidth aggregation technique for stream control transmission protocol. *Comput. Commun.*, 27(10):1012–1024, June 2004.

[Atk11] R. Atkinson. ILNP Concept of Operations. Internet draft, draft-rja-ilnp-intro-11.txt, work in progress, July 2011.

[Aug10] B. Augustin. *Tracing Internet Routes under Load Balancing*. PhD thesis, October 2010.

[Aur03] T. Aura. Cryptographically Generated Addresses (CGA) . In *Information Security Conference (ISC)*, volume LNCS 2851/2003 of *6th*, pages 29–43, Bristol, UK, October 2003. Springer.

[Aur05] T. Aura. Cryptographically Generated Addresses (CGA). RFC 3972 (Proposed Standard), March 2005. Updated by RFCs 4581, 4982.

[AvB09] J. Arkko and I. van Beijnum. Failure Detection and Locator Pair Exploration Protocol for IPv6 Multihoming. RFC 5534 (Proposed Standard), June 2009.

[Bag08] M. Bagnulo. Proxy Shim6 (P-Shim6). Internet draft, draft-bagnulo-pshim6-02.txt, expired, February 2008.

[Bag09] M. Bagnulo. Hash-Based Addresses (HBA). RFC 5535 (Proposed Standard), June 2009.

[Bag11] M. Bagnulo. Threat Analysis for TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6181 (Informational), March 2011.

[Bar08] S. Barré. Linshim6 - implementation of the shim6 protocol. Technical report, Université catholique de Louvain, February 2008.

[BB07a] S. Barré and O. Bonaventure. Implementing SHIM6 using the Linux XFRM framework. In *Routing In Next Generation workshop*, Madrid, Spain, December 2007.

[BB07b] S. Barré and O. Bonaventure. Improved Path Exploration in shim6-based Multihoming. In *SIGCOMM Workshop "IPv6 and the Future of the Internet"*, Kyoto, Japan, August 2007. ACM.

[BB09] S. Barré and O. Bonaventure. Shim6 Implementation Report : LinShim6. Internet draft, draft-barre-shim6-impl-03.txt, expired, September 2009.

[BDI+11] M. Becke, T. Dreibholz, J. Iyengar, P. Natarajan, and M. Tuexen. Load Sharing for the Stream Control Transmission Protocol (SCTP).

Internet draft, draft-tuexen-tsvwg-sctp-multipath-02.txt, work in progress, July 2011.

[BDMB09] S. Barré, A. Dhraief, N. Montavont, and O. Bonaventure. MipShim6: une approche combinée pour la mobilité et la multi-domiciliation. In *Colloque Francophone sur l'Ingénierie des Protocoles (CFIP)*, 14, pages 113–124. HAL - CCSD, October 2009.

[BEF+10] M. Bagnulo, P. Eardley, A. Ford, A. García-Martínez, A. Kostopoulos, C. Raiciu, and F. Valera. Boosting mobility performance with multi-path tcp. In *Future Network and Mobile Summit*, pages 1–8. IEEE, June 2010.

[Ben65] V. Benes. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Academic Press, New York, 1965.

[Ber] D.J. Bernstein. SYN Cookies. http://cr.yp.to/syncookies.html, visited in April 2011.

[BFF07] O. Bonaventure, C. Filsfils, and P. Francois. Achieving sub-50 milliseconds recovery upon BGP peering link failures. *IEEE/ACM Transactions on Networking (TON)*, 15(5):1123–1135, October 2007.

[BGMA07] M. Bagnulo, A. García-Martínez, and A. Azcorra. IPv6 multihoming support in the mobile internet. *Wireless Communications, IEEE*, 14(5):92–98, December 2007.

[BKG+01] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135 (Informational), June 2001.

[BOP94] L.S. Brakmo, S.W. O'Malley, and L.L. Peterson. *TCP Vegas: New techniques for congestion detection and avoidance*, volume 24. ACM, October 1994.

[Bor89] D.A. Borman. Implementing TCP/IP on a cray computer. *ACM SIGCOMM Computer Communications Review (CCR)*, 19:11–15, April 1989.

[BPB11a] S. Barré, C. Paasch, and O. Bonaventure. MultiPath TCP - Guidelines for implementers. Internet draft, draft-barre-mptcp-impl-00.txt, expired, March 2011.

[BPB11b] S. Barré, C. Paasch, and O. Bonaventure. MultiPath TCP: From Theory to Practice. In *IFIP Networking*, May 2011.

[BRB11] S. Barré, J. Ronan, and O. Bonaventure. Implementation and evaluation of the Shim6 protocol in the Linux kernel. *Comput. Commun.*, 34(14):1685–1695, September 2011.

[BRBH10] S. Barré, C. Raiciu, O. Bonaventure, and M. Handley. Experimenting with Multipath TCP. In *SIGCOMM 2010 Demo*, September 2010.

[BS04] F. Baker and P. Savola. Ingress Filtering for Multihomed Networks. RFC 3704 (Best Current Practice), March 2004.

[BS10] M. Blanchet and P. Seite. Multiple Interfaces and Provisioning Domains Problem Statement. Internet draft, draft-ietf-mif-problem-statement-15.txt, work in progress, May 2010.

[CAF10] B. Carpenter, R. Atkinson, and H. Flinck. Renumbering Still Needs Work. RFC 5887 (Informational), May 2010.

[CAG05] S. Cheshire, B. Aboba, and E. Guttman. Dynamic Configuration of IPv4 Link-Local Addresses. RFC 3927 (Proposed Standard), May 2005.

[CD98] A. Conta and S. Deering. Generic Packet Tunneling in IPv6 Specification. RFC 2473 (Proposed Standard), December 1998.

[Ciz05] G. Cizault. *IPv6 - Théorie et Pratique*. O'Reilly, 4th edition, November 2005.

[Cla88] David D. Clark. The Design Philosophy of the DARPA Internet Protocols. *ACM SIGCOMM Computer Communications Review (CCR)*, pages 106–114, August 1988.

[Cla04] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), October 2004.

[Clo53] C. Clos. A study of non-blocking switching networks. *Bell System Tech. J.*, 32:406–424, 1953.

[Cra98] M. Crawford. Transmission of IPv6 Packets over Ethernet Networks. RFC 2464 (Proposed Standard), December 1998. Updated by RFC 6085.

[CSP+11] B. Carr, O. Sury, J. Palet Martinez, A. Davidson, R. Evans, F. Yilmaz, and I. Wijte. IPv6 Address Allocation and Assignment Policy. ripe-512, February 2011.

[Cur04] A. Currid. TCP Offload to the Rescue. *Queue*, 2:58–65, May 2004.

[dB07] C. de Launois and M. Bagnulo. The Paths towards IPv6 Multihoming. *IEEE Communications Surveys and Tutorials*, 8(2):38–51, February 2007.

[dBGMS07] A. de la Oliva, M. Bagnulo, A. García-Martínez, and I. Soto. Performance Analysis of the REAchability Protocol for IPv6 Multihoming. In *Conference on Next Generation Teletraffic and Wired/Wireless Advanced Networking (NEW2AN)*, September 2007.

[DBV+03] R. Droms, J. Bound, B. Volz, T. Lemon, C. Perkins, and M. Carney. Dynamic Host Configuration Protocol for IPv6 (DHCPv6). RFC 3315 (Proposed Standard), July 2003. Updated by RFCs 4361, 5494, 6221.

[de 05] C. de Launois. *Unleashing Traffic Engineering for IPv6 Multihomed Sites*. PhD thesis, Université catholique de Louvain, September 2005.

[DH98] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), December 1998. Updated by RFCs 5095, 5722, 5871.

[DM08] A. Dhraief and N. Montavont. Toward Mobility and Multihoming Unification - The SHIM6 Protocol: A Case Study. In *Wireless Communications and Networking Conference (WCNC)*, pages 2840–2845, Las Vegas, April 2008. IEEE.

[DM09] A. Dhraief and N. Montavont. Rehoming decision algorithm: design and empirical evaluation. In *International Conference on Computational Science and Engineering*, pages 464–469. IEEE, August 2009.

[DPN03] G. Daley, B. Pentland, and R. Nelson. Movement detection optimizations in mobile IPv6. In *International Conference on Networks (ICON)*, 11th, pages 687–692. IEEE, September 2003.

[Dra03] R. Draves. Default Address Selection for Internet Protocol version 6 (IPv6). RFC 3484 (Proposed Standard), February 2003.

[Dro97] R. Droms. Dynamic Host Configuration Protocol. RFC 2131 (Draft Standard), March 1997. Updated by RFCs 3396, 4361, 5494.

[dVHD+07] G. Van de Velde, T. Hain, R. Droms, B. Carpenter, and E. Klein. Local Network Protection for IPv6. RFC 4864 (Informational), May 2007.

[EH06] D. Eastlake 3rd and T. Hansen. US Secure Hash Algorithms (SHA and HMAC-SHA). RFC 4634 (Informational), July 2006. Obsoleted by RFC 6234.

[FB07] P. Francois and O. Bonaventure. Avoiding transient loops during the convergence of link-state routing protocols. *IEEE/ACM Transactions on Networking (TON)*, 15(6):1280–1932, December 2007.

[Fer11] G. Ferro. The Importance of Provider Independent IPv6 Addressing. http://etherealmind.com/importance-provider-independent-ipv6-addresses/, January 2011.

[FF01] M. Fisk and W. Feng. Dynamic right-sizing in TCP. In *Los Alamos Computer Science Institute Symposium (LACSI)*, 2nd, October 2001.

[FFML11] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis. Locator/ID Separation Protocol (LISP). Internet draft, draft-ietf-lisp-15.txt, work in progress, March 2011.

[FJ94] S. Floyd and V. Jacobson. The synchronization of periodic routing messages. *IEEE/ACM Transactions on Networking (TON)*, 2(2):122–136, April 1994.

[FL06] V. Fuller and T. Li. Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan. RFC 4632 (Best Current Practice), August 2006.

[FRH+11] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural Guidelines for Multipath TCP Development. RFC 6182 (Informational), March 2011.

[FRHB11] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. Internet draft, draft-ietf-mptcp-multiaddressed-04.txt, work in progress, July 2011.

[Gao01] Lixin Gao. On Inferring Autonomous System Relationships in the Internet . *IEEE/ACM Transactions on Networking (TON)*, 9(6), December 2001.

[GHJ+09] A. Greenberg, J.R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D.A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. *ACM SIGCOMM Computer Communications Review (CCR)*, 39(4):51–62, October 2009.

[GLL+09] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: a high performance, server-centric network architecture for modular data centers. *ACM SIGCOMM Computer Communications Review (CCR)*, 39(4):63–74, October 2009.

[GN00] R. Gilligan and E. Nordmark. Transition Mechanisms for IPv6 Hosts and Routers. RFC 2893 (Proposed Standard), August 2000. Obsoleted by RFC 4213.

[HD06] R. Hinden and S. Deering. IP Version 6 Addressing Architecture. RFC 4291 (Draft Standard), February 2006. Updated by RFCs 5952, 6052.

[HH05] R. Hinden and B. Haberman. Unique Local IPv6 Unicast Addresses. RFC 4193 (Proposed Standard), October 2005.

[HP06] J. L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers Inc., Palo Alto, CA 94303, September 2006.

[HPK01] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *USENIX Security Symposium*, 10th. USENIX Association, August 2001.

[HRX08] S. Ha, I. Rhee, and L. Xu. CUBIC: A new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, July 2008.

[HS02] H.-Y. Hsieh and R. Sivakumar. pTCP: An End-to-End Transport Layer Protocol for Striped Connections. In *International Conference on Network Protocols (ICNP)*, 10th, pages 24–33. IEEE, November 2002.

[HSH+06] H. Han, S. Shakkottai, CV Hollot, R. Srikant, and D. Towsley. Multipath TCP: a joint congestion control and routing scheme to exploit path diversity in the internet. *IEEE/ACM Transactions on Networking (TON)*, 14(6):1260–1271, December 2006.

[Hus05] G. Huston. Architectural Approaches to Multi-homing for IPv6. RFC 4177 (Informational), September 2005.

[Hus10] G. Huston. Addressing 2009. ISP Column, `http://www.potaroo.net/ispcol/2010-01/addresses-2009.html`, January 2010.

[IAS06] J. Iyengar, P.D. Amer, and R.R. Stewart. Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths. *IEEE/ACM Transactions on Networking (TON)*, 14(5):951–964, October 2006.

[IAS07] J. Iyengar, P.D. Amer, and R.R. Stewart. Performance implications of a bounded receive buffer in concurrent multipath transfer. *Comput. Commun.*, 30(4):818–829, February 2007.

[ISO94] ISO/IEC. Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model. ISO/IEC 7498-1, November 1994.

[Jac93] V. Jacobson. Re: query about tcp header on tcp-ip. Available at ftp://ftp.ee.lbl.gov/email/vanj.93sep07.txt, September 1993.

[JLN11] E. Jankiewicz, J. Loughney, and T. Narten. IPv6 Node Requirements. Internet draft, draft-ietf-6man-node-req-bis-11.txt, work in progress, May 2011.

[JPA04] D. Johnson, C. Perkins, and J. Arkko. Mobility Support in IPv6. RFC 3775 (Proposed Standard), June 2004. Obsoleted by RFC 6275.

[KBSS11] M. Komu, M. Bagnulo, K. Slavov, and S. Sugimoto. Sockets Application Program Interface (API) for Multihoming Shim. RFC 6316 (Informational), July 2011.

[KLS04] M. Kodialam, TV Lakshman, and S. Sengupta. Efficient and robust routing of highly variable traffic. In *Workshop on Hot Topics in Networks (HotNets)*, III. Citeseer, November 2004.

[KMC+00] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, August 2000.

[KME04] M. Kanda, K. Miyazawa, and H. Esaki. USAGI IPv6 IPsec Development for Linux. In *International Symposium on Applications and the Internet*, pages 159–163, January 2004.

[KMN+07] S. Krishnan, N. Montavont, E. Njedjou, S. Veerepalli, and A. Yegin. Link-Layer Event Notifications for Detecting Network Attachments. RFC 4957 (Informational), August 2007.

[KS05] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), December 2005. Updated by RFC 6040.

[KV05] F. Kelly and T. Voice. Stability of end-to-end algorithms for joint routing and rate control. *ACM SIGCOMM Computer Communications Review (CCR)*, 35(2):5–12, April 2005.

[KVG07] A. Khurri, E. Vorobyeva, and A. Gurtov. Performance of host identity protocol on lightweight hardware. In *International workshop on Mobility in the evolving internet architecture (MobiArch)*, 2nd. ACM, August 2007.

[KWRG04] J. Kempf, J. Wood, Z. Ramzan, and C. Gentry. IP Address Authorization for Secure Address Proxying using Multi-key CGAs and Ring Signatures. *IEICE TRANSACTIONS on Communications*, E87-B(3):429–436, March 2004.

[L+06] G. Leduc et al. An open source traffic engineering toolbox. *Comput. Commun.*, 29:593–610, March 2006.

[LB09] D. Leroy and O. Bonaventure. Preparing network configurations for IPv6 renumbering. *International Journal of Network Management, Wiley InterScience*, 19(5):415–426, Sept-Oct 2009.

[LBS08] S. Liu, T. Basar, and R. Srikant. TCP-Illinois: A loss-and delay-based congestion control algorithm for high-speed networks. *Performance Evaluation*, 65(6-7):417–440, June 2008.

[Lei85] C. E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, c-34(10), October 1985.

[LIJM+10] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian. Internet inter-domain traffic. *ACM SIGCOMM Computer Communications Review (CCR)*, 40(4):75–86, October 2010.

[LLS07] Y.T. Li, D. Leith, and R.N. Shorten. Experimental evaluation of TCP Protocols for High-Speed Networks. *IEEE/ACM Transactions on Networking (TON)*, 15:1109–1122, October 2007.

[Lou06] J. Loughney. IPv6 Node Requirements. RFC 4294 (Informational), April 2006. Updated by RFC 5095.

[LWZ08] J. Liao, J. Wang, and X. Zhu. cmpSCTP: An extension of SCTP to support concurrent multi-path transfer. In *International Conference on Communications (ICC)*, pages 5762–5766. IEEE, May 2008.

[Mek07] M. Mekking. Formalization and verification of the shim6 protocol. Master's thesis, Radboud University - NLnet Labs, May 2007.

[MFB+11] P. Mérindol, P. Francois, O. Bonaventure, S. Cateloin, and J.-J. Pansiot. An efficient algorithm to enable path diversity in link state routing networks. *Computer Networks*, 55(5):1132–1149, April 2011.

[MFHK08a] A. Matsumoto, T. Fujisaki, R. Hiromi, and K. Kanayama. Problem Statement for Default Address Selection in Multi-Prefix Environments: Operational Issues of RFC 3484 Default Rules. RFC 5220 (Informational), July 2008.

[MFHK08b] A. Matsumoto, T. Fujisaki, R. Hiromi, and K. Kanayama. Requirements for Address Selection Mechanisms. RFC 5221 (Informational), July 2008.

[MK01] L. Magalhaes and R. Kravets. Transport Level Mechanisms for Bandwidth Aggregation on Mobile Hosts. In *International Conference on Network Protocols (ICNP)*, 9th, pages 165–171. IEEE Computer Society, November 2001.

[MKS+07] K. Mitsuya, R. Kuntz, S. Sugimoto, R. Wakikawa, and J. Murai. A policy management framework for flow distribution on multihomed end nodes. In *International workshop on Mobility in the evolving internet architecture (MobiArch)*, 2nd. ACM, August 2007.

[MN02] N. Montavont and T. Noel. Handover management for mobile nodes in IPv6 networks. *IEEE Communications Magazine*, 40(8):38–43, August 2002.

[MN04] K. Miyazawa and M. Nakamura. IPv6 IPsec and Mobile IPv6 Implementation of Linux. In *Proceedings of the Linux Symposium*, volume 2, pages 371–380, July 2004.

[MN06a] N. Montavont and T. Noel. Fast movement detection in IEEE 802.11 networks. *Wirel. Commun. Mob. Comput.*, 6(5):651–671, July 2006.

[MN06b] R. Moskowitz and P. Nikander. Host Identity Protocol (HIP) Architecture. RFC 4423 (Informational), May 2006.

[MNJH08] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson. Host Identity Protocol. RFC 5201 (Experimental), April 2008. Updated by RFC 6253.

[MZF07] D. Meyer, L. Zhang, and K. Fall. Report from the IAB Workshop on Routing and Addressing. RFC 4984 (Informational), September 2007.

[Nag84] J. Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, January 1984.

[NB09] E. Nordmark and M. Bagnulo. Shim6: Level 3 Multihoming Shim Protocol for IPv6. RFC 5533 (Proposed Standard), June 2009.

[ND01] T. Narten and R. Draves. Privacy Extensions for Stateless Address Autoconfiguration in IPv6. RFC 3041 (Proposed Standard), January 2001. Obsoleted by RFC 4941.

[NHVA08] P. Nikander, T. Henderson, C. Vogt, and J. Arkko. End-Host Mobility and Multihoming with the Host Identity Protocol. RFC 5206 (Experimental), April 2008.

[NL05] E. Nordmark and T. Li. Threats Relating to IPv6 Multihoming Solutions. RFC 4218 (Informational), October 2005.

[NNSS07] T. Narten, E. Nordmark, W. Simpson, and H. Soliman. Neighbor Discovery for IP version 6 (IPv6). RFC 4861 (Draft Standard), September 2007. Updated by RFC 5942.

[Nor01] W.B. Norton. Internet service providers and peering. In *Proceedings of NANOG*, volume 19 of *23rd*, pages 1–17. Citeseer, October 2001.

[NRO10] NRO. Remaining IPv4 Address Space Drops Below 5% . http://www.nro.net/news/remaining-ipv4-address-space-drops-below-5, October 2010. [Retrieved 2011-03-17].

[NZNP11] S.C. Nguyen, X. Zhang, T.M.T. Nguyen, and G. Pujolle. Evaluation of throughput optimization and load sharing of multipath tcp in heterogeneous networks. In *International Conference on Wireless and Optical Communications Networks (WOCN)*, 8th. IEEE, May 2011.

[Ong09] A. Ongenae. Multi-path congestion control. Master's thesis, Université Catholique de Louvain (Belgium), 2009.

[Pal09] J. Palet Martinez. Provider Independent (PI) IPv6 Assignments for End User Organisations. Ripe Policy Proposal 2006-01, February 2009.

[PCC+07] K. Park, H. Cho, Y. Choi, T. Kwon, T. You, and S. Lee. The Implementation of Layer-three Site Multihoming Protocol (L3SHIM). In *International Conference on Advanced Communication Technology*, 9th, pages 234–237, February 2007.

[Per10] C. Perkins. IP Mobility Support for IPv4, Revised. RFC 5944 (Proposed Standard), November 2010.

[PJ01] C. Perkins and D.B. Johnson. Route Optimization in Mobile IP. Internet draft, draft-ietf-mobileip-optim-11.txt, expired, September 2001.

[Pos80] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[Pos81a] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[Pos81b] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168, 6093.

[RA08] M.S. Rahman and M. Atiquzzaman. SEMO6 - a multihoming-based seamless mobility management framework. In *IEEE Military Communications Conference (MILCOM)*, November 2008.

[RBKY08] J. Ronan, S. Balasubramaniam, A.K. Kiani, and W. Yao. On the use of SHIM6 for mobility support in IMS networks. In *International Conference on Testbeds and research infrastructures for the development of networks & communities (TRIDENTCOM)*, 4th, May 2008.

[RBP+11] C. Raiciu, S. Barré, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving data center performance and robustness with multipath TCP. In *SIGCOMM*, Toronto, Canada, August 2011.

[RHW11] C. Raiciu, M. Handley, and D. Wischik. Coupled Congestion Control for Multipath Transport Protocols. Internet draft, draft-ietf-mptcp-congestion-07.txt, work in progress, July 2011.

[RLH06] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), January 2006. Updated by RFC 6286.

[RM10] J. Ronan and J. McLaughlin. An empirical evaluation of a Shim6 implementation. In *International ICST Conference on Mobile Networks and Management*, 2nd, September 2010.

[RNBH11] C. Raiciu, D. Niculescu, M. Bagnulo, and M. Handley. Opportunistic Mobility with Multipath TCP . In *International workshop on Mobility in the evolving internet architecture (MobiArch)*, 6th, June 2011.

[ROA05] K. Rojviboonchai, T. Osuga, and H. Aida. R-M/TCP: Protocol for Reliable Multi-Path Transport over the Internet. In *International Conference on Advanced Information Networking and Applications (AINA)*, 19th, pages 801–806. IEEE, March 2005.

[RPB+10] C. Raiciu, C. Pluntke, S. Barré, A. Greenhalgh, D. Wischik, and M. Handley. Data Centre Networking with Multipath TCP. In *Workshop on Hot Topics in Networks (HotNets)*, IX, Monterey, California, US, October 2010. ACM.

[SB09] J. Seedorf and E. Burger. Application-Layer Traffic Optimization (ALTO) Problem Statement. RFC 5693 (Informational), October 2009.

[SBS+10] M. Scharf, T.R. Banniza, P. Schefczik, A. Singh, and A. Timm-Giel. Evaluation and prototyping of multipath protocol mechanisms. In *Euroview*, August 2010.

[Sch05] J. Schiller. Inter-AS Traffic Engineering Case Studies as Requirements for IPv6 Multihoming Solutions. NANOG 34 presentation, May 2005.

[SCW+11] Y. Sun, Y. Cui, W. Wang, T. Ma, Y. Ismailov, and X. Zheng. Mobility support in multi-path tcp. In *International Conference on Communication Software and Networks (ICCSN)*, 3rd, pages 195–199. IEEE, May 2011.

[SE01] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022 (Informational), January 2001.

[SF11] M. Scharf and A. Ford. MPTCP Application Interface Considerations. Internet draft, draft-ietf-mptcp-api-02, work in progress, June 2011.

[SGTG+11] A. Singh, C. Görg, A. Timm-Giel, M. Scharf, and T.R. Banniza. Performance evaluation of multipath tcp linux implementations. In *Würzburg Workshop on IP: Joint ITG and Euro-NF Workshop "Visions of Future Generation Networks"*, 11th, August 2011.

[Sha08] S. Shankland. Google spotlights data center inner workings. http://news.cnet.com/8301-10784_3-9955184-7.html, May 2008. [retrieved 2011-05-24].

[SKKK03] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Linux Netlink as an IP Services Protocol. RFC 3549 (Informational), July 2003.

[SKO+11] V. Singh, T. Karkkainen, J. Ott, S. Ahsan, and L. Eggert. Multipath RTP (MPRTP). Internet draft, draft-singh-avtcore-mprtp-02, work in progress, July 2011.

[Ste97] W. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC 2001 (Proposed Standard), January 1997. Obsoleted by RFC 2581.

[Ste07] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), September 2007. Updated by RFCs 6096, 6335.

[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. RFC 2960 (Proposed Standard), October 2000. Obsoleted by RFC 4960, updated by RFC 3309.

[TH00] D. Thaler and C. Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. RFC 2991 (Informational), November 2000.

[TNJ07] S. Thomson, T. Narten, and T. Jinmei. IPv6 Stateless Address Autoconfiguration. RFC 4862 (Draft Standard), September 2007.

[TZL10] D. Thaler, L. Zhang, and G. Lebovitz. IAB Thoughts on IPv6 Network Address Translation. RFC 5902 (Informational), July 2010.

[Vog08] C. Vogt. Six/one router: a scalable and backwards compatible solution for provider-independent addressing. In *International workshop on Mobility in the evolving internet architecture (MobiArch)*, 3rd, pages 13–18. ACM, August 2008.

[Wat10] R. Watson. Protocol stacks and multicore scalability. Presentation at Maastricht IETF78 MPTCP workshop, July 2010.

[WB11] M. Wasserman and F. Baker. IPv6-to-IPv6 Network Prefix Translation. RFC 6296 (Experimental), June 2011.

[WDT+09] R. Wakikawa, V. Devarapalli, G. Tsirtsis, T. Ernst, and K. Nagami. Multiple Care-of Addresses Registration. RFC 5648 (Proposed Standard), October 2009. Updated by RFC 6089.

[WHB08] D. Wischik, M. Handley, and M. Bagnulo. The Resource Pooling Principle. *ACM SIGCOMM Computer Communications Review (CCR)*, 38(5):47–52, October 2008.

[WPR+04] K. Wehrle, F. Pahlke, H. Ritter, D. Muller, and M. Bechler. *The Linux networking architecture: design and implementation of network protocols in the Linux kernel*. Prentice Hall, 2004.

[WR11] R. Winter and A. Ripke. Multipath TCP Support for Single-homed End-systems. Internet draft, draft-wr-mptcp-single-homed-01.txt, work in progress, June 2011.

[WRGH11] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 8th, April 2011.

[YMN+04] H. Yoshifuji, K. Miyazawa, M. Nakamura, Y. Sekiya, H. Esaki, and J. Murai. Linux IPv6 Stack Implementation Based on Serialized Data State Processing. *IEICE TRANSACTIONS on Communications*, E87-B(3):429–436, March 2004.

[ZLK04] M. Zhang, Junwen Lai, and A. Krishnamurthy. A transport layer approach for improving end-to-end performance and robustness using redundant paths. In *USENIX*, pages 99–112, June 2004.

[ZSM04] R. Zhang-Shen and N. McKeown. Designing a predictable internet backbone network. In *Workshop on Hot Topics in Networks (HotNets)*, III, November 2004.