Multipath Aggregation of Heterogeneous Access Networks

by

Dominik Kaspar

Doctoral Dissertation submitted to
the Faculty of Mathematics and Natural Sciences
at the University of Oslo
in partial fulfilment of the requirements for
the degree of Philosophiae Doctor

September 2011

Abstract

The explosive deployment of wired and wireless communication infrastructure has recently enabled many novel applications and sparked new research problems. One of the unsolved issues in today's Internet – the main topic of this thesis – is the goal of increasing data transfer speeds of end hosts by aggregating and simultaneously using multiple network interfaces. This objective is most interesting when the Internet is accessible through several, relatively slow and variable (typically wireless) networks, which are unable to single-handedly provide the required data rate for resource-intensive applications, such as bulk file transfers and high-definition multimedia streaming.

Communication devices equipped with multiple network interfaces are now commonplace. Smartphones and laptops are often shipped with built-in network adapters of different wireless technologies, typically enabling them to connect to wireless local area networks and cellular data networks (such as WLAN and HSPA). At the same time, wireless network coverage has become so widespread that mobile devices are often located in overlapping coverage areas of independent access networks. However, even if multiple interfaces are successfully connected to the Internet, operating systems typically use only a single default interface for data transmission, leaving secondary interfaces idle. This technical restriction is based on the fact that the majority of current Internet traffic is conveyed by transport protocols (TCP and UDP) that do not support multiple IP addresses per endpoint. Another cruicial factor is that path heterogeneity introduces packet reordering, which can negatively affect the performance of transport protocols and strain the buffer requirements of applications.

This thesis explores the problem of multipath aggregation with the attempt to find solutions that achieve increased data throughput by concurrently utilizing heterogeneous and dynamic access networks. We strive for approaches that support existing applications without requiring significant modifications to the current infrastructure. The exploration starts at the IP layer with a study of the IP packet reordering that is caused by the use of heterogeneous paths. Our practical experiments confirm that multipath reordering exceeds the typical packet reordering in the Internet by an extent that renders the usual reordering metric useless. The outcome of our network-layer study is a proxy-based, adaptive multipath scheduler that is able to mitigate packet reodering while transparently forwarding a single transport connection over multiple paths.

After introducing a novel metric for quantifying the benefit of path aggregation, our analysis continues on the transport layer, where we investigate TCP's resilience to multipath-inflicted packet reordering. While IP packet reordering is known for its destructive effect on TCP's performance, our practical experiments indicate that a modern implementation is significantly more robust to packet reordering than standard TCP and achieves a substantial aggregation benefit even in certain cases of extreme multipath heterogeneity. In addition, we run a large set of emulation-based multipath experiments and identify several TCP parameters that lead to improved multipath performance when correctly tuned.

Finally, we present an application-layer solution that builds upon the idea of logical file segmentation for streaming a single video to multihomed clients. The novelity of this approach lies in diverting standard protocol features (i.e., HTTP pipelining and range retrieval requests) from their intended purpose and using them for scheduling video segments over different paths. Interoperable with existing server infrastructure, our proposed solution can be deployed in a lightweight and purely client-based manner. We validate the proposed algorithms by implementing them into an existing video streaming platform.

Acknowledgements

This dissertation is the result of a collaboration between many people and I would like to express my gratitude to all those who contributed in a direct or indirect way to the final outcome. Foremost, I thank my advisers Prof. Paal E. Engelstad, Dr. Audun F. Hansen, Prof. Carsten Griwodz and Prof. Pål Halvorsen for making this PhD project possible, taking part in many fruitful discussions, and for guiding me through my studies. I also thank my colleague Kristian Evensen for his active collaboration and ambition to form a productive research team. Together with Saif Shams, Hung Quoc Vo, Hai Ngoc Pham and Pål Grønsund, the mood in our lab was always pleasant and inspiring.

Special thanks go to my brother Peter and to my wife Eunah. With their knowledge and patience, they were like secret advisers to me. With her farsight and management skills, Eunah always kept me focussed and fuelled my motivation when things did not go as planned. Although busy himself with his doctoral studies, mathematical discussions with Peter always sparked new ideas and lead to a clearer understanding of my own research field. I also thank my parents Christine and Hans Peter for immense support, dedication, love, and enduring the far distance to Norway.

There are many people that gave me good advice during my studies. Knut-Helge Vik often encouraged me with helpful tips from his own doctoral studies. Detlef Bosau taught me more about TCP than he is willing to admit. Adam Piórowicz and Tom Ekeberg answered many of my questions about patenting and the complicated language related to it. Haakon Riiser and Andreas Petlund helped me out numerous times with detailed questions about the Linux kernel and network programming in general.

Last but not least, I would like to express my thanks everyone who made my PhD studies more fun at work, on conferences, and during my free time (in no particular order): Jie Xiang and Wenjie Wei for welcoming me to their flat. Shaukat Ali for competitive breaks at the pool table. Marko Höyhtyä for hunting pythons in Capetown. Peng Peng Ni, Tomas Kupka and Åshild Grønstad Solheim for many inspiring lunchtime conversations. Matthias Wille for being my best mate. Ahmed Elmokashfi for introducing me to Belgian Ramadan. Aiko Yamashita for being congenial like a dolphin. The Simula coffee machine for an estimated 340 liters of energy. Alexander Eichhorn for an exciting visit to Ernen. Tatiana Polishchuk for exchanging interesting ideas. Sofiene Jelassi for keeping in touch after Tunisia. Rajwinder Kaur Panesar-Walawege for her positive aura.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Internet was born in the year 1969, when the first computer communication network was established that spanned an area larger than a laboratory. At that time, the Internet was called the ARPANET, named after the Advanced Research Projects Agency (ARPA), and it contained 50 Kbit/s links between three American universities and the Stanford Research Institute [170]. It was on October 29 of the same year when the world's first datagram traveled over a long-distance, packet-switched network. From this point onward, communication technology progressed rapidly (see Figure 1.1). The size of the Internet increased from a thousand hosts in the year 1984 [170] to well over a billion Web users in the year 2010 [6]. While the first phone call from a handheld mobile phone was made in 1973 [148], there were an estimated four billion mobile phone subscribers in the year 2008 [74], and more than ten million people worldwide are currently connecting to the Internet exclusively through a wireless access network [37]. In addition, the speed of Internet connections has grown at an extreme pace. While the Internet backbone was upgraded to 1.5 Mbit/s in the year 1988 [170], average Internet connection speeds of 5 Mbit/s are now common in people's homes in many countries around the world [6]. Connectivity to the global information flow has become so prevalent that Internet access is even considered by some as a human right [21].

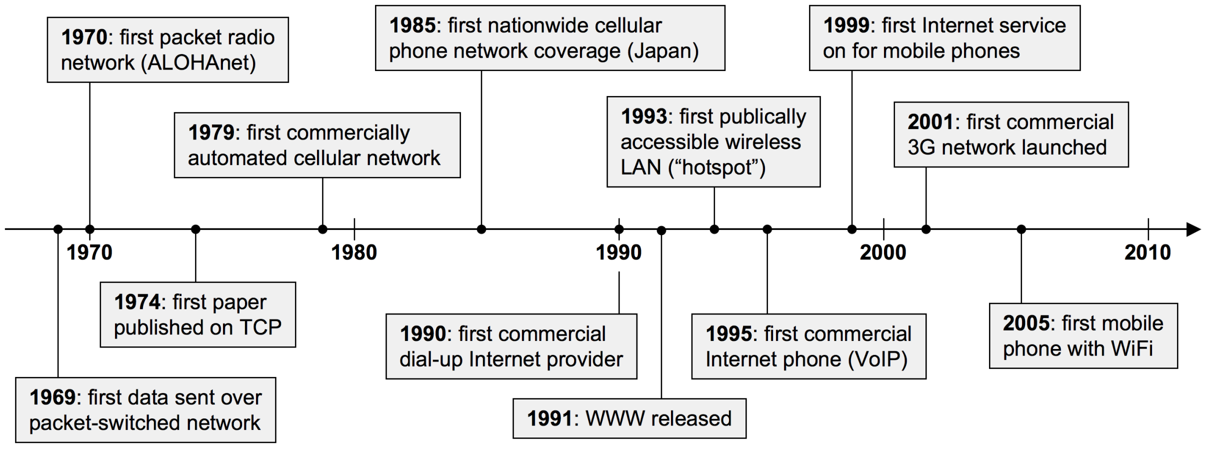

1970: first packet radio network (ALOHAnet)
1979: first commercially automated cellular network
1985: first nationwide cellular phone network coverage (Japan)
1993: first publically accessible wireless LAN ("hotspot")
1999: first Internet service on for mobile phones
2001: first commercial 3G network launched
1970
1980
1990
2000
2010
1974: first paper published on TCP
1969: first data sent over packet-switched network
1990: first commercial dial-up Internet provider
1991: WWW released
1995: first commercial Internet phone (VoIP)
2005: first mobile phone with WiFi

Figure 1.1: *Milestones in the evolution of the Internet.*

The recent explosion of wired and wireless telecommunication infrastructure has enabled many new applications and sparked new research problems and engineering tasks.

One of the unsolved issues in the current Internet – the main topic of this thesis – is the goal of increasing the data transfer speed by simultaneously using multiple network interfaces at a user's end host. This objective is most interesting in cases when a computer has several, relatively slow (typically wireless), Internet connections, and each connection alone is unable to provide the required transfer speed for a resource-intensive application, such as high-quality multimedia streaming.

1.1 Multiple Network Interfaces

Communication devices equipped with multiple network interfaces are now commonplace. For instance, smartphones and laptops are often shipped with built-in network adapters of different wireless technologies, typically enabling them to connect to third-generation (3G) cellular data networks, such as High Speed Packet Access (HSPA), and wireless local area networks (WLAN). At the same time, wireless network coverage has become so widespread that mobile devices are often located in overlapping coverage areas of independent access networks. As illustrated in Figure 1.2, if a device is within range of multiple access networks that are bottlenecks on the path to a server, then applications may potentially benefit from increased performance by simultaneously transferring data over several Internet connections.

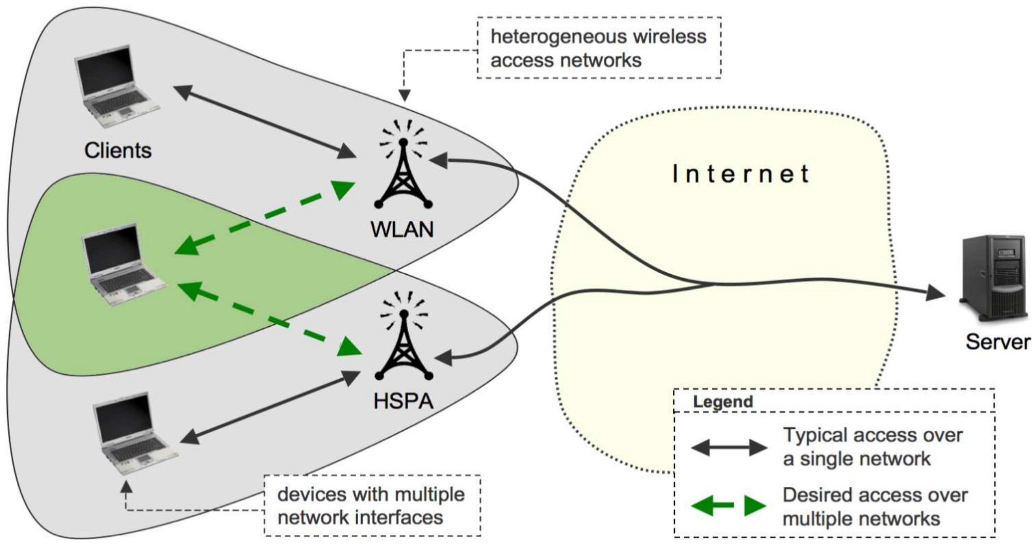

heterogeneous wireless access networks
Clients
WLAN
Internet
Server
HSPA
Legend
Typical access over a single network
Desired access over multiple networks
devices with multiple network interfaces

Figure 1.2: *An example scenario of simultaneous communication over two access networks with different wireless technologies.*

There are many scenarios in which supplementary interfaces could potentially be used to collaboratively increase the overall performance of an application. For example, while taking the bus home, a commuter might be watching the latest news clips on a smartphone using an expensive HSPA connection. When the bus passes through additional zones of open WLAN access, the application should be able to use the newly detected network to either collaboratively enhance the data throughput, or to reduce the user's phone bill by briefly switching to the free connectivity. Similarly, many homes are connected to the Internet by wired, high-speed connections, but use a wireless access point to distribute connectivity to mobile and stationary computers located in several rooms. If the distance to the access point is large and several walls are obstructing the wireless channel, even a high-speed WLAN may degrade, get unstable, and provide unsatisfactory transfer speeds for applications that require stable network conditions, such as video conferencing. In such cases, it would be very convenient for a user to establish an additional 3G network connection to improve the communication robustness.

1.2 Problem Statement and Scope

In reality, today's Internet users are unable to conveniently manage multiple network connections and to aggregate their data rate for improving the performance of applications. Even if multiple interfaces are successfully connected to the Internet, the operating system typically decides on a single default interface to which all flows of traffic get bound. This technical restriction is based on the fact that the majority of current Internet traffic is sent using the Transmission Control Protocol (TCP) [136] and the User Datagram Protocol (UDP) [134], which became the norm in the early 1980s and still today do not support multiple IP addresses per end-to-end session. A more research-relevant reason is that path heterogeneity poses significant challenges for efficient aggregation, even when alternative communication protocols and solutions are used that natively support multiple IP addresses per endpoint.

Deployment Challenges

In the forty years since the first Internet protocols were developed, their fundamental operation remained unchanged. According to Borgnat et al. [27], the two protocols TCP and UDP conveyed more than 90% of the packets through the Internet in the years 2001 to 2008. With a traffic share of more than 70%, TCP holds its position as the dominating network protocol for reliable data transmission through the Internet, and there is no evidence for a change in TCP's popularity in the near future. While TCP experienced numerous challenges over the following decades to adapt to increasing network capacity and new types of networks, such as wireless communication, establishing a TCP connection still involves exactly one IP address per endpoint. This makes it difficult to concurrently transfer data over multiple network interfaces.

Alternative transport protocols, such as the Stream Control Transmission Protocol (SCTP) [154] or recent developments of a Multipath TCP (MPTCP) [56], are attempts to fill this gap. However, they require severe changes at both ends of a connection and carry only a negligible amount of the global data traffic. Therefore, many other ways have been explored to provide TCP functionality over multiple network interfaces, for example by modifying TCP itself, by introducing specialized network-layer proxies, or by establishing multiple concurrent TCP connections on the application layer.

None of these solutions have yet been widely deployed, either because they require unacceptable changes to operating systems and standardized protocols, and/or because the proposed mechanisms are not verified to function in arbitrarily heterogeneous network scenarios that may additionally include Network Address Translation (NAT), firewalls, and other technical restrictions. It may take years for an entirely new protocol to come to widespread use, because a transition requires sufficient trust about correct protocol operation and it may introduce costly updates to existing software.

Network Heterogeneity and Dynamics

Apart from practical deployment challenges, the main difficulty of efficiently combining multiple interfaces is posed by the heterogeneity of different network paths. Different transmission technologies can cause significant differences in speed, latency and packet loss, leading to various problems in aggregating their capacity. If a sender has the choice of multiple paths, it must decide for each data packet over which path to send it. Therefore, an efficient scheduling technique is required to distribute data in proportion to the available capacity on each path, while considering potential differences in delay and loss.

When IP packets are sent over network paths with different round-trip latency, they are bound to arrive at the receiver out of order. Packet reordering may result in head-of-line blocking, increasing buffer requirements, potentially causing applications to lose information or to become bursty. For instance, multimedia players may frequently stall when video frames are received in the wrong order. Packet reordering is also known for its negative effects on the performance of transport protocols. For example, TCP interprets reordered packets as lost and might therefore unnecessarily retransmit data and reduce its transmission speed.

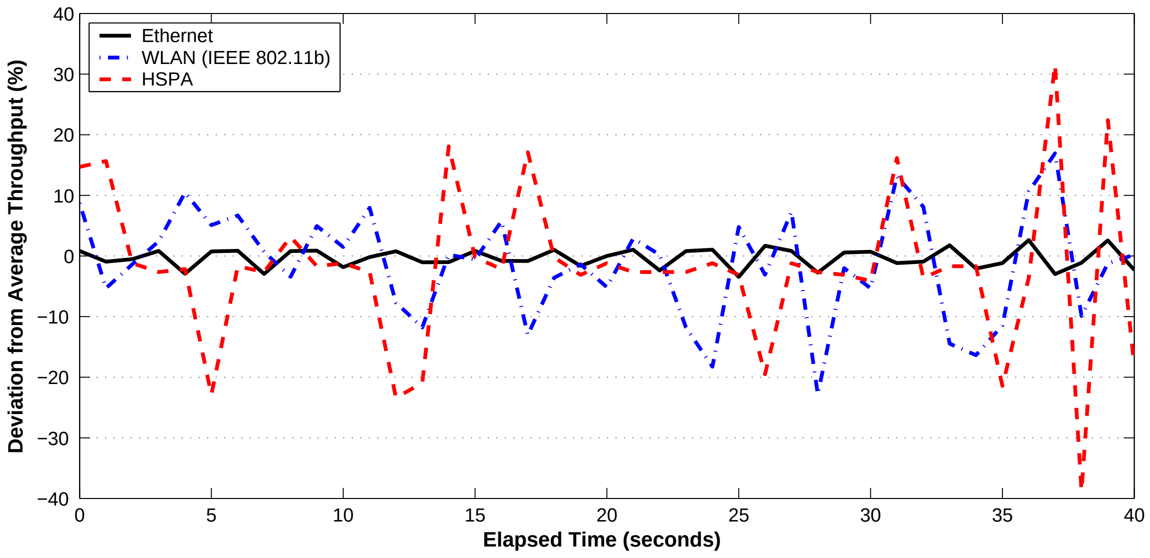


Figure 1.3: *Short-term throughput variance of different access technologies: Ethernet, Wireless LAN, and High-Speed Packet Access (HSPA). The results are based on downloading a 25.2 MB large file from a geographically local web server (7 to 9 IP hops).*

An additional challenge is caused by the instability of certain paths, especially through wireless access networks. While wired networks are usually stable and fairly predictable in terms of throughput, latency and loss, wireless links often exhibit severe changes in their characteristics. A typical example of throughput variances in wired and wireless networks is depicted in Figure 1.3. While the used Ethernet connection exhibits a stable throughput, both WLAN and HSPA fluctuate significantly over short periods of time. An important goal is to achieve an aggregated throughput that equals the sum of all links individually, regardless of variations.

Problem Scope

The fundamental problem of multipath aggregation is the difficulty of fulfilling a range of conflicting criteria (Section 2.1.1 lists them in detail). For example, it is difficult to schedule IP packets over heterogeneous paths without causing reordering and performance penalties at the transport layer. It is also hard to implement a generic multipath solution without modifying standardized protocols or changing third-party network equipment.

In this thesis, we explore solutions for aggregating multiple paths to end hosts and evaluate the multipath behavior of existing systems. For this, we use the throughput improvement of using multiple paths as the main performance metric. All our scenarios assume that each access network is the bottleneck link on the end-to-end path. Our research is largely focused on downlink data transfer but generally applicable to uplink scenarios as well.

Certain topics related to multipath aggregation are out of the scope of this thesis. For example, certain performance improvements of multipath aggregation other than improved download quality, such as improved resilience, faster handoff between multiple networks, or lower end-to-end latency, are not part of our analysis. In addition, although many experiments in this work are carried out using mobile devices, mobility itself is no integral part of the research. We consider mobility as part of the *path heterogeneity* factor, which equally affects the performance of individual paths in single- or multipath scenarios. However, providing vertical handover [109] and seamless mobility [174] is a very related, but orthogonal problem to multipath aggregation.

1.3 A Motivating Example

In order to verify the potential of using multiple network interfaces concurrently, we have set up a simple experiment that motivates our further work. This initial experiment of path aggregation combines BitTorrent [38], a peer-to-peer protocol for bulk data transfers, with equal-cost multipath routing (ECMP) [157, 69]. The BitTorrent protocol logically splits a large file into segments, which are downloaded independently and concurrently from various peers; and for each peer, a new TCP connection is established. ECMP is a network-layer technique used by routers for load balancing. In order to reduce IP packet reordering, routers generally use only a single outgoing interface for packets that belong to the same flow [111]. When a router encounters packets of an unknown end-to-end connection it must look up the best path for this connection. If multiple outgoing interfaces are evaluated to be equally suitable for forwarding, ECMP ensures that different flows are fairly spread over the outgoing paths which have an equal cost metric. However, ECMP always forwards IP packets of a given connection through the same interface.

In the Linux operating system, ECMP routing rules can be configured not only on routers, but also on end hosts. Thus, after connecting to a WLAN and an HSPA network and defining an equal-cost rule for both interfaces, new connections will automatically be established over either the WLAN or the HSPA network. For applications that open many connections to download a single file, such as BitTorrent, ECMP naturally leads to path aggregation by spreading the connections to different peers over the available interfaces.

Our initial experiment of path aggregation was carried out by downloading a total of 75 copies of a 700 MB large Ubuntu Linux distribution from the official source (reaching

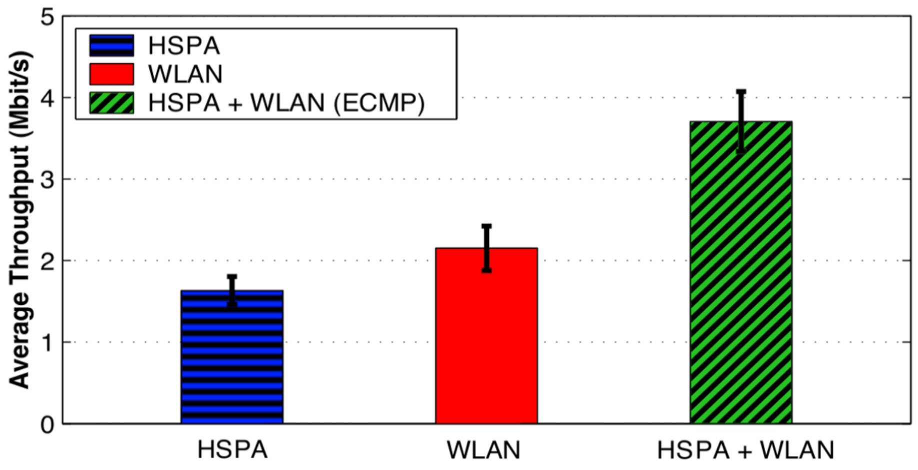


Figure 1.4: *Multipath aggregation performed through simultaneous HDPA and WLAN network access, based on a combination of the BitTorrent protocol and ECMP routing.*

an average of about 50 peers) and measuring the average transfer speed. As depicted in Figure 1.4, the transfer speeds of WLAN and HSPA are aggregated very efficiently, because all traffic is equally balanced in fair proportion to the path capacities. The results clearly show the potential of simultaneously utilizing multiple interfaces. The concurrent download of a file over heterogeneous WLAN and HSPA networks does not suffer from interference or other negative side-effects, and an aggregation benefit is achieved even under highly variable path conditions. In addition, this example solution requires only a minor configuration at the client and no changes to the network infrastructure and communication protocols are necessary.

It must be emphasized, however, that this solution completely ignores compatibility with existing transport protocols and applications. A solution that only applies to a single protocol (BitTorrent in this case) is clearly insufficient. In addition, this type of path aggregation only works for a rather rare kind of applications that are programmed to open a large number of concurrent transport connections. For the majority of network applications, which transfer data through a single transport connection, a purely ECMP-based approach is not applicable. In order to perform path aggregation for a single flow, a more sophisticated solution is required, which distributes individual data packets, not entire flows, over the available paths.

1.4 Thesis Contributions

This thesis explores the problem of multipath aggregation with the attempt of finding and evaluating solutions at various layers of the TCP/IP protocol stack. Figure 1.5 illustrates the problem of multipath aggregation as the task of implementing two abstract functions, (1) a *scheduling point*, where data packets are distributed over different paths, and (2) a *merging point*, where the data packets get reassembled and brought back in order. No matter at which protocol layer a solution operates, it must provide an implementation of at least one of the scheduling and merging black boxes.

Our exploration starts at the IP layer with a study of the packet reordering that is introduced by the use of multiple heterogeneous paths. It continues with an analysis of

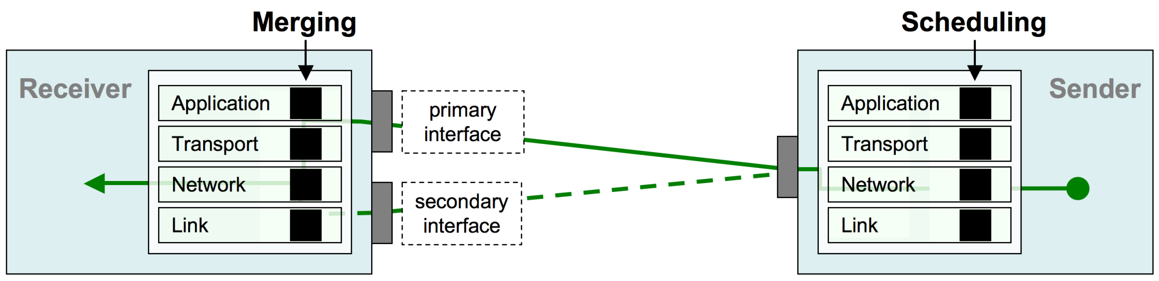


Figure 1.5: *Simplified scenario of multipath aggregation functionality implemented at various layers of the TCP/IP protocol stack.*

how IP packet reordering affects the transport layer. We also present an application-layer approach that comes very close to satisfying all the requirements and offers itself as a promising solution for a variety of multimedia streaming applications. The following paragraphs summarize the contributions of this thesis based on their target protocol layer.

Network-layer Contributions

In order to verify the impact of path heterogeneity, our network-layer study begins with measurements of access link characteristics and an analysis of the IP packet reordering that is caused by multipath forwarding. We show through practical experimentation that multipath reordering exceeds the packet reordering in the Internet by an extent that makes the typically used packet reordering metric useless. After realizing the strong effect of path heterogeneity on IP packet reordering, we propose a reorder-compensating scheduler. This sender-side scheduler is then incorporated into a network-layer adaptive multipath aggregation prototype called NAMA, which relies on path monitoring statistics to efficiently utilize multiple paths simultaneously. We observe that in scenarios with stable throughput and latency, IP packet reordering can be fully eliminated by equalizing the latency of the paths. Even under dynamic path conditions, our solution achieves close to optimal aggregation by adapting to arbitrary changes in bandwidth.

Transport-layer Contributions

Our main contribution on the transport layer is to provide a better understanding of how TCP is affected when IP packets are forwarded over multiple paths with different latency, bandwidth, and randomly introduced packet loss. For the purpose of a fair comparison of TCP's performance, we first propose a novel metric to determine the usability of an aggregation solution. Using this metric, we experimentally analyze TCP's multipath performance in both practical and emulated WLAN and HSPA networks. Furthermore, we evaluate the effects of numerous performance-enhancing mechanisms on TCP's robustness to multipath-inflicted packet reordering. Based on these results, we fine-tune Linux TCP to achieve an improved benefit from aggregating multiple paths. Our analysis suggests that no new protocol is required for achieving a considerable aggregation benefit. In scenarios with fairly homogeneous paths which induce only moderate packet reordering, Linux TCP reaches an aggregation benefit of 80% and more. Even under very heterogeneous conditions, Linux TCP is often able to adjust to the heavy packet reordering and to gain from multiple paths.

Application-layer Contributions

In the final part of our work, we introduce the idea of using existing mechanisms of the HTTP protocol to achieve efficient multipath aggregation. Although these mechanisms (i.e., range retrieval requests and request pipelining) were originally developed for different purposes, they enable a single file to be downloaded over multiple Internet connections simultaneously. However, the core idea of dividing a file into segments and transferring them over multiple interfaces causes new challenges and tradeoffs, such as finding the optimal segment size. We analyze these challenges and conclude our work with promising results from practical experiments with bulk data transfer and quality-adaptive video streaming. For example, by implementing our findings into the existing DAVVI [85] video streaming platform, the number of high-quality video segments drastically increases.

1.5 Thesis Outline

The remainder of this thesis is organized as follows:

Chapter 2, "*Background and Related Work*", provides an overview of existing research on multipath aggregation and emphasizes that packet reordering is an omnnipresent problem on any protocol layer. The phenomenon of packet reordering is then addressed in more detail by presenting existing literature on measurements of reordering in the Internet and several improved metrics are introduced.

Chapter 3, "*System Design and Method*", motivates the scientific method used for performance evaluation and describes the network configurations used in our experiments. This chapter also defines the *aggregation benefit*, a novel metric we developed for comparing the performance of multipath aggregation solutions with qualitative fairness, independent of path heterogeneity.

Chapter 4, "*Multiple Paths at the Network Layer*", presents an analysis of IP packet reordering inflicted by multipath forwarding and discusses methods for mitigating it. In this chapter, we also propose NAMA, a network-layer prototype capable of aggregating multiple paths while keeping packet reordering at a minimum. The content of this chapter was published partly in [44] and [91].

Chapter 5, "*Multiple Paths at the Transport Layer*", begins with a detailed explanation of the functionality of TCP, various congestion control algorithms, and other TCP mechanisms that are relevant in the presence of packet reordering. Afterward, the impact of packet reordering on TCP is evaluated in an emulated network and in a scenario with multiple heterogeneous wireless access networks.

Chapter 6, "*Multipath Aggregation at the Application Layer*", is based on four conference papers [45, 48, 88, 90] and describes an HTTP-based method for concurrently downloading multimedia content over multiple network interfaces. The main ideas introduced in this chapter were filed as a United States patent application [89].

Chapter 7, "*Conclusions*", summarizes our work and brings it into a future perspective.

Chapter 2

Background and Related Work

This chapter is divided into two parts. In the first part, existing approaches to the problem of multipath aggregation are presented, structured by the Internet protocol layer they operate at. In the second part, the problem of IP packet reordering is treated in more detail, because most approaches – no matter at which protocol layer they operate – have to deal with the consequences of out-of-order data delivery in one way or the other.

2.1 Multipath Aggregation

The research literature often refers to the problem of multipath aggregation as *bandwidth aggregation*, data/network *striping*, *load sharing*, *inverse multiplexing*, or with phrases that include the terms *multilink* or *multihoming*. While being largely interchangeable, the usage of these terms is often influenced by the protocol layer where a solution is implemented, and whether its main objective is increased throughput, enhanced resilience, or network handover. As our main target is an increased data transfer rate, the following sections have a strong emphasis on solutions with the same goal. We first introduce a set of criteria for evaluating multipath aggregation solutions and then evaluate existing approaches according to them.

2.1.1 Evaluation Criteria for Multipath Aggregation

The problem domains of deployment and path heterogeneity can be translated into evaluation criteria that we consider crucial for any complete multipath aggregation solution to fulfill. The following list defines these criteria with the keywords "*must*" and "*should*" used in the spirit of RFC 2119 [28]. The keyword "*must*" implies an absolute requirement of a solution, while the keyword "*should*" means that a requirement may be ignored if the full implications of choosing a different approach are known and reasonably evaluated.

C1 *The average throughput of a feasible multipath aggregation solution* must *be higher than the throughput of the single path with the highest average throughput.*

This criterion is fundamental when the main purpose of path aggregation is to increase the overall data throughput. Other benefits that potentially come along with path aggregation, such as achieving a lower average latency, enhanced connection robustness, or improved security, are out of the scope of this work.

C2 *A multipath aggregation solution* should *achieve an average throughput equal to the throughput sum of what the individual paths can achieve on their own.*

This criterion allows room for solutions that do not achieve a 100% efficient path aggregation at all times. It implies that achieving more throughput than a single path is still worthwhile under specific circumstances, for example in the presence of severe path heterogeneity.

C3 *A robust multipath aggregation solution* should *be able to cope with any kind of path heterogeneity, throughput fluctuations, and jitter.*

This criterion states that a solution is inappropriate if it cannot avoid scenarios in which it fails to achieve an aggregation benefit. Given the criterion C1, a solution must fall back to using a single path if path conditions are too harsh for a throughput improvement. A consequence of this criterion is that solutions should be able to deal with persistent reordering of data packets.

C4 *A complete multipath aggregation solution* should *support and be compatible with all existing transport protocols and applications.*

This criterion calls for solutions to be generic and universally applicable. Applications should not have to be completely redesigned to support path aggregation. Similarly, a solution that only supports TCP-based but not UDP-based solutions, or vice versa, may never gain wide-spread deployment.

C5 *A multipath aggregation solution* must not *introduce significant changes to third-party equipment and software.*

This criterion guarantees incremental deployability, allowing solutions to be rolled out in a timely manner, without causing significant cost and effort to service providers. Examples that violate this criterion are solutions that rely on changes in backbone routers or cellular base stations.

C6 *A multipath aggregation solution* should not *introduce significant changes to end hosts.*

This criterion targets a convenient installation of solutions on devices with multiple interfaces. Examples that violate this criterion include changes to device drivers or operating system software. However, in comparison to C5, this criterion is not absolute, because certain changes, such as client-side protocol modifications, might be acceptable if they do not force third-parties to upgrade their systems for compatibility.

C7 *A multipath aggregation solution* should not *introduce additional infrastructure.*

In order to avoid significant client- and server-side modification and to satisfy criteria C5 and C6, solutions may outsource their intelligence to performance enhancing proxies/middleboxes. However, this is generally discouraged due to potential scalability problems and violation of the end-to-end principle (which states that operations of communication systems should occur at the endpoints, not inside the network).

In the following sections, several contributions to multipath aggregation will be presented and evaluated on the basis of these criteria. We use this list of criteria to fairly judge the usefulness of solutions presented in the research literature. It will become clear that the fundamental problem of multipath aggregation is the seeming impossibility for a solution to completely fulfill all these criteria. Too many conflicts exist between them, so that tradeoffs must be made. For example, achieving an aggregation benefit (C2) under heterogeneous and variable path conditions (C3) is the core research problem at hand. At the same time, it is difficult to implement a generic solution (C4) in a non-intrusive way (C5 - C7). We thus use these criteria as a guideline for our own research, but we do not target a complete solution that satisfies them all.

2.1.2 Link-layer Bonding

At the link layer, multipath aggregation is typically called *bonding*, *trunking*, or *bundling*, because multiple physical channels of equal technology are bundled into a single logical channel. For instance, Ethernet bonding [13], also known as IEEE 802.3ad or EtherChannel, allows hosts to be connected to a switch by multiple Ethernet cables for increased throughput, or to multiple switches for higher availability. Since Ethernet speeds are typically higher than the requirements of single applications, Ethernet bonding typically prevents frame reordering by using the same physical Ethernet port per flow (flows are assigned to ports using hash values based on MAC or IP headers).

A standard similar to Ethernet bonding is Multilink PPP [152], which was designed for ISDN and aggregates links that use the PPP protocol [149]. Multilink PPP uses a 4-byte sequencing header for synchronization and for detecting lost fragments at the receiver. However, the protocol does not specify any scheduling strategy but suggests that data fragments could be distributed proportional to the links' transmission rates. Snoeren [153] proposes a scheme based on Multilink PPP for bundling multiple channels of the same wireless wide-area network (WWAN) technology, in which data is scheduled in proportion to short-term throughput averages. This scheme achieves link aggregation by splitting packets into fragments that are dynamically sized to equalize the transmission time, but relies on the assumption of small bandwidth $*$ delay products.

A wireless example of channel bonding is Atheros' non-standard *Super G*[14] technology, which doubles the 54 Mbit/s signaling rate of IEEE 802.11g networks to a total of 108 Mbit/s by simultaneously using non-overlapping channels. Another WLAN-based approach is described in [147], proposing a method for clients to split their traffic among all available access points. A method for striping across parallel channels in cellular data networks is described in [36].

When bundling channels of constant bit rate but of unequal technology, data packets might get out of order when delays are skewed on the channels or when the packets are of different size. A link-layer approach that operates transparent to the IP layer is described by Adiseshu et al. [5]. Introducing the Surplus Round Robin (SRR) scheme and a resequencing algorithm at the receiver, the authors achieve a consistent gain in throughput by aggregating an ATM and an Ethernet interface.

The main advantage of link-layer bonding is that the signaling rate of the channel is a known constant and can be used to efficiently reduce reordering. However, link-layer aggregation only works between two fixed systems and is not applicable in a general scenario of connecting to multiple Internet service providers. In other words, link-layer solutions interfer with several of the evaluation criteria introduced in Section 2.1.1. Most obviously, the criterion of allowing path heterogeneity (C3) is not fulfilled. In order to aggregate links to different access providers, changes to the network infrastructure are certainly needed, breaking the important criterion C5.

2.1.3 Network-layer Multipath Forwarding

At the network layer, the technique of using different end-to-end paths between a source and a destination is typically called *multipath routing* or *load balancing*. In order to forward IP traffic over multiple paths, a host or site must be *multihomed*. A multihomed system has multiple providers of independent IP address ranges. Multihomed hosts typically connect to independent Internet service prodivers via multiple network cards, while multihomed sites are “autonomously operating IP networks with more than one transit provider” [2]. In general, we can categorize network-layer multipath solutions into *flow-based* and *packet-based* forwarding. Flow-based solutions do not introduce IP packet reordering, but they prevent a single flow from aggregating multiple paths. We do not focus on flow-based solutions in this thesis, but provide a short description for the sake of completeness. In order to allow a single flow to benefit from the capacity of multiple paths simultaneously, a packet-based load balancing algorithm is necessary.

Flow-based Forwarding

Forwarding IP traffic over different paths can be beneficial for load balancing and to avoid bottlenecks. Routers and multihomed sites mainly implement methods such as Equal-Cost Multipath Routing (ECMP) [69] to increase their availability and their robustness against link failures. If an ECMP-enabled router is aware of multiple paths that have an equal distance to the destination (e.g., an equal number of IP hops), then the egress interface for the next hop is chosen based on a hash value calculated from the address and port fields. This ensures that packets of the same flow remain on a single end-to-end path.

Typically, flow-based load balancing is used to achieve an even distribution of a large number of traffic flows. However, some researchers address the idea of multipath routing and inducing on-the-fly route changes of end-to-end connections for increased flexibility and resilience. Kandula et al. [86] introduce a *flowlet*-based method to reduce IP packet reordering caused by abrupt route changes. The main idea behind flowlet-based load balancing is to allow route changes of an on-going flow if the inter-arrival delay between two consecutive packets is large enough for the flow to be rerouted without cause reordering.

Several protocol platforms have recently been developed for the purpose of multihoming and path reselection, including the Locator/Identifier Separation Protocol (LISP) [50], the Host Identity Protocol (HIP) [119] and Shim6 [125]. These approaches target a better scalability of the Internet and provide multihoming support for failover with the possibility of flow-based load balancing. They are all based on the idea of introducing an additional addressing layer to allow changing IP addresses on network interfaces, while keeping constant transport-layer identifiers. These protocols enable IP packet flows to dynamically change paths in mobile scenarios and in the presence of link failures, however, they do not enable simultaneous multipath transmission.

Packet-based Forwarding

A frequently used network-layer approach for aggregating multiple paths is to use tunneling mechanisms for transparently redirecting packets from the server or a proxy to all IP addresses at the client. Phatak et al. [130, 131] propose the use of IP-in-IP encapsulation to distribute IP packets across multiple network interfaces, which allows unmodified transport protocols and applications to enjoy the benefits of multipath aggregation. In addition, with the goal of minimizing TCP timeouts and spurious retransmissions, the authors devise a set of rules for tuning TCP over paths with different capacities. However, the authors' results are only valid under the assumption that the path latency is bandwidth-dominated (i.e., $RTT \approx packet\_size/bandwidth$), which is not generally true. In reality, the transfer time of a packet does not only depend on its size and a constant bandwidth, but is heavily influenced by queuing delays at intermediate routers.

Chebrolu et al. [34, 35] present a network-layer architecture for bandwidth aggregation with an infrastructure proxy as its core element. Aware of the IP addresses on multihomed clients, the proxy estimates packet delivery times to intelligently stripe packets with the goal of minimizing packet reordering at the receiver. However, the introduced *earliest delivery path first* scheduler assumes knowledge of base station queues, information that is generally not available to a proxy. Gurtov et al. [133] suggest a similar *fastest path first (FPF)* scheduler that is embedded in an architecture based on HIP. The authors have only run simulations and not tested their scheduler over paths with different latencies. Without clear evidence, the authors' claim that the FPF scheduler can fully eliminate packet reordering is questionable. As an alternative to HIP, the Shim6 protocol was enhanced to handle several paths simultaneously and published by Barré et al. [17] under the name Multipath LinShim6. However, the authors only present implementation details without any results about multipath aggregation.

Kim et al. [96, 97] introduce PRISM, a combination of a network-layer proxy and sender-side TCP modifications. The authors' main idea is not to avoid reordering at the receiver, but to manipulate the reverse-path flow of ACKs to appear in-order to the sender. This enables the server's send rate to be reduced only on packet loss, not when packets are delivered in the wrong order. Even though the authors show convincing results in both simulations and an emulated network, we question PRISM's deployability because its performance depends on a proxy *and* on a modified TCP congestion control algorithm.

In wireless mesh networks, hosts typically have routing functionalities too, and due to the broadcast nature of wireless environments, multipath routing is a very obvious approach for enhancing end-to-end communication. An overview of multipath routing in wireless ad-hoc and sensor networks is provided in [4]. However, since nodes in such networks typically have only a single transceiver, multiple paths are usually used for resilience and optimized routing, not for aggregation.

In contrast to flow-based forwarding, packet-based solutions typically satisfy our evaluation criteria C1 and C2 of achieving an improved throughput by using multiple paths simultaneously. Their main challenge is to fulfill criterion C3 of adapting their packet scheduling to paths with heterogeneous characteristics without introducing IP packet reordering. Their main strength lies in their transparency to upper layers and compatibility with virtually any existing application (C4). However, at least one of the requirements C5 to C7 is usually violated, because packet scheduling and reassembly must be introduced at either endpoint or on a proxy.

2.1.4 Transport-layer Multipath Protocols

Transport protocols build on the unreliable network layer functionality to transparently provide end-to-end connections to applications. Current standardized transport protocols do not support multipath aggregation and may experience performance degradation from IP packet reordering. Nevertheless, numerous attempts have been made to tune existing transport protocols for multipath capability, typically improvements to existing transport protocols, such as the Transmission Control Protocol (TCP) and the Stream Control Transmission Protocol (SCTP) [154].

SCTP Modifications

A common approach for aggregating the capacity of multiple paths on the transport layer is to extend SCTP's multihoming capability. SCTP supports failover between multiple IP addresses, but the standard protocol has no means for using them at the same time. Concurrent Multipath Transfer (CMT) is an SCTP extension developed by Iyengar et al. [75, 76] to allow simultaneous data transfer via multiple end-to-end paths. After identifing the problem that paths with different delay and capacity cause packet reordering and unnecessary fast retransmission, the authors suggest the use of virtual queues at the sender to keep a history of SACKs and to detect not only packet loss, but also packet reordering. The congestion window evolution of CMT comes close to having two separate SCTP connections.

Many researchers continued to build on the idea of CMT, including Huang et al. [71] with *WiMP-SCTP*, Liao et al. [103] with *cmpSCTP* and Abd El Al et al. [1] with *Load-Sharing SCTP (LS-SCTP)*. For example, LS-SCTP is a similar approach to CMT, which employs per-path congestion control and per-association flow control. The main difference to CMT is that a new set of sequence numbers and a new type of *load sharing data chunk* are introduced, requiring a change to SCTP's standard packet format.

As both CMT and LS-SCTP do not specifically address performance over wireless networks, several authors suggest using the Westwood and Westwood+ congestion control algoritms in combination with SCTP. The *Wireless SCTP Extension (WiSE)* introduced by Fracchia et al. [57] deals mainly with handover and uses path monitoring to continuously set the best path as primary, while both Fiore et al. [52] and Perotto et al. [129] introduce new SCTP-based transport protocols. They all make use of Westwood's inherent bandwidth estimation to send chunks efficiently over the available paths.

Budzisz et al. [30] propose the *mSCTP-CMT* protocol for concurrent multipath transfer during handover, in those transition moments when multiple networks are available at the same time. The mSCTP-CMT protocol is based on SCTP and its *Dynamic Address Reconfiguration (DAR)* extension described in RFC 5061 [155]. Dreibholz and Adhari et al. [42, 3] consider SCTP-based multipath aggregation over asymmetric paths and identify several problems related to buffering. For example, large differences in path latency cause out-of-order chunks, which prevent in-order chunks from being delivered to the application. The suggested solution against this problem is to retransmit chunks that cause too much blocking on a different path.

TCP Modifications

In contrast to SCTP, TCP is unaware of multihomed clients and allows only a single IP address per endpoint. However, the majority of current end-to-end applications run over TCP, which has sparked a lot of interest in enabling TCP to support the concurrent use of multiple end-to-end paths. Examples include pTCP [70], which diverges a data stream over multiple TCP connections, and mTCP [173], which provides increased fairness among flows by the use of an overlay network that guarantees interoperability with arbitrary IP networks and detects shared bottlenecks. Other approaches try to introduce less profound modifications to TCP by attempting to increase robustness against packet reordering, either by generating a more appropriate congestion response to reordered packets or by deferring spurious retransmissions. An extensive overview of challenges and solutions to packet reordering with TCP is presented in [102] and will be discussed in more detail in Section 2.2.

Tsao et al. [161] focus on the aggregation of paths with very large differences in capacity and present a set of mechanisms that jointly achieve *super-aggregation*, or in the authors' words: "a performance that is better than the sum of throughputs". The authors' idea of *ACK offloading* is to receive TCP data segments over a comparably high-speed WiFi and return ACKs over a low-speed 3G link. This reduces self-contention on the WiFi path and leads to a throughput increase higher than the sum of both paths. However, this is only valid under the assumption that (a) the 3G transmission speed is negligibly small compared to WiFi, and (b) that TCP congestion control is unable to efficiently utilize the full bandwidth.

Even though the first solutions were introduced years ago, multipath TCP has remained an active field in academia and is currently facing a high interest within the Internet Engineering Task Force (IETF). The most relevant contributions are now carried out by the Multipath TCP (MPTCP [73]) and Multiple Interfaces (MIF [72]) working groups. While the MIF working group is mainly concerned with client configuration and address selection policies, the MPTCP working group is targeting multipath extensions to TCP that are backward compatible to existing applications. As stated by its architectural guidelines in RFC 6182 [56], MPTCP must support both purposes of improved throughput and resilience, while being backward compatible to TCP and existing networks (including firewalls and other middleboxes). MPTCP's main idea consists of managing a transport connection as a set of TCP-like subflows, each with an independent congestion control instance and an own sequence number space. Connection-level sequence numbering is used for reassembly and for allowing segments to be retransmitted as part of another subflow. A recent implementation by Barré et al. [16] shows that MPTCP efficiently aggregates two 100 Mbit Ethernet links with up to 500 ms added delay on one of the links. In addition, Raiciu et al. [140] have shown with simulations that MPTCP improves load balancing and bottleneck avoidance in highly interconnected structures, such as data centers.

Multipath Modifications to Other Transport Protocols

Not only TCP and SCTP were extended for multipath support. One of the earliest methods for multipath aggregation was presented in 1996 by Allman et al. in [8]. The authors modify the File Transfer Protocol (FTP) [137] to establish several connections when a file transfer is initiated. The data to be transferred is divided into fixed-size segments and sent on the first connection that is able to transfer data (i.e., when the socket is not blocking). Another example is the Multi-flow Realtime Transport Protocol (MRTP) [110], an extension to the Realtime Transport Protocol (RTP) [146] that is specialized for real-time multimedia transfer over ad hoc networks. The protocol assumes an underlying multipath routing topology and a complementary TCP-variant for session/flow control.

Similar to network-layer approaches, transport-layer multipath aggregation solutions face the same challenge of scheduling data proportional to dynamic throughput changes of the involved paths. Many transport protocols work with sequence numbers, so that the challenge of packet reordering is more easily handled than at the network layer. However, only a few transport protocols are actually used in reality, which makes it very challenging to introduce and popularize a new modification or an altogether new protocol.

In general, transport-layer solutions are able to provide efficient aggregation under heterogeneous situations and are able to fulfill the evaluation criteria C1 to C3. However, most of the other criteria are typically violated, because transport-layer solutions require significant changes to both endpoints and are thus disadvantaged to get broadly accepted and quickly deployed. However, we believe that solutions like MPTCP may prove themselves in the long term, after successful standardization.

2.1.5 Session-layer Middleware

Session-layer striping is an approach that tries to avoid modifications to transport protocols by opening multiple TCP ”subflows” for each application-layer TCP flow. A common approach is to provide specialized middleware or virtual sockets, such as *Parallel Sockets (PSockets)* [151], that transparently partition application-layer data into multiple transport streams. While this has the advantage that existing applications do not need to be changed, the implementation of such middleware requires software modifications to both clients and servers. In [158], a purely client-based method is described, which assigns newly established transport-layer connections to one of the available interfaces and exploits traffic patterns of individual subscribers for optimal flow scheduling. However, this solution is purely flow-based and therefore fails to achieve bandwidth aggregation for single transport connections.

An architecture for session-layer striping is proposed by Habib et al. [64], which they claim supports throughput maximization. However, the authors fail to provide a performance evaluation and avoid a discussion of how a single flow can benefit from multiple paths. Driven by the motivation of streaming video from an ambulance vehicle to a hospital, Qureshi et al. [139] present Tavarua, a technically more mature middleware for providing network striping capabilities to applications with high demands on uplink throughput. The main limitation of Tavarua is its very application-specific implementation and the need to install the middleware at both endpoints.

The strong point of session-layer multipath aggregation solutions is that they introduce changes to neither transport protocols nor to applications (potentially satisfying the criteria C1 to C4 and C7). Their main drawback is that they either require significant client- and server-side modifications or fail to provide aggregation to individual transport flows (therefore violating the criteria C5 and C6).

2.1.6 Application-layer Programs

Application-layer striping is very similar to session-layer striping, but describes the method of directly adding support for multiple interfaces into a given application. A number of application-layer techniques for multipath aggregation have been presented in the literature. The common practice is that the application establishes multiple transport connections, binds them to different IP addresses, and distributes the data in proportion to the available path capacity over these connections. The core of application-layer solutions is to split a single file or byte stream into logical segments, which are concurrently transmitted over different paths.

Wang et al. [164] propose a scheme for live streaming over multiple wired TCP connections, referred to as Dynamic MPath-Streaming (DMP). No TCP modifications are performed, but multiple TCP senders fetch packets with video data from a server that queues and sends them over multiple paths. Packets are assigned additional sequence numbers to ensure correct reassembly at the receiver. Simulation results show that DMP-Streaming provides satisfactory performance when the aggregate achievable TCP throughput is 1.6 times the video bit rate. This *push-based* approach has two significant drawbacks. First, any push-based solution inherently requires server-side modifications and therefore lacks flexibility in deployment. To make matters worse, these modifications require a way for intelligently scheduling data in proportion to the data rate of the available paths. This may lead to inefficient aggregation, if data is not "pushed" in the optimal way. In mobile scenarios with typically large delays, generally low and rapidly varying throughput, a push-based TCP approach that adapts only by observing the TCP send buffer (such as Wang et al. [163] or Krasic's priority progress streaming [98]) would require several seconds before detecting a drop in the available bandwidth.

Pull-based approaches are typically used in scenarios with content that is mirrored on different servers. Rodriguez et al. [142] describe a parallel access scheme that allows users to download different parts of a single file from a variety of servers and reassembling them locally. The placement of multimedia file segments on replica proxies is also a commonly used technique to improve the response time of video-on-demand distribution systems [62]. For example, *Move Networks* [120] offers a delivery service to video-on-demand providers that imports content into a delivery system and distributes it to clients. Each client implements an HTTP-based pull approach that makes transparent use of replicated servers. An additional advantage of dividing multimedia files into segments is to allow smooth adaptation of the quality. For instance, Johansen et al. [85] divide a large video file into small self-contained clips that are encoded in different quality levels. A client can thus request the most suitable quality according to the observed path capacity. However, all these pull-based approaches assume that the throughput bottleneck is caused by overloaded content providers and fail to leverage the potential of hosts with multiple network interfaces.

In general, application-layer solutions have the advantage that the application is in full control of the striping decisions and that clients and servers can find an optimal way to collaborate. As we will show in Chapter 6, this makes it possible to achieve highly efficient path aggregation and to fulfill the evaluation criteria C1 to C3. The main drawback of application-layer solutions is that only one application is supported (violating C4), and that software changes are usually required at both endpoints (violating C5 and C6). Often, the server belongs to a third party, which poses a barrier to immediate deployment.

2.2 IP Packet Reordering

In the scope of packet-based path aggregation, packet reordering is an ever-present phenomenon. Every method that stripes data packets across multiple paths, be it at the link layer or at the application layer, faces the challenge of merging them back into sequence at a later point. Packet reordering is not only a result of packet-level multipath routing. As Leung et al. [102] point out, IP packet reordering is a property of network paths with many causes, including route fluttering (instable, oscillating routes), parallelism in high-speed routers and switches, and link-layer retransmissions. Since the research community became aware of the fact that packet reordering is "not a rare event along some network paths" [128], many attempts have been made to measure and quantify the typical amount of reordered packets in the Internet. Results from these studies often served as the motivation for network-layer algorithms that create less reordering (such as flowlet switching [150]) or for methods to enhance TCP's robustness against packet reordering.

The background on IP packet reordering measurements in the Internet is presented in Section 2.2.1, while the problem of defining an appropriate metric for quantifying packet reordering is discussed in Section 2.2.2.

2.2.1 Packet Reordering in the Internet

Some of the earliest measurements of packet reordering in the Internet were conducted in the 1990s by J. Mogul [115], Paxson et al. [128], and Bennett et al. [18]. In 1992, J. Mogul [115] traced thousands of TCP connections through the Internet and observed that "very few connections suffer from out-of-order delivery". In his experiments, about 3% of the connections contained a single out-of-order packet, while only 1% of the connections experienced two or more occurrences of packet reordering. A few years later, Paxson et al. [128] conducted two large-scale sets of experiments (E_1 and E_2) based on 20000 TCP connections between 35 sites. The authors observed that packet reordering can *sometimes* be very severe (as high as 36% of all packets reordered in a connection) – depending on the Internet path and the time of measurement. On average, 36% of the connections (2% of all packets) were affected by reordering in experiment E_1, and 12% of the connections (0.6% of all packets) were out-of-order in experiment E_2. In an attempt to prove that packet reordering is not at all a rare event, Bennett et al. [18] sent bursts of 56-byte ICMP ping packets to 140 different hosts and discovered that there is a 90% probability of the ping sequence to be reordered.

These initial, and fairly inconsistent, results quickly attracted more researchers to carry out larger experiments during the last decade and to emply more advanced measurement techniques. Rather than tracing connections at the endpoints, Jaiswal et al. [80, 81, 82] passively measured out-of-sequence packets at a single point in the backbone, allowing monitoring and analyzing millions of TCP connections between thousands of autonomous systems. Their studies report an average of 1.6% of the connections and 0.3% of all packets being affected by reordering. A similar study was performed by Wang et al. [165] in China, resulting in a tenfold higher packet reordering of 3.2%.

Instead of measuring reordering with a "variable bit rate" protocol like TCP, Gharai et al. [61] conducted experiments across the Internet backbone in the United States using UDP packet flows and different sending bit rates. Almost half (47%) of the connections experienced small amounts of reordering, however, with a moderate worst-case of 1.65% reordered packets in a single connection. Identifying an increased send rate as a main cause for out-of-sequence packets, and given the evolution of router technology, the authors foresee a future growth in packet reordering. Other measurement studies with UDP traffic were conducted by Loguinov et al. [105], Zhou et al. [175] and Tinta et al. [159], with all results summarized in Table 2.1.

Table 2.1: Measurement Studies of Packet Reordering in the Internet

Author	Year	Protocol	Affected connections	Affected packets
J. Mogul [115]	1992	TCP	4%	–
Paxson et al. [128]	1997	TCP	36% and 12%	2% and 0.6%
Bennett et al. [18]	1999	ICMP	90%	–
Jaiswal et al. [80]	2002	TCP	1.57%	0.16%
Loguinov et al. [105]	2002	UDP	9.5%	0.04%
Jaiswal et al. [81]	2003	TCP	1.84%	0.37%
Gharai et al. [61]	2004	UDP	47%	max. 1.65%
Zhou et al. [175]	2004	UDP	56%	5.8%
Wang et al. [165]	2004	TCP	5.6%	3.2%
Jaiswal et al. [82]	2007	TCP	1.58%	0.33%
Tinta et al. [159]	2009	UDP	10 – 84%	0.01% – 2.4%

Even though the current routing infrastructure attempts to maintain end-to-end connections on a single path through the Internet, this goal cannot always be accomplished (for example when links fail and routes are updated). As Table 2.1 suggests, most end-to-end paths do not experience significant packet reordering, but in certain cases, up to 5.8% of the packets in a connection may be affected by reordering, leading to potential performance deficits of transport-layer protocols like TCP.

Unlike these measurement studies on "single-path" packet reordering in the Internet, the focus of this thesis is *multipath* packet reordering, which is directly caused by the explicit use of multiple paths for a single transport connection. If IP packets are forwarded over different paths, more parallelism is introduced and much higher packet reordering is expected. In Chapter 4 we show through measurements that multipath packet reordering can be orders of magnitude larger than the numbers represented by Table 2.1. In Chapter 5, we evaluate how a modern TCP implementation reacts to multipath-inflicted IP packet reordering.

2.2.2 Packet Reordering Metrics

Most of the packet reordering measurement studies presented in the above Section 2.2.1 were carried out based on the metric by Paxson et al. [128], which identifies a packet arriving as reordered "if it was sent after the last non-reordered packet". In other words, packets that arrive *late*, and fail to monotonically increase the list of sequence numbers observed at the receiver, are counted as reordered. In an example packet sequence $S_1 = \{1, 6, 2, 3, 4, 5, 7\}$, the four packets $\{2, 3, 4, 5\}$ would therefore be counted as reordered, while only the single packet $\{2\}$ would be detected as reordered in sequence $S_2 = \{1, 3, 4, 5, 6, 2, 7\}$. Analogously, the number of *early* packets could be counted as reordered, resulting in one reordered packet in sequence S_1 and four reordered packet in sequence S_2.

A metric is usually chosen to give a measure of the (negative) implications of the packet reordering. Since reordering might affect transport protocols and application in different ways, there is a demand for a variety of metrics that depend on the context. In order to settle for a limited number of standard metrics, the IETF has specified a number of ways to measure the magnitude of IP packet reordering.

Ratio of Reordered Packets in a Stream

RFC 4737 [118] proposes several packet reordering metrics, including a TCP-like metric that considers a dupACK threshold, and metrics that focus on certain application-specific properties, such as the real-time lateness in seconds. As a basic, singleton metric, a packet is defined as *Type-P-Reordered* if its sequence number is smaller than the *next expected* sequence number, which is the highest observed sequence number plus one. This definition is equivalent to declaring late packets as reordered.

For expressing the percentage of reordered packets in a stream, RFC 4737 introduces the *Type-P-Reordered-Ratio-Stream* metric, denoted as M_{TPRRS}, which is defined as the number of Type-P-Reordered packets divided by the total number of packets received. With this definition, the packet sequences S_1 and S_2 would be evaluated to $M_{TPRRS}(S_1) = \frac{4}{7}$ and $M_{TPRRS}(S_2) = \frac{1}{7}$. If reordering occurs to only a very small fraction of all packets, M_{TPRRS} is an intuitive metric for getting a sense of the reordering. However, M_{TPRRS} does not contain any information about the severity of disorder in a sequence. In an extreme case of receiving a packet sequence, such as $S_3 = \{2, 1, 4, 3, 6, 5, ...\}$, where pairs of sequence numbers are “flipped”, M_{TPRRS} would be equal to 0.5, indicating that 50 % of all packets arrived late. In a scenario of round-robin, packet-based forwarding over paths with different delays, M_{TPRRS} is therefore naturally expected to approach 0.5.

Reorder Density and Entropy

The *reorder density* is an improved packet reordering metric able to convey the total magnitude of disorder in the received sequence. It was first introduced by Piratla et al. [132] and later specified in RFC 5236 [84]. To compute the reorder density M_{RD}, the sequence number of each packet is compared to the position in which it arrived at the destination. For each packet, this allows the calculation of a displacement within the sequence, with negative values indicating “early” packets and positive displacements indicating “late” packets. M_{RD} is calculated by accumulating the number of packets with a specific displacement. Figure 2.1 illustrates the reorder density of two example packet sequences. Both sequences result in an equal ratio of reordered packets ($M_{TPRRS} = 0.5$), but their reorder density is completely different.

Although M_{RD} provides an exact summary of the reordering in a packet sequence, it is less intuitive than M_{TPRRS} and not suitable for comparing the disorder of two independently measured sequences. A density function does not give a clear indication of the system (e.g., scheduling solutions) that performs the best. Thus, a singleton metric, called *reorder entropy*, was formulated by Ye et al. [169]. The metric M_{RE} expresses the total disorder of a packet sequence, including both the fraction of displaced packets as well as

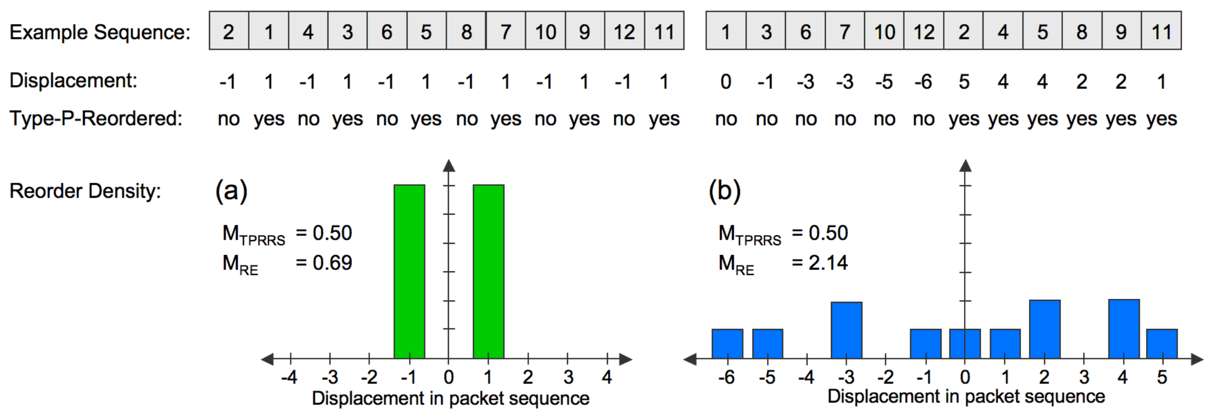


Figure 2.1: *Different packet sequences may contain an equal ratio of late packets M_{TPRRS}, but result in reorder densities (a) and (b), with different reorder entropy values M_{RE}.*

the degree to which the packets are displaced. Given a set D of all existing displacements in M_{RD}, the total reorder entropy M_{RE} is directly derived from M_{RD} as follows:

$$M_{RE} = -\sum_{i \in D}(M_{RD}(i) * ln(M_{RD}(i))) \tag{2.1}$$

The reorder entropy is zero if $M_{RD}(0) = 1$, i.e., if all packets have a displacement of zero. On the other hand, the reorder entropy is maximal, when each displacement occurs equally often. If packets are pairwise flipped, as in example (a) of Figure 2.1, the reorder entropy is much lower than in example (b), where most packets are displaced by several positions from their original location. In other words, the reorder entropy grows with increasing disorder in a packet sequence, even if the number of Type-P-Reordered packets remains the same.

Reorder Buffer Occupancy Density

Applications that require ordered data delivery often use a *reorder buffer* to recover from packet reordering by placing arriving packets with a sequence number greater than the currently expected into the buffer. Every out-of-order packet therefore increments the occupancy of the buffer by one. When missing packets arrive, the buffer occupancy decreases by the number of in-order packets that can be removed from the buffer for processing by the application. When a lot of reordering happens, the reorder buffer will be heavily occupied, while in a scenario of permanent in-order delivery, the reorder buffer occupancy would amount to zero.

In order to express the workload of such a reorder buffer, RFC 5236 [84] defines the *reorder buffer occupancy density* metric, M_{RBD}, as "the buffer occupancy frequencies normalized by the total number of non-duplicate packets". In other words, M_{RBD} is a distribution that indicates how frequently a reorder buffer stored a given number of packets. For example, $M_{RBD} = [0.5, 0.3, 0.2]$ indicates that 50% of the received packets arrived in order (zero buffer occupancy), 30% resulted in a buffer occupancy of one packet, and 20% of all packet arrivals caused two packets to be stored in the reorder buffer.

If more out-of-order packets arrive than fit into the buffer, a packet in the buffer has to be discarded, typically the one with the lowest sequence number (the most excessively delayed packet). From an application's point of view, such discarded packets are lost (another method would be to remove packets from the reorder buffer once a certain application-specific delay has expired). The *average buffer occupancy* can thus be used to summarize the overall performance as a singleton value. However, the average buffer occupancy does not give much insight on the number of packets discarded due to buffer overflows. The 95th (or 99th) percentile values of the buffer occupancy are more suitable to express the buffer size needed to guarantee at most 5% (or 1%) packet discards.

2.3 Discussion

Apart from an abundance of technical details, there are several things we have learned from the research literature:

The Potential of Various Protocol Layers for Multipath Aggregation

Numerous approaches to multipath aggregation exist that satisfy a subset of the evaluation criteria defined in the introductory Section 2.1.1), but no solution has yet been proposed that fulfills their entire scope. Evidently, there exists a tradeoff between a solutions' transparency (the number of applications supported without a need for modifications) and its deployability (the duration, cost and complexity of implementation and deployment). In the following list, we judge the potential of implementing a multipath aggregation solution at various layers of the TCP/IP protocol stack:

Link Layer We rule out link-layer approaches from our further work, because they are only suitable for bonding links of equal technology and unable to aggregate the bandwidth of independent access networks.

Network Layer Network-layer multipath forwarding is prone to cause IP packet reordering and to degrade the performance of higher-layer protocols and applications. However, they achieve a high degree of transparency and with the use of a proxy, modifications to endpoints can be avoided. In Chapter 4, we therefore pursue the goal of implementing an intelligent network-layer prototype that aims at providing efficient path aggregation while avoiding packet reordering.

Transport Layer Transport-layer multipath protocols require complex changes to both endpoints and are thus hindered from being standardized and deployed in foreseeable time. We therefore do not propose new multipath protocols or modifications. In our transport-layer analysis in Chapter 5, however, we explore ways of tuning easily accessible TCP parameters for improving its data throughput over multiple paths.

Session Layer We rule out session-layer approaches from our work, because they require both endpoints to be modified and typically entail minor changes to applications, too.

Application Layer Application-layer solutions are quick to implement, but they are tailored to a single application and completely lack transparency. However, due to the popularity of HTTP-based multimedia streaming and our idea to transparently add support for multiple interfaces to such applications, we pursue an application-layer study. In Chapter 6, we propose and analyze a combination of techniques for pull-based HTTP streaming solutions to efficiently aggregate multiple access networks.

Avoiding Wrong Assumptions

The majority of the related work that we presented in this chapter is based on simulations. Researchers working with simulators must make assumptions and simplifications to approximate and model reality. If this is not done very thoroughly, results are quickly perceived as unrealistic by reviewers and colleagues. In fact, we noticed many questionable assumptions made in simulation-based research. For example, researchers working on path aggregation tend to oversimplify the heterogeneity of network characteristics. Huang et al. [71] propose a solution to aggregate WLAN and 3G (UMTS) interfaces, whose latency heterogeneity certainly poses a great challenge. However, the authors configure their simulations with a 30 ms propagation latency for *both* technologies. In reality, the latency characteristics of these wireless technologies exhibit significant differences and major variations over time, as we will demonstrate in Section 3.2.

As another example, Fracchia et al. [57] assume that the one-way propagation delay is 40 ms on the fixed, wired part of the network, but negligible on the wireless channel. While a fairly stable latency over a backbone path can be justified, wireless propagation delays are by no means negligible. Wireless access networks are often the major component of a significant end-to-end latency, especially when the link layer uses its own retransmission scheme. Similarly, Phatak et al. [131] assume that the latency of a path is bandwidth-dominated (i.e., $RTT \approx packet\_size/bandwidth$) and that queuing delays in immediate routers are “probably similar for all paths”. While this is true to a certain degree, for paths through wireless networks it can easily be verified that the RTT is much more difficult to predict than Phatak et al. suggest.

Evaluating a solution under simplistic assumptions does not provide concrete clues on whether the solution would function properly when faced with path hetergeneity of real-world networks. It is therefore our goal to always validate our findings with practical experiments using truly heterogeneous network access, whenever possible. In addition, the observations motivate us to avoid simulations and to perform our own research based on the TCP/IP protocol stack of real operating systems.

Chapter 3

System Design and Method

This chapter describes our method for performance analysis of path aggregation in IP networks. First, Section 3.1 motivates our focus on practical experimentation and the use of a network emulator for performance analysis. Our practical testbed is introduced in Section 3.2, where already the first measurement results are shown as examples of the heterogeneity of wireless access networks. Section 3.3 goes into details on our testbed for network emulation and discusses the difficulty of properly sizing network queues.

In addition, we use this chapter to introduce two concepts that we have developed in the course of our research. First, Section 3.5 describes a generic scheduler that we apply in many of our experiments to distribute IP packets over multiple paths in a given proportion, while mitigating burstiness. Second, Section 3.6 presents a metric we have designed to provide a fair comparison of the performance of multipath aggregation under heterogeneous conditions.

3.1 Performance Analysis

In the field of computer networking there are many possible ways to analyze a system, to validate whether a solution operates as desired, and to compare a new algorithm to previously suggested approaches. The Internet is such a complex and large system that networking researchers often resort to simulators, such as the popular *ns-2* [126] network simulator, to validate their ideas. A simulator permits a convenient configuration of large and arbitrary network topologies. It allows the setup of networks that in reality might be impossible to get access to, composed of equipment that is too expensive to purchase and configure. A simulator is also a suitable facility for testing an algorithm under various network properties, such as a path's bandwidth, round-trip latency and loss rate. In reality, such characteristics cannot be controlled by researchers, because they depend on the physical infrastructure and on random traffic induced by other Internet users.

On the other hand, there are many pitfalls in using simulations, especially in research that involves unpredictable channels, such as wireless networks. As we discussed in Section 2.3, using a network simulator bears the risk of making false assumptions and to wrongly model the properties of practical networks. The dynamics and variability of wireless networks are difficult to approximate in simulation. An example of long-term throughput variations is illustrated in Figure 3.1, which shows the throughput dynamics of an HSPA network over a period of two weeks.

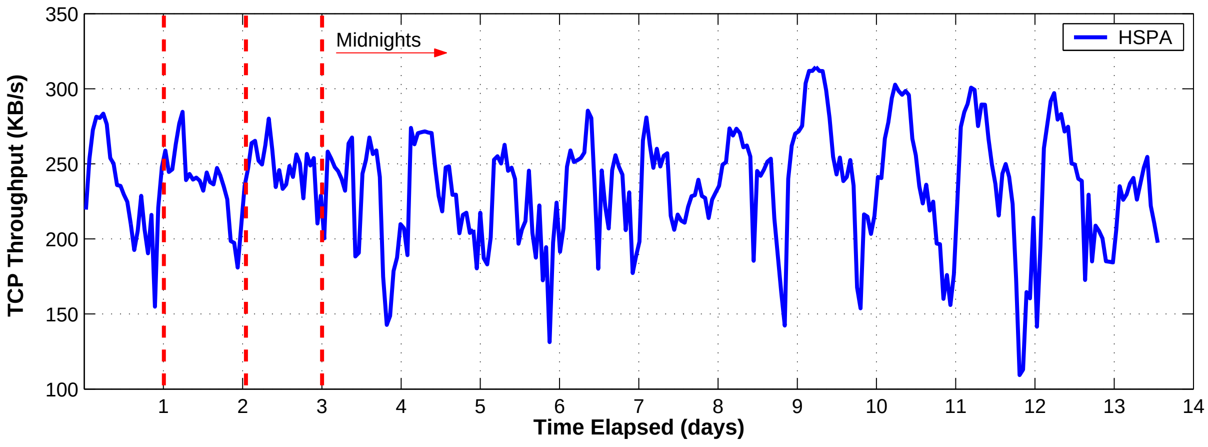


Figure 3.1: *HSPA throughput variations over a period of 14 days (these results were obtained by repeatedly downloading a 5 MB large file from our Web server).*

In addition, network simulators rarely include specialized link-layer protocols, such as HSPA, and do not implement the TCP/IP protocol stack in the exact same way as popular operating systems. We thus believe that simulators should only be used for experiments that are too expensive or intricate to be conducted with real network equipment. As the basic network scenario of multipath aggregation is fairly simple and allows testing with off-the-shelf components, we abstain from using simulations throughout this thesis.

The results presented in this thesis are exclusively based on prototype implementations with real hardware. In order to explore a large diversity of path characteristics, we use either emulated links or Internet subscriptions to different service providers. Our *practical testbed* introduced in Section 3.2 combines commercial WLAN and HSPA access networks, while our *network emulation testbed* described in Section 3.3 is a small Ethernet-based configuration that allows path properties to be arbitrarily configured, similar to a simulator but with real machines. Emulation may require some networking assumptions, such as queue sizes, but they are fewer than for simulations. In contrast to a simulation-based approach, network emulation allows the usage of an actual protocol stack that runs on modern operating systems, such as Linux.

The decision for practical implementation does not come without a cost, as it involves a steep learning curve and lacks a simulator's possibilities of repeatability. Dealing with real network components introduces time-consuming technical problems that might not be relevant to the scientific problem at hand, such as difficulties with device drivers, unexpected network outages while running overnight experiments, or being forced to postpone testing because the monthly download limit of a 3G data subscription is reached.

3.2 Practical Network Testbed

In this thesis, all practical experiments involving multiple wireless networks are carried out with a laptop that has a built-in Broadcom wireless adapter (IEEE 802.11a/b/g) and several USB ports for connecting additional 3G dongles. Except otherwise noted, the laptop is located stationary at our laboratory, where two different WLAN networks (referred to as WLAN_1 and WLAN_2) are openly accessible.

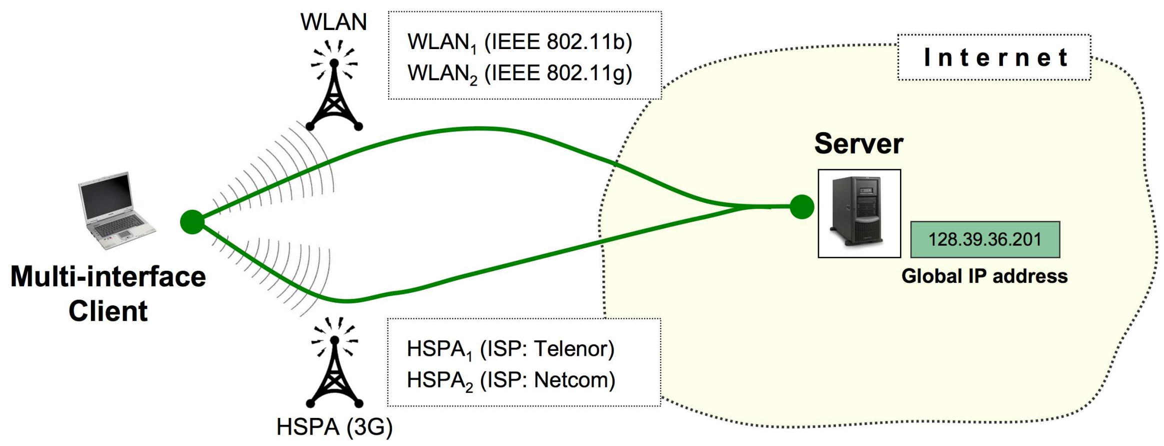


Figure 3.2: *Configuration of our practical network testbed.*

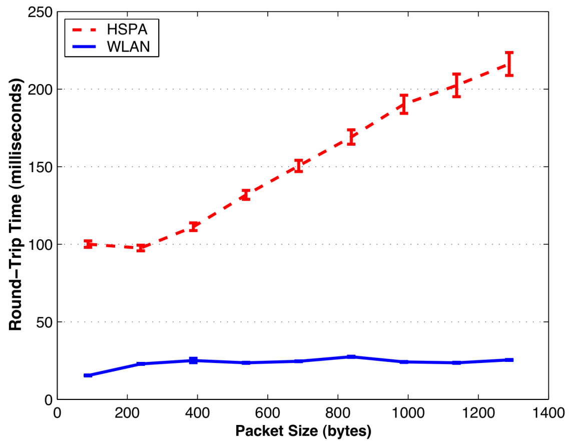
As shown in Figure 3.2, we have additional HSPA-based cellular data subscriptions to two Internet service providers (referred to as HSPA$_1$ and HSPA$_2$). We have chosen these two wireless technologies due to their widespread use in smart phones and laptops, because their comparable capacity makes them interesting candidates for aggregation, and because the combination of two different wireless networks appears as a challenging environment to validate new algorithms. The available networks give us the possibility to simultaneously connect to a total of four different networks from a single location. However, scenarios involving multiple WLANs are not included in any of our tests, because of severe interference problems when connecting to multiple WLANs on a single machine. All of our experiments are based on a client/server communication to a globally reachable server, which is also located in our laboratory.

Table 3.1 shows a summary of the path characteristics of our four test networks. The throughput numbers are approximate averages based on experience from many TCP-based data transfers. They provide a good speed comparison of the four test networks, but the throughput is highly variable over short and long periods of time. The round-trip times and loss rates were measured using ICMP Echo requests over a period of a full day, one *ping* per second. Both WLANs have similar latency characteristics, it takes about 10 milliseconds on average for packets to travel to the server and back, and roughly 2% of them get lost in transit. The two HSPA networks are also alike, but they differ from the WLANs by a generally lower loss rate and a significantly higher average RTT. In fact, as depicted in Figure 3.3, the average RTT through HSPA networks often jumps between two levels of roughly 80 ms and 300 ms.

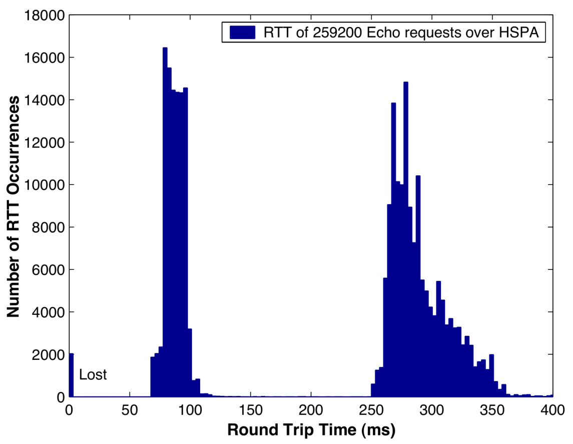
Table 3.1: Path Characteristics

Name	Network	Round-trip time (ms)				Packet	Typical
		min.	max.	mean	std.	loss	throughput
WLAN$_1$	IEEE 802.11b	2.2	5010	10.1	61.2	2.7 %	600 KB/s
WLAN$_2$	IEEE 802.11g	3.2	1800	11.2	59.0	1.5 %	1200 KB/s
HSPA$_1$	Telenor	67.5	1350	210	106	0.8 %	300 KB/s
HSPA$_2$	Netcom	69.0	3430	158	41.2	0.0 %	460 KB/s

The numbers in Table 3.1 do not account for the dependency of the RTT on packet size, all the RTT measurements listed are based on small 64-byte packets. Figure 3.3(a) depicts the average round-trip times through the networks $WLAN_1$ and $HSPA_1$ with various packet sizes. The graphs also illustrate that the round-trip times over heterogeneous networks may experience a significant difference, no matter how the packet sizes are chosen. In addition, changing between header-only and full-size IP packets resulted in a small difference of about 10 ms over $WLAN_1$, but in a much larger difference of over 100 ms through the $HSPA_1$ network.



(a) The Effect of packet size on the RTT



(b) HSPA's bimodal RTT distribution

Figure 3.3: *Round-trip time measurements over our experimental networks $WLAN_1$ and $HSPA_1$ (a) using various packet sizes, and (b) plotted as a cumulative distribution function to emphasize HSPA's* bimodal *behavior.*

3.3 Network Emulation Testbed

A disadvantage of testing algorithms in practical networks is the difficulty of configuring path properties exactly as desired. For example, with a simulator, it is easy to set the throughput and RTT between the client and the server to a constant 400 KB/s and 150 ms, respectively. In real networks, however, the path characteristics fluctuate and differ at every instant, making it hard to measure an algorithm under fixed conditions. In order to evaluate the performance of multipath aggregation with stable levels of heterogeneity, we use *network emulation* as a compromise between using unpredictable wireless networks and purely deterministic simulations. The main idea behind network emulation is to use a high-speed wired network, throttle its throughput to a desired level and introduce additional latency and loss, in order to obtain the desired path characteristics.

We have implemented the network emulation setup shown in Figure 3.4, consisting of a dedicated sender and a receiver with multiple network interfaces. Both machines run Linux version 2.6.32 and are connected through a switch using 100 Mbit/s Ethernet links. When network emulation is disabled, the RTT between the two machines is roughly 0.2 ms, two orders of magnitude below the average RTTs in our practical wireless networks. At the receiving interfaces, network emulation allows an arbitrary configuration of the two different paths, in terms of data rate, latency and loss. The emulator operates in

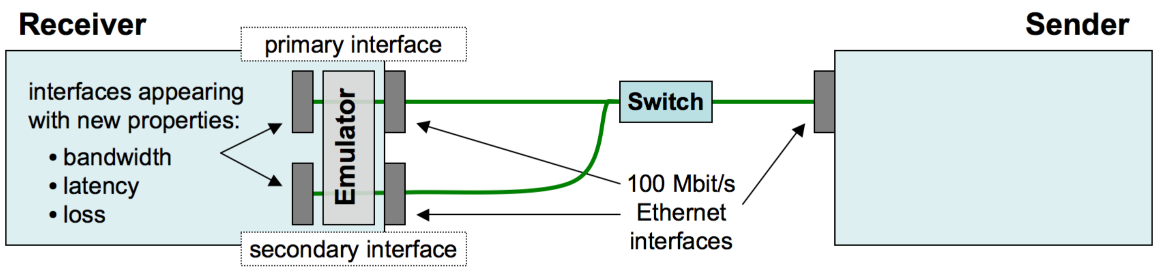


Figure 3.4: *Configuration of our network emulation testbed.*

both outgoing and incoming directions by the use of virtual ingress interfaces, called intermediate function blocks (IFB) in Linux. One-way latency and loss are introduced with the network emulation tool `netem`, while the data rate is limited by the traffic control tool `tc` using a hierarchical token bucket (`htb`) queuing discipline. Table 3.2 summarizes all adjustable parameters in our network emulation testbed.

Table 3.2: Network Emulation Parameters

Parameter name	Notation	Unit	Tool/command used
Primary bandwidth	BW_{pri}	KB/s	`tc` … `htb rate` BW_{pri}
Secondary bandwidth	BW_{sec}	KB/s	`tc` … `htb rate` BW_{sec}
Primary RTT	RTT_{pri}	ms	`netem delay` RTT_{pri}
Secondary RTT	RTT_{sec}	ms	`netem delay` RTT_{sec}
Primary loss probability	L_{pri}	%	`netem loss` L_{pri}
Secondary loss probability	L_{sec}	%	`netem loss` L_{sec}
Emulation queue length	Q	♯ packets	`netem limit` Q

The terms *primary* and *secondary* may carry a significant meaning depending on the protocol and the application used. With the goal of avoiding protocol modifications, a distinction between the paths must be made. For example, in our experiments with TCP in Chapter 5, we define the primary path as the one over which all ACKs return back to the sender. On the other hand, when using a stateless protocol like UDP, designating the interfaces with different names is typically not necessary.

The Problem of Emulating a Realistic RTT

After setting a network emulation parameter, we expect to measure exactly the configured value in a subsequent experiment. For the throughput limit and the loss rate, it can easily be verified that the emulator achieves the desired path properties. However, during bulk data transfers, the emulator always results in a variable and much higher RTT than the one specified. This is because the RTT depends on the size of the queue that the emulator uses for shaping the traffic to a given bandwidth. If packets enter the queue faster than they are allowed to leave, the queue starts accumulating packets, increasing the end-to-end delay with every additional packet waiting in the queue.

In order to prevent *buffer bloat* [60] (the problem of overly large RTTs due to excessive buffering), it is generally recommended to keep network queues short. However, configuring the queue too short has other negative side-effects. In the worst case of a tiny queue with space for only one packet, every packet that arrives a little too early would immediately be dropped, even if the rate limit is not yet reached. Obviously, there exists a tradeoff between choosing a large queue that introduces excessive latency and a small queue that leads to unnecessary packet loss. A general rule of thumb is to compromise between these conflicting goals by setting the queue length to a bandwidth-RTT product worth of packets [10]. Following this recommendation, the RTT can increase to double the configured value under heavy traffic (because it takes one RTT to drain a full queue). In reality, however, this rule of thumb seems not to be followed. When measuring the RTT through heavily loaded wireless access networks, we frequently observed RTTs in the order of seconds, although `ping` returns much smaller RTTs. Figure 3.5 illustrates the results of an extensive measurement based on a total of 400 TCP bulk transfers over the WLAN$_1$ and HSPA$_1$ networks. Each data transfer lasted for 60 seconds and TCP's average RTT estimation was extracted using the Linux `tcp_probe` module.

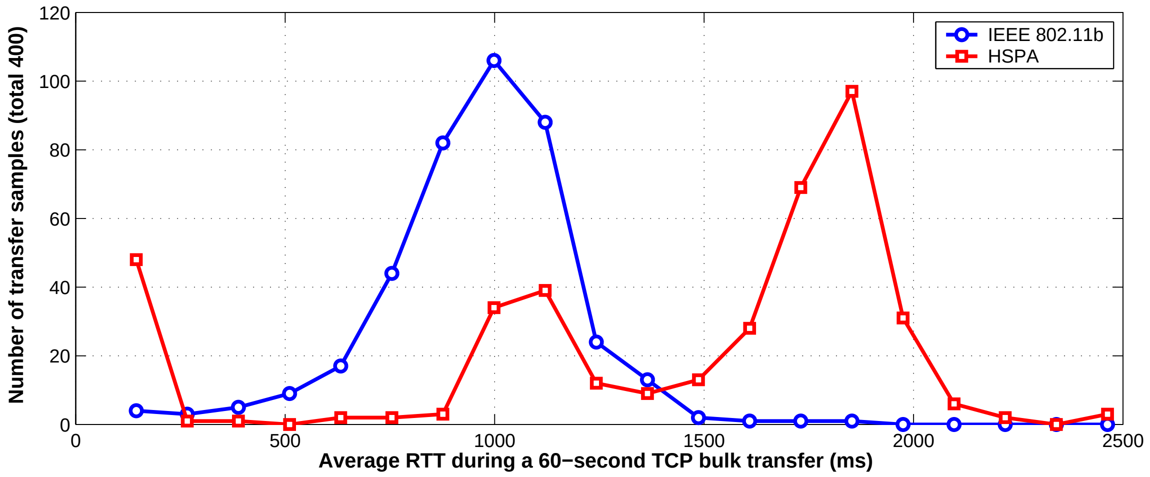


Figure 3.5: *RTT measurements during bulk data transfers over WLAN and HSPA.*

The numbers in Figure 3.5 seem to contradict the average RTT of 10 ms over WLAN and 212 ms over HSPA listed in Table 3.1. We attribute this large disparity to self-contention and unknown buffers in the WLAN and HSPA access networks. Since TCP attempts to fully utilize the available path capacity, and only reduces its send rate when packets are lost, it naturally causes router queues to be filled, resulting in a growing RTT. On the other hand, the results in Table 3.1 were collected by sending only a tiny 64-byte packet per second, far too little traffic to cause a router queue to grow, therefore leading to much lower average RTT measurements.

In conclusion, although we can implement existing rules for reasonable queue sizing in our network emulator, these rules do not seem to be followed in reality. Instead of always using a buffer size exactly representing a bandwidth-RTT product worth of (full-sized) packets, we thus introduce an additional parameter Q_{scale} to run tests with differently scaled queue lengths. A small constant margin of 5 packets is added to prevent unnecessary packet drops in case the queue length is too close to zero. After specifying a data rate BW, an RTT and a scale factor Q_{scale}, we calculate the queue length Q as follows:

$$Q = Q_{scale} * BW * (RTT/MSS) + 5 \tag{3.1}$$

3.4 Performance Measurement Tools

For our experiments, we use numerous tools that are common for measuring the performance of computer networks and applications. They are briefly described in the following list. More information to many of these commands can be found in the Linux manual [94].

`iperf` measures data throughput between hosts using either UDP or TCP (to transfer bulk data). When UDP is used, the sending bitrate must be specified. Details about this utility is available in [124].

`iptables` is an adminitration tool for IPv4 packet filtering and traffic rule specifications. We use it to translate IP addresses, for the purpose of diverging and merging traffic flows over multiple paths.

`netem` provides functionality to emulate network properties. We use it to emulate variable delay and to introduce random packet loss. It can also be used to duplicate and reorder packets. An outline of its design and usage examples are provided in [66].

`ping` measures the RTT between hosts by requesting an "echo" response datagram with the ICMP protocol.

`tc` is a utility for configuring network traffic control in the Linux kernel. We use `tc` in our network emulation testbed to limit data throughput to a fixed bandwidth.

`tcp_probe` is a Linux kernel module for testing TCP and obtaining detailed informating about the TCP's state (such as its estimated RTT and the size of the congestion window). Examples for using this utility are provided in [67].

`wget` is a tool to download resources from Web servers. It outputs some statistics about application-layer data throughput.

3.5 Design of a Generic Packet Scheduler

For many of our multipath aggregation experiments, we developed a generic scheduling scheme that efficiently distributes IP packets according to a pattern that can be dynamically updated. Unless otherwise stated, we use our *send vector* approach [44] for enabling both static and dynamic packet striping. The send vector contains a simple sequence of integer numbers that identify forwarding destinations, such as different paths, client interfaces, or IP addresses. For example, the send vector $V_{rr} = \{0, 1\}$ represents a round-robin striping of IP packets across two paths identified as 0 and 1.

In scenarios with paths of different transfer speeds, round-robin scheduling leads to inefficient aggregation, because one path will be underutilized while the other easily becomes overloaded and drops packets. When aggregating two example paths with an average throughput of 300 KB/s and 500 KB/s, respectively, a possible send vector could be constructed as $V_{bursty} = \{0, 0, 0, 1, 1, 1, 1, 1\}$. However, this send vector would result in bursts of packets, with likely negative effects on higher-layer protocols and applications. We therefore constructed an Algorithm 1 to generate send vectors that result in a smoother scheduling behavior. The algorithm takes the desired send vector length n as an input, as well as two[1] weights w_0 and w_1, which correspond to the throughput of the paths. For example, the weights $w_0 = 300$ and $w_1 = 500$, and a desired send vector of length $n = 8$, result in a weighted round-robin pattern $V_{wrr} = \{0, 1, 0, 1, 1, 0, 1, 1\}$.

Algorithm 1 generateSendVector(n, w_0, w_1)

```
Input: Vector length $n \in \mathbb{N}_{>0}$ and weights $w_0, w_1 \in \mathbb{R}_{\geq 0}$
Output: Send vector $V$ of length $n$
1: $V = zeros(n)$;                 {initialize $V$ with $n$ zeros}
2: $r = w_1/(w_0 + w_1)$;          {calculate weight ratio}
3: $r = round(r * n)/n$;           {adjust $r$ such that $r * n$ is an integer}
4: for $i = 1$ to $r * n$ do
5:     $V(\lceil i/r \rceil) = 1$;
6: end for
```

A binary send vector V of constant length n is easy to implement and requires little memory. When n is small, such as 64 or 128 used in our experiments, send vectors are often unable to accurately represent the ratio of two weights w_0 and w_1. Errors in approximating the true ratio translate directly into a loss in aggregation efficiency. However, the error is bounded and never exceeds $\frac{1}{2n}$.

Proof. The approximation error is the difference between the weight ratio $r_w = w_0/(w_0 + w_1)$ and the send vector's approximated ratio $r_V = m/n$, where m is the number of zeros in the vector and $n - m$ the number of ones ($n \geq m, m \in \mathbb{N}$). The approximation error Err is thus defined as:

$$Err = |r_w - r_V| \tag{3.2}$$

In the worst case, m will differ from its ideal value $r_w * n$ by $\pm\frac{1}{2}$. Thus, we can write $m = r_w * n \pm \frac{1}{2}$ and replace it into Equation 3.2:

$$Err = \left| r_w - \frac{m}{n} \right| = \left| r_w - \frac{r_w * n \pm \frac{1}{2}}{n} \right| \tag{3.3}$$

For an upper limit on the error, this simplifies to:

$$Err \leq \frac{1}{2n} \tag{3.4}$$

□

3.6 Design of a Novel Performance Metric for Multipath Aggregation

In the course of our research, we realized the lack of a suitable performance metric for multipath aggregation. There exists no measure for comparing the performance of different multipath aggregation scenarios. As an example, consider the PRISM solution [96], which, according to the authors, "yields 208% better performance than TCP in case of

[1]Details on how to create send vectors for more than two weights can be found in [44].

a large bandwidth disparity". Although this result sounds very impressive at first, the statement itself carries little information. On a closer look, the author's claim refers to the aggregation of a primary path with a bandwidth of 360 KB/s and a secondary path with a bandwidth of 1800 KB/s. Knowing these exact circumstances, achieving 1110 KB/s (208% more than 360 KB/s) appears rather humble compared to the maximum of 2160 KB/s that are actually possible. It could even be argued that PRISM is in fact highly inefficient, because it does not achieve close to what the secondary path can provide on its own, but only 51% of the maximum.

It is therefore necessary to define a metric that is able to express the performance of multipath aggregation scenarios on a linear scale. Preferably, this scale should have constant boundaries (such as -1 and 1), independent of the absolute bandwidth values of the paths involved. We consider multipath aggregation to have a *positive utility* if it outperforms the path with the highest average throughput. On the contrary, a *negative utility* is achieved if the aggregated throughput is smaller than what any single path could provide alone. Clearly, any scenario with a positive utility is ranked higher than any scenario with a negative utility. However, comparing the performance of multipath aggregation scenarios with either both positive or both negative utility, is not straightforward.

We define a multipath aggregation scenario S as a set of capacities $C_i > 0$ ($i \in \{1, ..., n\}$), representing the average performance of n individual communication paths, and a measured throughput x (with $0 \leq x \leq \sum C_i$) that was achieved by a solution. Introducing $\oplus$ as an operator for aggregation, we use the following notation to represent a multipath aggregation scenario S:

$$S = \{C_1 \oplus C_2 \oplus ... \oplus C_n \rightarrow x\}$$

For illustrating the problem of defining a fair performance metric, consider the following example scenarios S_A and S_B:

$$S_A = \{500\,KB/s \oplus 500\,KB/s \rightarrow 750\,KB/s\}$$

$$S_B = \{800\,KB/s \oplus 200\,KB/s \rightarrow 900\,KB/s\}$$

If scenario S_A achieves a throughput of 750 KB/s by aggregating two paths with 500 KB/s capacity, and scenario S_B is able to achieve 900 KB/s by combining capacities of 800 KB/s and 200 KB/s, then which solution is better? In other words, assuming that the same solution is used in scenarios S_A and S_B, which of the two scenarios is more challenging for an efficient aggregation? There are at least three convincing answers, giving three possible performance metrics for the benefit of performing aggregation:

1. Scenario S_B performs better than S_A, because 900 KB/s out of the maximum possible 1000 KB/s are achieved, a ratio of 90%. Scenario S_A only achieves 750 KB/s out of 1000 KB/s, a considerably smaller ratio of only 75%. Such an *aggregation ratio* $A_{ratio}(S)$ of a scenario S can be expressed as follows:

$$A_{ratio}(S) = \frac{x}{\sum_{i=1}^{n} C_i} \tag{3.5}$$

2. Scenario S_A performs better than S_B, because 250 KB/s are gained in addition to the highest single capacity, an improvement of 50%. In scenario S_B, the throughput is increased by only 100 KB/s over the capacity of the fastest communication path, a gain of merely 12.5%. This aspect of gained performance in a scenario S can be formalized as the *aggregation gain* $A_{gain}(S)$ over the highest capacity:

$$A_{gain}(S) = \frac{x - \max\limits_{i=1}^{n} C_i}{\max\limits_{i=1}^{n} C_i} \tag{3.6}$$

3. Both scenarios can be considered as equally efficient, because they both gain 50% of what possibly can be gained. In scenario S_A, 250 KB/s of 500 KB/s is gained, while scenario S_B gains 100 KB/s of the added 200 KB/s. In order to capture this aspect of efficiency, we define the *aggregation efficiency* $A_{eff}(S)$ of a scenario S as:

$$A_{eff}(S) = \frac{x - \max\limits_{i=1}^{n} C_i}{\sum\limits_{i=1}^{n} C_i - \max\limits_{i=1}^{n} C_i} \tag{3.7}$$

From these examples, it becomes evident that various measures for an objective comparison exist, but it is not obvious which measure is most suitable. Therefore, a set of requirements is necessary to verify whether one of the above aggregation measures is generally applicable. A well designed performance metric should be a monotonic function, allowing a ranking of different scenarios on a consistent scale – consistent in the sense that the span of the scale is independent of the number of paths n and the absolute value of their respective capacities C_i. We aim for a figure of merit which provides a sound basis for statistical evaluation. Its most elementary property must be that it can directly express whether aggregation is beneficial in a given scenario or whether using a single path would be preferable. In other words, the function should return 0 if and only if the throughput is equal to the largest capacity. For example, the following scenarios S_C and S_D should equally result in *zero utility*:

$$S_C = \{1000\,KB/s \oplus 1000\,KB/s \rightarrow 1000\,KB/s\}$$

$$S_D = \{21\,KB/s \oplus 12\,KB/s \oplus 83\,KB/s \oplus 44\,KB/s \oplus 65\,KB/s \rightarrow 83\,KB/s\}$$

In addition, there exists a distinct condition of a worst-case utility, i.e., when a solution fails to transfer any data at all (iff $x = 0$). This worst-case condition should lead to a constant utility value (e.g., normalized to -1), no matter the magnitude and heterogeneity of the capacities C_i. For instance, the following scenarios S_E and S_F should both result in a *worst-case utility* of -1:

$$S_E = \{1000\,KB/s \oplus 1000\,KB/s \rightarrow 0\,KB/s\}$$

$$S_F = \{21\,KB/s \oplus 12\,KB/s \oplus 83\,KB/s \oplus 44\,KB/s \oplus 65\,KB/s \rightarrow 0\,KB/s\}$$

Analogously, there exists a single condition for best-case aggregation performance, i.e., when a solution manages to perfectly aggregate the entire available capacity (iff $x = \sum C_i$). Any such scenario with optimal throughput should be ranked by the performance metric with the same value of maximum utility (e.g., normalized to 1). For example, the following two scenarios S_G and S_H should equally result in a *best-case utility* of 1:

$$S_G = \{1000\,KB/s \oplus 1000\,KB/s \rightarrow 2000\,KB/s\}$$

$$S_H = \{10\,KB/s \oplus 20\,KB/s \oplus 30\,KB/s \oplus 40\,KB/s \rightarrow 100\,KB/s\}$$

A meaningful figure of merit for multipath aggregation, let it be called the *aggregation benefit* $A_{ben}(S)$, should be a linear function. Any change in benefit, ∂A_{ben}, induced by a change in throughput, ∂x, of a specific scenario S should be independent of the throughput x ($\partial A_{ben}/\partial x$ is independent of x). However, for scenarios with heterogeneous capacities, there exists no linear function through the three points that define best-case, worst-case, and zero utility. Figure 3.6 visualizes this dilemma by means of three example scenarios. Each scenario has a minimum aggregated throughput of 0 and a maximum of 1000 KB/s, but the point of zero utility is different in each case. Only for the homogeneous example S_B it is possible to draw a linear slope through the three given fixed points. We therefore opt for a weaker linearity requirement that $\partial A_{ben}/\partial x$ must be piecewise constant for positive and negative benefits.

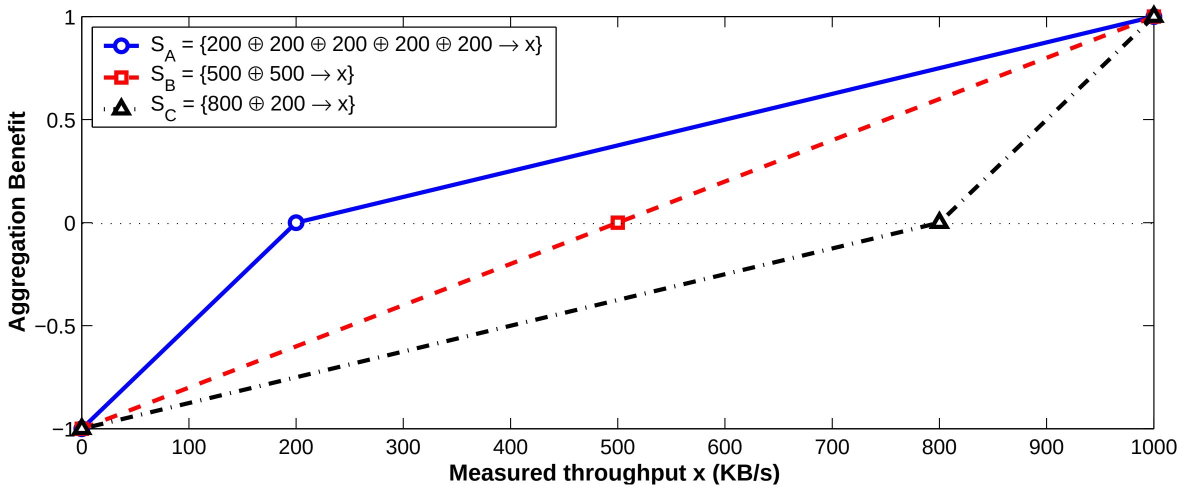


Figure 3.6: *The aggregation benefit of three example multipath scenarios.*

In summary, given a scenario $S = \{C_1 \oplus C_2 \oplus ... \rightarrow x\}$, the aggregation benefit $A_{ben}(S)$ must fulfill the following four requirements:

R_1 $A_{ben}(S = \{C_1 \oplus C_2 \oplus ... \rightarrow \max C_i\}) = 0$. The figure of merit $A_{ben}(S)$ must always return zero if neither a gain nor a loss in performance is achieved (when the measured throughput x is equal to the largest capacity $\max C_i$).

R_2 $A_{ben}(S = \{C_1 \oplus C_2 \oplus ... \rightarrow 0\}) = A_{worst}$. The figure of merit $A_{ben}(S)$ must always return a constant value A_{worst} for worst-case scenarios (when the measured throughput x is zero, meaning that no data was transfered at all). Without loss of generality, we chose $A_{best} = 1$.

R_3 $A_{ben}(S = \{C_1 \oplus C_2 \oplus ... \to \sum C_i\}) = A_{best}$. The figure of merit $A_{ben}(S)$ must always return a constant value A_{best} for all best-case scenarios (when the measured throughput x is equal to the sum of all capacities $\sum C_i$). We assign equal weights to best case and worst case scenarios, i.e., $A_{best} = -A_{worst} = 1$.

R_4 Linearity: $\partial A_{ben}/\partial x$ must be *piecewise* constant for positive and negative benefits.

Given these requirements, it is possible to check whether the above mentioned performance metrics for multipath aggregation (i.e., the aggregation ratio $A_{ratio}(S)$, the aggregation gain $A_{gain}(S)$ and the aggregation efficiency $A_{eff}(S)$) are appropriate. As shown in Table 3.3, every suggested performance metric fails to fulfill one of the requirements of our target function $A_{ben}(S)$.

Table 3.3: Comparison of Performance Metrics for Multipath Aggregation

Requirement	$A_{ratio}(S)$	$A_{gain}(S)$	$A_{eff}(S)$	$A_{ben}(S)$
R_1: $x = \max C_i \to 0$	$\frac{\max C_i}{\sum C_i}$	0	0	0
R_2: $x = 0 \to$ -1	0	-1	$\frac{-\max C_i}{\sum C_i - \max C_i}$	-1
R_3: $x = \sum Ci \to 1$	1	$\frac{\sum C_i - \max C_i}{\max C_i}$	1	1
R_4: monotony	✓	✓	✓	✓

A performance metric that satisfies all four requirements can be constructed by a piecewise combination of the aggregation gain and the aggregation efficiency. As the aggregation gain defines a linear scale between -1 and 0 for $x < \max C_i$, and the aggregation efficiency provides a linear scale from 0 to 1 for a throughput measurement $x > \max C_i$, they can be combined to a single performance metric. The *aggregation benefit* $A_{ben}(S)$ is therefore defined as follows:

$$A_{ben}(S) = \begin{cases} \frac{x - \max_{i=1}^{n} C_i}{\sum_{i=1}^{n} C_i - \max_{i=1}^{n} C_i} & \text{if } x \geq \max_{i=1}^{n} C_i \\ \frac{x - \max_{i=1}^{n} C_i}{\max_{i=1}^{n} C_i} & \text{if } x < \max_{i=1}^{n} C_i. \end{cases} \tag{3.8}$$

or in short:

$$A_{ben}(S) = \begin{cases} A_{eff}(S) & \text{if } x \geq \max_{i=1}^{n} C_i \\ A_{gain}(S) & \text{if } x < \max_{i=1}^{n} C_i. \end{cases} \tag{3.9}$$

The aggregation benefit $A_{ben}(S)$ indicates, on a scale from -1 to 1, how beneficial a multipath solution is compared to the exclusive use of a single path. For any two measurements carried out in different network scenarios S_1 and S_2, their figures of merit $A_{ben}(S_1) < A_{ben}(S_2)$ give a notion of which configuration has a higher benefit for multipath aggregation. If two scenarios S_1 and S_2 result in aggregation benefits of opposite sign, i.e., if $x_0 < \max C_i$ and $x_1 > \max C_i$, then it might not seem sensible to make a quantitative comparison between the two different quantities $A_{gain}(S)$ and $A_{eff}(S)$. However, there is no single linear scale that fulfills all of the above-mentioned requirements. Thus, we have to compromise on the intuitive clarity of the measure $A_{ben}(S)$ to provide a sound basis for statistical evaluation.

Chapter 4

Multiple Paths at the Network Layer

Even though the IP layer is meant to act fully transparently to upper layers without guarantees about ordered delivery, algorithms in routers and switches are often designed to preserve order in IP packet flows. As an example guideline, RFC 3246 [40] states that "packets belonging to a single microflow [...] passing through a device *should not* experience re-ordering in normal operation of the device". The reason behind such a code of best practice is that the performance of applications and transport protocols (with TCP as the most prominent example) can be negatively affected when IP packets get out of order. In the context of path aggregation, however, the common practice of flow-based forwarding poses a big restriction. As motivated in the introductory Section 1.3, only applications that simultaneously open many transport connections can naturally achieve multipath aggregation through flow-based forwarding. The only way to allow a single flow of IP packets to utilize the capacity of multiple access networks, is to diverge the packets over the different networks (i.e., packet-based forwarding).

This chapter intentionally breaks the common practice of flow-based forwarding to get a sense of how much IP packet reordering is caused by packet-based forwarding. To this end, Section 4.1 discusses experimental results of multipath-inflicted packet reordering using a variety of packet reordering metrics. Afterward, Section 4.2 presents a description of our NAMA prototype, which achieves transparent multipath aggregation and additionally reduces the IP packet reordering observed by the receiver.

4.1 Multipath Packet Reordering

In this section, we present experiments to measure the magnitude of *multipath packet reordering* induced by scheduling packets over multiple wireless networks that substantially differ in throughput and latency. Forwarding a sequence of IP packets over heterogeneous paths is likely to cause IP packet reordering far beyond the *single-path reordering* of 0% to 5.8% reported in the literature ($\rightarrow$ Section 2.2.1). Especially in wireless networks, the amount of reordering may be additionally affected by unpredictable parameters, such as varying signal quality, interference, queuing at base stations, and contention caused by other users on the same channel. Our experiments pursue several goals:

1. It is important for later experiments to get a sense of the amount of multipath reordering that is introduced by packet-based forwarding over highly heterogeneous access networks.

2. Although multipath reordering is foreseen to significantly exceed single-path reordering, the current research literature does not include examples or concrete numbers. Our experiments may serve other researchers as a motivation and point of reference.

3. Our experiments test the usefulness of improved packet reordering metrics that have emerged in the recent literature when applied to multipath scenarios.

4.1.1 Measurement Results

We perform our multipath packet reodering experiments in a practical testbed configured as shown in Figure 4.1, in which a client receives a packet sequence from the server over the HSPA$_1$ and WLAN$_1$ networks. In some cases, the WLAN$_1$ is exchanged by WLAN$_2$. The characteristics of these networks are the same as presented in Chapter 3.

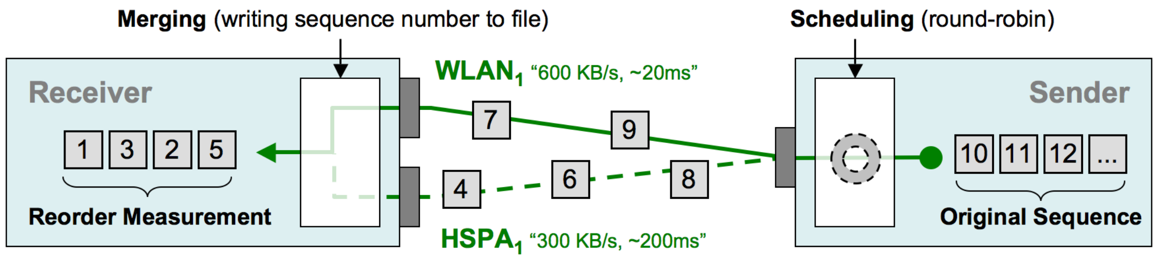


Figure 4.1: *Experimental setup for measuring IP packet reordering over multiple networks.*

At the beginning of each data transfer, the client informs the server about its network interface addresses and about all other relevant parameters used for the intended test run (i.e., data rate, test duration, packet size, type of scheduling, etc.). Upon receipt of all necessary information, the server sends an ordered packet sequence at a specified constant bitrate to the given client addresses. A scheduler at the server decides to which client address each packet will be sent, by splitting the UDP packet sequence over the two available paths. If not otherwise specified, round-robin scheduling is used. The client logs the sequence numbers in the order received and calculates different packet reordering metrics. The data transfer is carried out on the application layer based on the User Datagram Protocol (UDP) and a total packet size of 1480 bytes. UDP does not add any intelligence on top of IP and can thus be considered a suitable protocol for measuring IP packet reordering perceived by the transport and application layers.

Packet Reordering in Terms of Lateness (M_{TPRRS})

Figure 4.2 depicts our measurements of the IP packet reordering expressed by the metric M_{TPRRS} (Type-P-Reordered-Ratio-Stream), which represents the percentage of packets in a stream that arrive later than expected. The two curves were obtained by scheduling packets round-robin over HSPA$_1$ and WLAN$_1$, and over HSPA$_1$ and WLAN$_2$, respectively. The figure illustrates that almost no packet reordering is observed when the send rate is low. However, as the send rate increases, more and more packets are received out of order, until at some point (around 0.5 Mbit/s), close to 50% of all packets are reordered. In addition, the plots show that the same magnitude of reordering is observed, no matter which of the two WLANs is used in combination with the HSPA$_1$ network. This can be explained by the very homogeneous latency characteristics of WLAN_1 and WLAN_2. During the experiment, the average one-way delay on the application layer was measured as 80 ms over HSPA_1, while WLAN_1 and WLAN_2 resulted in a lower, but very similar one-way delay of 42 ms and 44 ms, respectively.

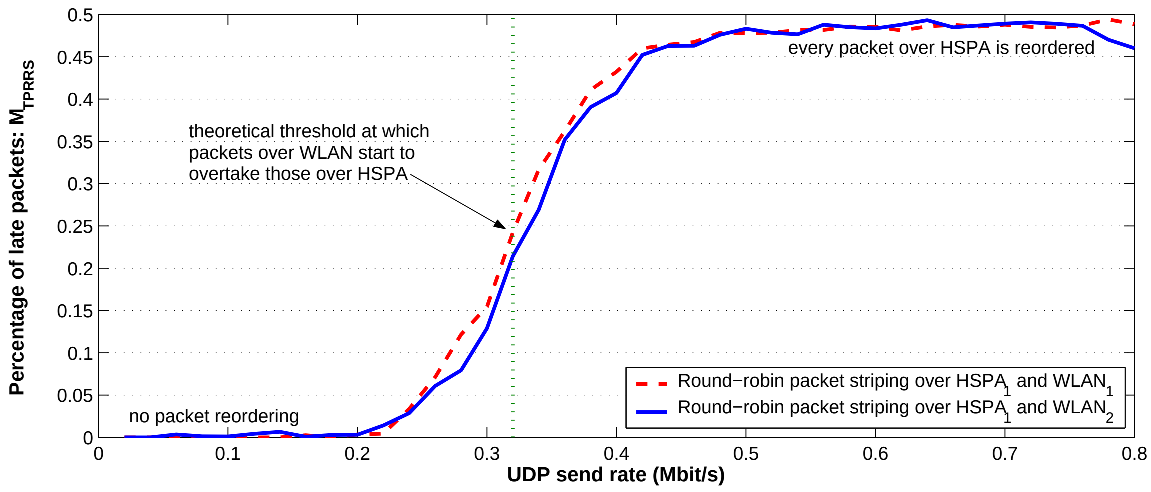


Figure 4.2: *The metric* Type-P-Reordered-Ratio-Stream *expresses the percentage of late packets in a sequence. For different WLAN networks, the ratio of late packets is alike.*

The vertical line in Figure 4.2, located roughly at 0.32 Mbit/s, indicates the critical bitrate B_{crit} at which equally-spaced packets over the WLAN paths start to "overtake" packets sent over the HSPA path, causing the latter to arrive late. Below this theoretical threshold, the packets are sufficiently spaced apart in the stream, so that packets sent over WLAN do not overtake those sent over HSPA. Much above the B_{crit} threshold, the inter-packet spacing is so small that after a packet p is sent over HSPA, several packets can be transmitted over the WLAN path before p arrives. If d_1 and d_2 denote the average transmission delays over two different access networks and s designates the packet size in bits, then the critical bitrate B_{crit}, at which late packets are expected, is defined as:

$$B_{crit} = s/(|d_1 - d_2|) \tag{4.1}$$

The special case of $d_1 = d_2$ corresponds to $B_{crit} \to \infty$ and to a reorder-free transfer. In a deterministic world without jitter, each packet would travel with a constant delay, and M_{TPRRS} would jump from 0 % to 50 % precisely when the data rate surpasses B_{crit}.

In the experiment described above, the average difference in transmission delays $|d_1 - d_2|$ is 37 ms, and the total packet size used was 1480 bytes, which leads to a critical bitrate of $B_{crit} = 0.32$ Mbit/s. In a practical experiment, such as the one presented here, B_{crit} is not sharply defined because of delay variations. With a growing bitrate, a gradual increase in reordering can thus be observed. From the point where almost every packet on the low-latency path is passed by a packet with higher sequence number on the high-latency path, M_{TPRRS} remains close to 50 %.

In contrast to previous Internet measurements on IP packet reordering (→ Section 2.2.1), this experiment confirms that in a scenario of multipath forwarding, the ratio of reordered packets increases far beyond the stated 0 % to 5.8 % and quickly approaches 50 % for data rates greater than the critical bitrate B_{crit}.

Packet Reordering in Terms of Displacement (M_{RD})

When multipath forwarding is the main reason for out-of-order delivery, a metric like M_{TPRRS}, which expresses the ratio of reordered packets in a stream, is unable to provide enough information about the full extent of IP packet reordering. For this reason, RFC 5236 [84] introduces an improved metric, the reorder density M_{RD}, providing a detailed distribution of how far packets are displaced from their original position.

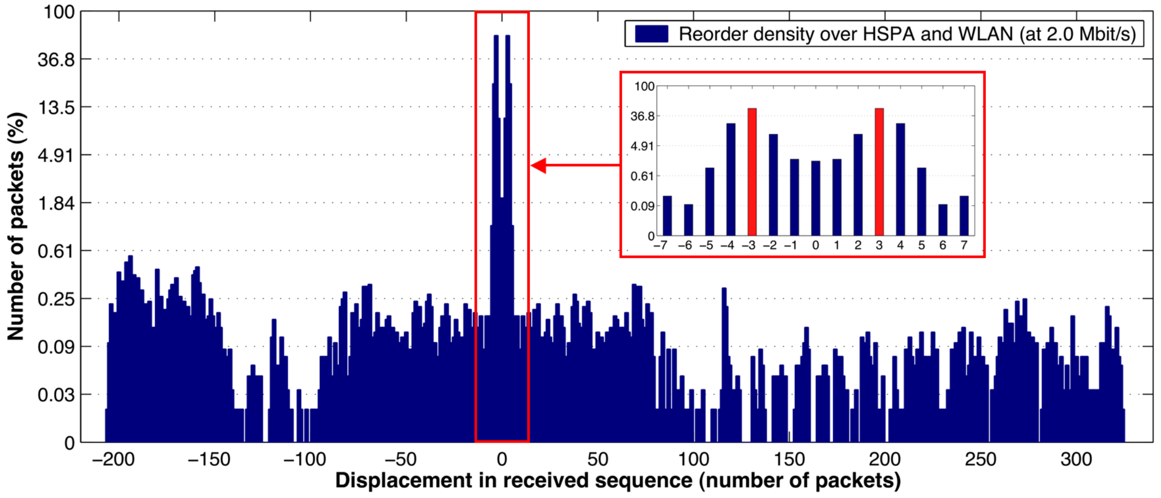


Figure 4.3: *An example reorder density obtained by round-robin packet scheduling over our HSPA$_1$ and WLAN$_1$ networks (for illustrative purposes, the y-axis is in logarithmic scale).*

Figure 4.3 shows a graphical representation of an example M_{RD} obtained by sending packets in a round-robin manner over the HSPA$_1$ and WLAN$_1$ networks at a constant data rate of 2.0 Mbit/s. This bar plot demonstrates that a considerable fraction of the packets experienced a large degree of reordering. Some packets were displaced as much as 300 positions from their original location in the sequence. Only 2 % of the packets were received as expected (i.e., $M_{RD}(0) = 0.02$), while the majority of the packets were displaced by three positions (highlighted in the zoomed portion).

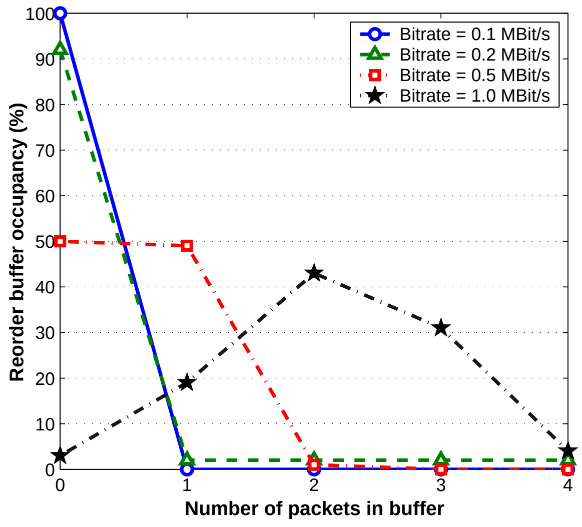
The most frequent packet displacement (where M_{RD} reaches a maximum) can also be estimated by calculating the number of packets that fit into the average delay difference $|d_1 - d_2|$ between the two paths. In the example of Figure 4.3, the data rate $B = 2$ Mbit/s corresponds to about 167 full-sized packets sent per second. Round-robin scheduling implies that half of the data travels over each path, roughly 83 packets per second. The delay difference between the paths used in the experiment is 37 ms on average. In this time span, three packets ($83 * 0.037 \approx 3$) are sent over the high-latency HSPA$_1$ path and overtaken by packets sent over WLAN$_1$. In summary, the average expected displacement x (in packets of size s bit) caused by round-robin scheduling over two paths with average delays d_1 and d_2 can be estimated as follows:

$$x \approx \frac{B}{2s} * |d_1 - d_2| \tag{4.2}$$

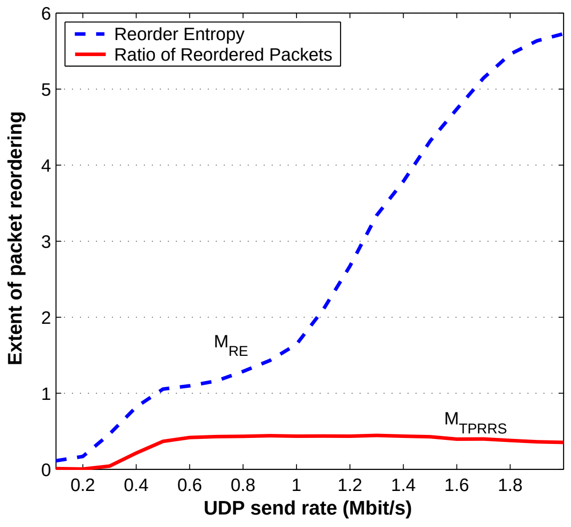
In the context of TCP, the reorder density metric might be very useful because TCP is not simply affected by the number of reordered packets, but mostly by their displacement. TCP's standard mechanism of fast retransmit (→ Section 5.1.2) interprets packets as lost if they have a positive displacement of three or more positions, in which case TCP retransmits the presumably lost packet and additionally reduces its send rate. If a standard TCP connection was subjected to packet reordering as high as shown by Figure 4.3, fast retransmit would be triggered continuously, resulting in a very low throughput and a large amount of the data needlessly sent multiple times.

Packet Reordering in Terms of Buffer Occupancy (M_{RBD})

If in-order delivery is a requirement, transport protocols and applications typically use a buffer to restore order. The *reorder buffer occupancy density* metric, M_{RBD}, gives a sense of the buffer size required by a receiver to restore order in a packet sequence. Figure 4.4(a) illustrates the M_{RBD} metric in the example of packet-based scheduling over two heterogeneous paths WLAN_1 and HSPA_1. When the bitrate is set to a low enough value, such 0.1 Mbit/s in this example, no reordering occurs and the reorder buffer remains unutilized. With a higher send rate, however, packets experience more reordering, which demands a larger buffer. In the example of 1.0 Mbit/s, the buffer becomes occupied by 2 packets in 45 % of the time and by a maximum of 4 packets in 5 % of the time. The area under each curve in Figure 4.4(a) amounts to 100 %, indicating that no packets were discarded, since we used an unlimited buffer in the experiments. However, with a buffer capacity of only 3 packets, 5 % of the received packets would have been dropped due to the lack of buffer space. If packet discard is not tolerable, the required size of a reorder buffer grows quickly with an increasing bitrate. In other words, the buffer capacity needs to be large enough to store the most displaced packet in the corresponding M_{RD} histogram. On the other hand, applications that are resilient to a certain degree of packet loss or reordering might cope with a smaller buffer.



(a) Reorder buffer occupancy density



(b) Reorder entropy

Figure 4.4: *Examples of enhanced packet reordering metrics include (a) the reorder buffer occupancy density, and (b) the reorder entropy.*

Packet Reordering in Terms of Entropy (M_{RE})

In contrast to the M_{TPRRS} metric, which simply counts the number of late packets, improved packet reordering metrics provide more information. Both the reorder density M_{RD} and the reorder buffer occupancy density M_{RBD} express not only how many packets are affected by reordering, but include a notion of the displacement from their original position in the sequence. On the other hand, both the M_{RD} and the M_{RBD} metrics are not singleton values, but distributions of values, which complicates the direct comparison of results. Ye et al. [169] therefore proposed the reorder entropy metric M_{RE}, which compresses the reorder density into a single value. Although the M_{TPRRS} and M_{RE} metrics are not directly comparable, Figure 4.4(b) clearly illustrates the ability of the reorder entropy to express severe packet reordering. While the reorder entropy gradually increases as the send rate of packets over WLAN$_1$ and HSPA$_1$ is raised, the M_{TPRRS} metric quickly flattens out. The reorder entropy is thus a very suitable metric for a generic comparison of the performance of multipath schedulers. As a next step, we employ the reorder entropy to verify whether an enhanced scheduler at the sender is capable to reduce packet reordering at the client.

4.1.2 An Experiment of Sender-side Reorder Reduction

A possible way of dealing with multipath-inflicted packet reordering is to send packets in a precalculated disorder, so that they arrive in correct order at the receiver. To measure the potential of this idea, we have modified the sender-side UDP packet scheduler from simple round-robin assignment to a more sophisticated method that delays the transmission of packets over the low-latency WLAN$_1$ path. Using the estimated average packet displacement given by Equation (4.2), the scheduler calculates how much packets over the low-latency path have to be displaced (delayed) in the original sequence so that they arrive in order. Figure 4.5 shows how the total packet disorder reduces when a reorder-compensating scheduler is used at the sender. Compared to the purely round-robin packet scheduling, the improved scheduler decreases the reorder entropy M_{RE} on average by 38 %.

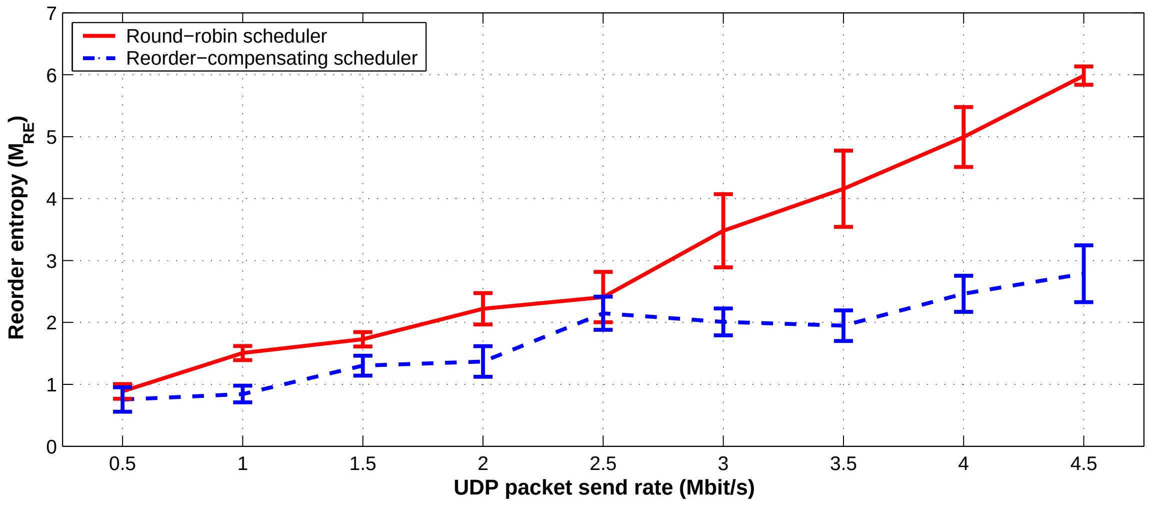


Figure 4.5: *Comparison of the reorder entropy M_{RE} for a purely round-robin scheduler versus a scheduler that attempts to compensate for reordering at the sender.*

As the reorder-compensating scheduler lacks information of the two paths' latency, a constant difference in one-way delays of 40 ms between WLAN_1 and HSPA_1 was assumed. The inaccuracy of this assumption results in the reordering observed. A reason why packet reordering was not fully reduced in this experiment is the static nature of the scheduler, which does not adjust itself to currently monitored path characteristics. Additionally, the latency of each path varies over time and indirectly depends on other factors, such as the send rate. Therefore, a static scheduler based on fixed delay estimates cannot be expected to perfectly solve packet reordering in an environment with the dynamic heterogeneity we observed in wireless networks. The design of a scheduler that adaptively adjusts to the link delays is desirable.

4.2 Network-layer Adaptive Multipath Aggregation

Although the experiments in the previous Section 4.1 indicate a large degree of IP packet reordering caused by multipath forwarding, it seems to be possible to actively reduce the reordering at the client by a proper scheduler at the server. Our experiments have also shown that aggregating heterogeneous access networks is feasible, but for aggregation to be efficient it is necessary that the scheduler gains knowledge of the currently available bandwidth and the latency of each path. For mitigating IP packet reordering and obtaining efficient aggregation at the same time, a dynamic scheduler is needed that adapts to both throughput and delay estimates. In this section, we incorporate these insights into a network-layer adaptive multipath aggregation prototype called NAMA[1]. Details of the Linux 2.6.28 implementation are presented in Section 4.2.1, and its performance is analysed in Section 4.2.2.

4.2.1 NAMA Architecture and Components

Figure 4.6 shows the four main components of NAMA and illustrates their interworking.

1. **NAT** The first component uses network address translation (NAT) to redirect traffic and to provide the basic functionality of network-layer multipath support. The destination IP address of outgoing packets is rewritten to the IP address of the desired receiver interface. An inverse address translation at the receiver is performed with the Linux *iptables* command to transparently merge IP packets back into a single flow.

2. **Packet scheduler** The second component is implemented as a dynamically loadable kernel module and performs simple lookups in the send vector to determine the path to be used for the next outgoing packet. The send vector itelf is updated regularly by the path monitor module according to the throughput experienced over each path, thus allowing adaptive multipath aggregation.

3. **Delay equalizer** The third component is implemented as an independent user space module that attempts to reduce packet reordering at the client. By briefly buffering packets that are destined to travel over a low-latency path, the delay equalizer effectively causes all paths to have close to equal latency. Path monitoring feedback is used to ensure up-to-date latency values.

[1]The NAMA prototype was published in [44] as a solution without name.

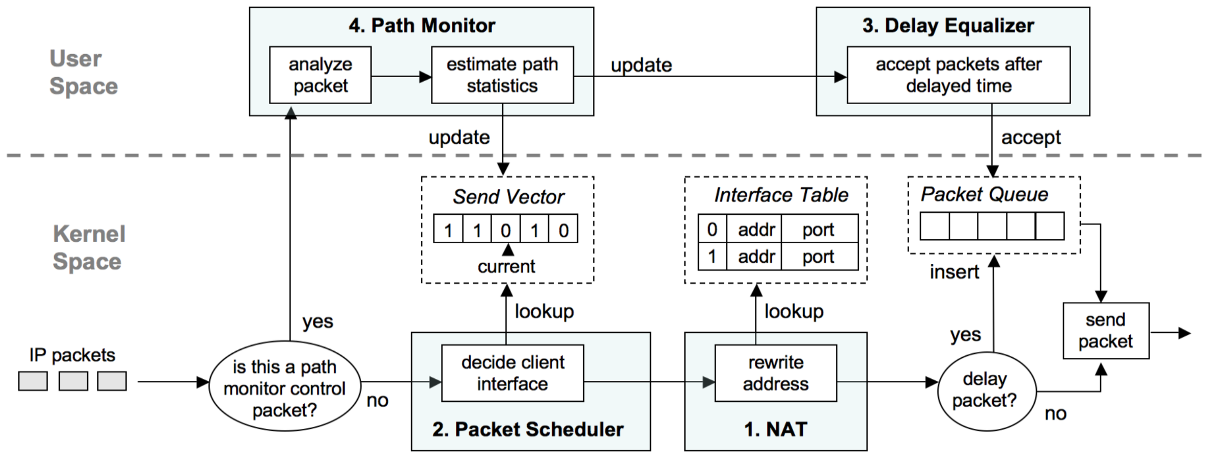


Figure 4.6: *Architecture of NAMA. The destination address of outgoing packets is rewritten using NAT (1), according to a send vector (2). Latency differences are equalized by queuing packets destined to travel over a low-latency path (3). Dynamic adaptation to differences in throughput and latency is facilitated through path monitoring feedback from the client (4).*

4. **Path monitor** The fourth component is a user space tool that collects estimates of the path throughput and latency and uses them to update the send vector and the delay equalizer. Latency estimates are derived from sending TCP probe packets every 200 ms and measuring the time until an ACK returns. Throughput estimates are obtained by measuring the client's incoming data rate. When the send vector locks to an unwanted pattern, such as $V = \{0, 0, 0, 0, ...\}$, packet-pair probing [95, 138] is used to estimate the available bandwidth. Several back-to-back packets are sent to the receiver, which infers the path capacity from the dispersion of the inter-arrival times.

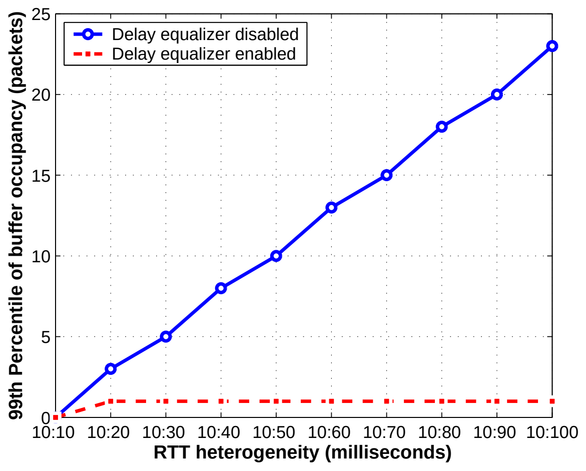
In the following section, the performance of NAMA is evaluated based on network emulation experiments.

4.2.2 Performance Evaluation of NAMA

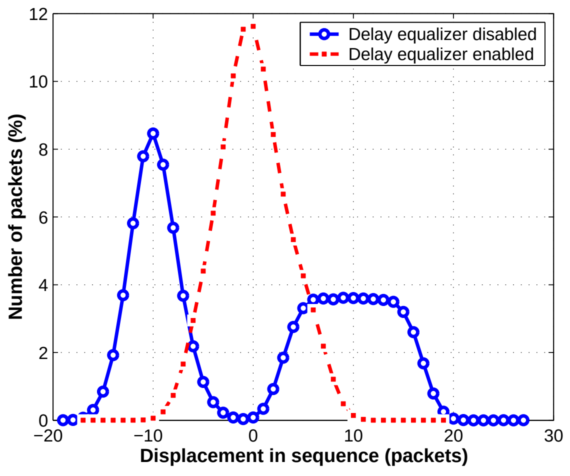
In the analysis of NAMA, two characteristics are of particular interest: first, the ability of the delay equalizer to mitigate IP packet reordering at the receiver, and second, the capability of the entire system to aggregate multiple heterogeneous paths and adapt to changes in throughput. All the following experiments are performed using the emulation testbed described in Section 3.3. The `iperf` application is used to send 1480-byte UDP packets at a constant bitrate of 6 Mbit/s from the server to the client, and NAMA transparently schedules the packets over two available paths.

Reducing IP Packet Reordering at the Receiver

In a first experiment, we investigate how NAMA reduces the receiver's workload of buffering out-of-order packets with heterogeneous and variable path latencies. A static send vector $V = \{0, 1\}$ is used to equally spread the traffic over both paths. Figure 4.7(a) illustrates the delay equalizer's robustness to large differences in round-trip time. When the delay equalizer module is disabled, IP packet reordering rapidly grows as the path RTTs become increasingly heterogeneous, requiring a much larger buffer at the receiver to store out-of-order packets. On the other hand, when the delay equalizer is enabled, packet reordering is significantly reduced, and the buffer occupancy at the client decreases to a minimum. Under stable network conditions, a buffer with the capacity for only a single packet is sufficient to guarantee less than 1% packets to be lost due to buffer overflows.



(a) Robustness to RTT heterogeneity



(b) Robustness to RTT jitter

Figure 4.7: *The delay equalizer is able to eliminate IP packet reordering in situations with (a) RTT heterogeneity and (b) RTT jitter.*

Figure 4.7(b) illustrates the delay equalizer's ability to considerably reduce IP packet reordering even if the used links exhibit variable delays. In the experiment that produced this plot, the two paths were emulated to roughly resemble the heterogeneity between WLAN and HSPA in a realistic scenario. The one-way latency of the first path was set to an average of 10 ms with ±5 ms jitter (normally distributed) and the one-way delay of the second path was set to 50 ms and ±15 ms jitter (uniformly distributed). According to Equation (4.2), this network configuration should cause packets to be displaced on average by about ten positions. When the delay equalizer was disabled, as predicted by the equation, the majority of packets experienced reordering with a displacement of −10 and +10 sequence numbers, clearly visible as two peaks in the figure. With the delay equalizer enabled, these two peaks were merged into a single peak at a displacement of 0, implying that the majority of the packets exhibited no reordering at all. This shift results in a reduction of the 99th percentile value of the buffer occupancy from 18 packets to 9 packets; in other words, a 50% decrease in buffer requirements.

Adapting to Variable Path Characteristics

In a second experiment, we test NAMA's robustness to bandwidth heterogeneity and the ability of the path monitor module to accurately update the send vector. For this, the bandwidths of the two paths were throttled to different values, while ensuring that their sum is always equal to the data send rate of 6 Mbit/s. The send vector was limited to 64 elements and no additional latency was introduced, implying RTTs below 1 ms. Table 4.1 shows that for a variety of bandwidth ratios, the aggregated throughput at the client was measured as 5.99 Mbit/s and at most 0.12% of the traffic was lost. These results are averages of five experiment batches, each lasting 10 minutes.

Table 4.1: NAMA – Throughput Aggregation for various Bandwidth Ratios

Bandwidth ratio	1:11	2:10	3:9	4:8	5:7	6:6
Avg. aggregated throughput (Mbit/s)	5.99	5.99	5.99	5.99	5.99	5.99
Out-of-order packets (M_{TPRRS})	1.56	1.98	1.68	1.29	2.03	0.09
95th percentile buffer occupancy (packets)	0	0	0	0	0	0
99th percentile buffer occupancy (packets)	1	1	1	1	1	0
Lost packets (%)	0.10	0.12	0.11	0.05	0.04	0.01

Table 4.2: NAMA's Robustness to Variable Bandwidth and RTT

	Round-robin scheduling	NAMA
Average aggregated throughput	4.49 Mbit/s	5.86 Mbit/s
99th percentile buffer occupancy	26 packets	9 packets
Packet loss	25%	2.2%

While the results in Table 4.1 were conducted under stable path conditions, Figure 4.8 depicts results obtained from an experiment in which the bandwidth of the emulated paths was abruptly changed at constant intervals of 30 seconds. Systems without multipath support can only achieve the throughput of either Path 1 or Path 2, while NAMA is able to adaptively aggregate the throughput of both paths. The spikes in the aggregated throughput are caused by the path monitor's time to update the send vector pattern.

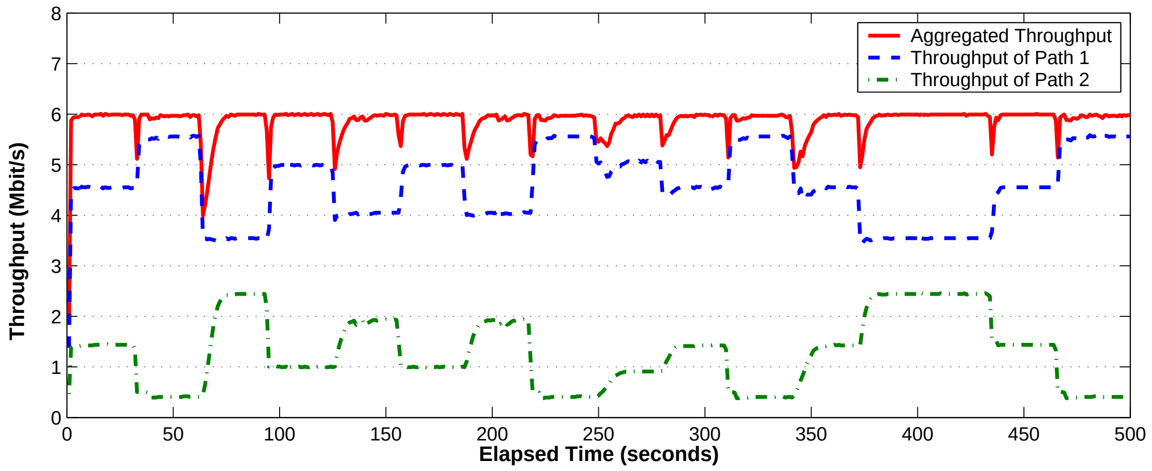


Figure 4.8: *NAMA dynamically adapts to arbitrary bandwidth changes. In this example, the bandwidth of Path 1 and Path 2 are configured to always sum up to 6 Mbit/s.*

We also compared the performance of NAMA to pure round-robin packet scheduling, when both the bandwidth *and* RTT are dynamically changing over time. The parameters used are a combination of the two experiments conducted to produce Figures 4.7(b) and 4.8. In other words, the bandwidths of the two paths were given random values that add up to 6 Mbit/s, while their RTT characteristics were modeled as an approximation of WLAN$_1$ and HSPA$_1$. Table 4.2 summarizes the results and clearly shows that NAMA outperforms pure round-robin packet striping in all aspects: it achieves higher aggregated throughput, a lower packet loss ratio, and it requires less buffer capacity at the client.

4.3 Discussion

From the results in Section 4.1 it can be concluded that IP packet reordering over multiple paths is typically an order of magnitude higher than when regular flow-based forwarding over single paths is used. In fact, our measurements with heterogeneous WLAN and HSPA networks showed that multipath packet reordering can be so severe that the expressiveness of the M_{TPRRS} metric is exhausted, even at very low send rates. While the M_{TPRRS} metric, which is commonly used in Internet measurements, is unable to provide enough insight into heavy packet disorder, improved metrics, such as the reorder density (M_{RD}) and the reorder buffer occupancy density (M_{RBD}), provide a clearer picture.

In order to reduce multipath-inflicted IP packet reordering, we presented the idea of sending data in a modified sequence from the sender. We first tested our idea in Secion 4.1.2 using a static scheduler. In Section 4.2.2, as part of NAMA, we evaluated an improved scheduler that dynamically adapts to changes in path latency and throughput. In various experiments including large ranges of path heterogeneity, NAMA was able to efficiently aggregate multiple paths and at the same time to reduce the amount of reordering observed by the receiver. While the send vector stood out as a viable approach to aggregate multiple path, we learned two important lessons:

- It is practically impossible for a sender-side scheduler to fully eliminate packet reordering. Although we achieved a significant reorder reduction in all experiments, a certain degree of reodering always occurred under dynamic path conditions. Only in scenarios with stable bandwidth and latency, NAMA achieved zero packet reordering. An alternative solution with a resequencing buffer at the receiver may be more suitable, even though the added workload might burden certain low-power devices with small memory.

- Relying on active path monitoring to make scheduling decisions turned out to be a viable performance enhancement, but it is vulnerable to RTT jitter. Overloaded and highly dynamic paths can lead to significant inaccuracies in the estimated throughput and latency, which increases packet reordering and reduces the aggregation benefit. We are convinced that relying on active path monitoring is not a viable approach to eliminate packet reordering caused by multipath forwarding.

We also realized that a solution based on rewriting IP destination addresses allows an easy configuration in a local testbed, but does not work on all paths through the Internet. Many access networks interpret our approach as a security threat and deny forwarding packets with address fields that not topologically correct (i.e., *ingress filtering*). A solution to this problem is to set up a virtual network by encapsulating all traffic, similar to the method suggested in [131] and the configuration we employ in Section 5.2.1. An approach that is part of our future work is the MULTI [46] framework, our successor to NAMA. MULTI builds an overlay network between a proxy and multihomed clients and uses virtual interfaces and IP tunneling to enable multipath aggregation and vertical handover.

NAMA's Compliance with the Requirements for Multipath Aggregation

Our experiments have shown that NAMA's adaptive nature achieves efficient multipath aggregation and copes with large path heterogeneity, therefore satisfying the first three evaluation criteria listed in Section 2.1.1. The transparent concept of forwarding IP packets over multiple paths also provides compatibility with any IP-based transport protocols and existing applications, therefore covering criteria C4. However, criteria C5 of leaving servers unchanged is clearly violated by the complex scheduling software introduced on the sender. This problem could be resolved by moving the NAMA software from the sender to an intermediate proxy node, as illustrated in Figure 4.9. On the other hand, outsourcing NAMA to a proxy violates criterion C6, which states that introducing additional infrastructural elements endangers scalability. It also introduces *fate sharing* between the client and the proxy. However, we consider a proxy-based approach as highly promising, since it allows quick deployment and compatibility with any third-party servers. The proxy can be located anywhere in the Internet, preferably close to the clients it serves.

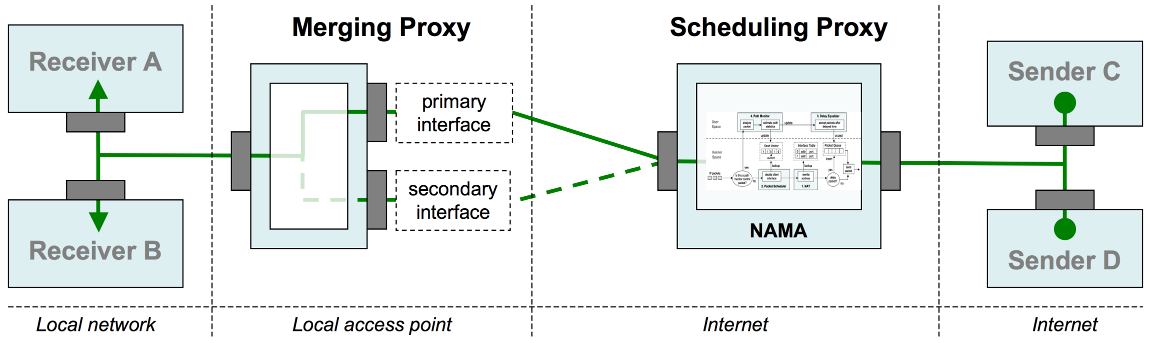


Figure 4.9: *Proxy-based multipath aggregation at the network layer allows quick deployment, because third-party servers can be left completely unmodified. Merging IP packets at a local access point can be used to improve the performance of single-interface receivers.*

For proper operation of NAMA, the multihomed receiver needs some modifications, but none that affect network protocols or its operating system. NAMA's client-side multipath support is solved by simple configuration commands, while path monitoring can be implemented exclusively in user space. Criterion C6 can thus be considered satisfied.

Implications on TCP

Even though criterion C4 is fulfilled by NAMA's protocol-transparent packet forwarding, it is not always capable to fully eliminate reordering. In cases where packet reordering still occurs, transport protocols like TCP and SCTP may be negatively affected. Preliminary tests with TCP lead us to the conclusion that NAMA's path monitor interfers with TCP's own RTT estimation and a proper growth of the congestion window. With the goal of improving NAMA's performance and gaining a better understanding of TCP's behavior to multiple paths, the following Chapter 5 contains an in-depth analysis.

Chapter 5

Multiple Paths at the Transport Layer

The Transmission Control Protocol (TCP) is the most frequently used protocol in the Internet today. Initially designed several decades ago, it does not support multiple network interfaces per endpoint and it makes strong assumptions about in-order delivery of IP packets. If packets are not delivered in the expected order, for example because they are forwarded over multiple paths, TCP is prone to wrongly interpret reordered packets as lost, which might lead to a severe throughput degradation.

In Section 4.1 we conducted experiments to quantify the heterogeneity of commonly used WLAN and HSPA networks, which lend themselves as excellent candidates for multipath aggregation. Our measurements verified that packet-based forwarding causes packet reordering much higher than the 0 to 5.8% commonly observed in the Internet. Even at low transfer speeds, differences in the path latency have a significant impact on packet reordering. For example, forwarding packets in a round-robin fashion over two paths with equal capacity, but different latency, easily causes 50% of the packets to arrive out of order. Under such circumstances, it becomes meaningless to count the fraction of reordered packets. What really counts for TCP is not how often packets get reordered, but the number of positions they get out of sequence. Multipath forwarding likely causes packet reordering of more than three positions, which, according to RFC 5681, is the standard threshold for TCP to assume a packet as lost (rather than arriving out of sequence).

Motivated by the existence of IP packet reordering in the Internet and TCP's standard mode of operation, researchers suggested many methods for improving TCP's robustness against reordering. These methods will be discussed in Section 5.1.6. However, to the best of our knowledge, no studies exist that measure the impact of *multipath*-inflicted packet reordering on TCP in practical scenarios and running the protocol stack of a real operating system. Apart from simulation-based comparisons of novel reorder-robust schemes and standard TCP, such as Bhandarkar's comparison of TCP-DCR [22] to standard SACK TCP [49, 114], there exists no experimental evidence of how either the standard TCP promoted by the IETF, or a full-featured and readily available TCP variant like Linux TCP [145] performs in the presence of packet-based forwarding over multiple, heterogeneous paths. The work presented in this chapter thus aims at the following goals:

1. We are curious to see what *really* happens to TCP's data throughput, when IP packets are forwarded over heterogeneous access networks, such as WLAN and HSPA.

It is conceivable that multipath forwarding completely destroys TCP's performance (as hinted in simulation-based tests with SACK TCP by Bhandarkar et al. [22]). On the other hand, TCP implementations in modern operating systems, such as Linux, contain a variety of performance-enhancing mechanisms protecting against IP packet reordering. Before designing new solutions from scratch, such as MPTCP [56], we therefore consider it necessary to first investigate the multipath performance of TCP as it exists today by testing its operation in a severe case of path heterogeneity given by the widely used wireless access networks of WLAN and HSPA.

2. For a thorough investigation of TCP's multipath performance, it is important to consider a large variety of path characteristics and TCP mechanisms. In addition to a scenario based on WLAN and HSPA networks, it is important to analyze a large spectrum of possible network combinations, including path heterogeneity in terms of bandwidth, latency, and randomly added packet loss. In addition, TCP's multipath performance may heavily depend on the congestion control algorithm used and the loss recovery mechanisms in place. To this end, we run a large set of emulation-based experiments considering a wide variety of TCP parameter configurations.

3. Furthermore, we want to find out if TCP's multipath performance can be improved without changes to the protocol itself, but by properly tuning its parameters.

Before analyzing the multipath performance of TCP, Section 5.1 begins with a detailed explanation of TCP with an emphasis on mechanisms that are related and relevant to IP packet reordering. In Section 5.2 we then address the first two of the above mentioned goals by testing TCP in practical and emulated multipath scenarios with WLAN and HSPA. In addition, we carefully design an extensive set of experiments to analyze the effect of path heterogeneity and the impact of various TCP-related mechanisms on multipath performance. The insights gained from this analysis are then applied in Section 5.3 for achieving an improved aggregation benefit by fine-tuning Linux TCP parameters.

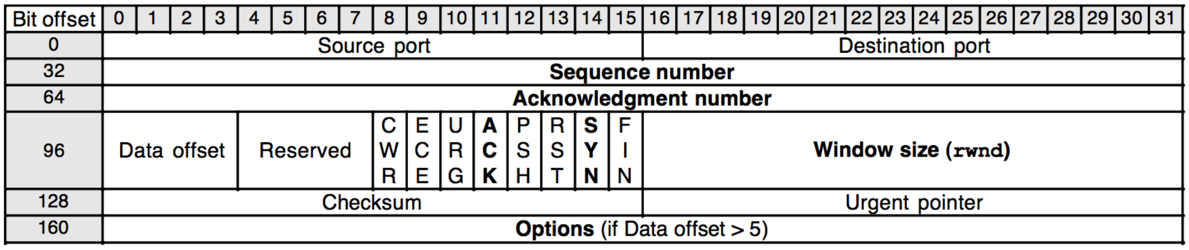
5.1 Relevant Features of the TCP Protocol

The main purpose of TCP is to *reliably* deliver an *ordered* sequence of bytes from a sender to a receiver. The ARPANET's initial host-to-host communication protocol [39] had no provision for error control between two endpoints. In fact, the world's first datagrams, which were sent on October 29 in 1969, also serve as the Internet's first example of a transmission failure. An anecdote [141] states that the full message was supposed to contain the word "login". However, an error at the letter "g" caused only the fragment "lo" to successfully arrive at the other end. In addition, the host-to-host protocol permitted only a single message to traverse a connection at any time and each message was individually acknowledged in a "stop-and-wait" procedure [162] – a highly inefficient utilization of the available capacity.

In 1974, with the vision to overcome such limitations and to create an *inter*-network communication protocol, Vint Cerf and Robert Kahn presented a protocol for reliable communication between different packet-switched networks, TCP [32, 33]. An improved TCP specification [136] became the norm when the ARPANET officially adopted the TCP/IP protocol suite on January 1, 1983 [135].

5.1.1 Connection Establishment and Flow Control

Every TCP data transfer (often referred to as a *flow*) begins by establishing a connection between two endpoints in a three-way handshake that is defined by an enabled `SYN` control flag in the TCP header (see Figure 5.1). In this synchronization phase, the sender and receiver use the header's sequence number field to negotiate an initial sequence number. After the connection is established, the `SYN` flag is disabled and the sequence number field is interpreted as a pointer to the starting position of a data segment in the byte stream.



Bit offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	Source port																Destination port															
32	**Sequence number**																															
64	**Acknowledgment number**																															
96	Data offset				Reserved				CWR	ECE	URG	**ACK**	PSH	RST	**SYN**	FIN	**Window size (`rwnd`)**															
128	Checksum																Urgent pointer															
160	**Options** (if Data offset > 5)																															

Figure 5.1: *TCP's header format.*

TCP's flow control ensures that a sender does not send more data than a receiver can handle. Applications that use TCP expect that data arrives at the receiver in the exact same order in which it was sent. A TCP receiver must therefore buffer out-of-order data until all gaps in the stream are filled before forwarding a complete and ordered sequence of bytes to the receiving application. In order to control the data flow sent by the sender, the receiver specifies a receive window (`rwnd`) in every ACK. The `rwnd` informs the sender about how many bytes (starting with the acknowledgment number) it is allowed to transmit without overloading the receiver buffer. This concept of flow control has remained unchanged since 1981, when it was documented in RFC 793 [136].

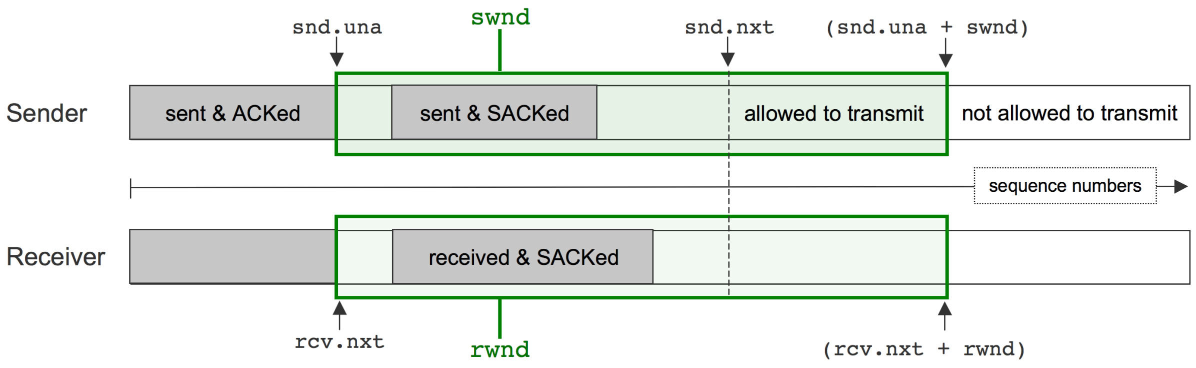


Figure 5.2: *TCP flow control windows and variables*

Figure 5.2 illustrates the `rwnd`, the sender's corresponding send window (`swnd`), and several variables relevant for flow control. In the context of this work, the most important variables are `snd.una`, which points to the lowest unacknowledged sequence number, and `snd.nxt`, which points to the first byte that has not been sent yet. All data between `snd.una` and `snd.nxt` is either outstanding in the network or lost, except the dark shaded areas, which represent selectively acknowledged data using the optional SACK mechanism described later in Section 5.1.5.

5.1.2 Loss Recovery

TCP works on top of the unreliable link and network layers, where data segments can get lost (e.g., dropped by an overloaded router queue or corrupted by a wireless link). In order to guarantee reliable data transmission, a TCP sender must retransmit lost data segments until they successfully arrive at the receiver (in multiple attempts if necessary). When a new segment arrives, the TCP receiver answers with an acknowledgment (ACK) to inform the sender of the successful data transfer. The receiver creates an ACK by enabling the `ACK` flag in the header and writing its next expected sequence number into the acknowledgment number header field. The sender interprets an ACK as the successful arrival of all bytes up to this acknowledgment number. In other words, TCP can *cumulatively* acknowledge multiple segments with a single ACK.

Timeout-based Retransmission

TCP senders assume a segment as lost if no ACK returns in a time period called the *retransmission timeout* (RTO), in which case the missing segment is retransmitted. Known as *Jacobson's Algorithm* [77], the RTO value is continuously updated based on *smoothed round-trip time* estimates (`SRTT`) and the *round-trip time variation* (`RTTVAR`). When a timeout occurs, the sender retransmits the lost segment and backs off by doubling the RTO. This *exponential backoff* was introduced as a counter-measure to congestion collapses in the early 1980s (although, some researchers claim that the RTO backoff could safely be removed from TCP without causing instability to the Internet [117]). Algorithm 2 outlines the basic procedure for computing the RTO as specified in RFC 2988 [127].

Algorithm 2 Computing TCP's Retransmission Timer

```
1: Initialize: RTO = 3 (seconds)
2: Set constants: K = 4, $\alpha = 1/8$ and $\beta = 1/4$
3: if the first RTT measurement R is made then
4:    SRTT = R
5:    RTTVAR = R / 2
6:    RTO = SRTT + K * RTTVAR
7: else if a subsequent RTT measurement R' is made then
8:    RTTVAR = (1 - $\beta$) * RTTVAR + $\beta$ * |SRTT - R'|        {RTT variation}
9:    SRTT = (1 - $\alpha$) * SRTT + $\alpha$ * R'                   {Smoothed RTT}
10:   RTO = SRTT + K * RTTVAR
11: end if
12: if RTO < 1 second then
13:   RTO = 1                                                     {Enforce minimum RTO}
14: end if
```

A TCP sender can potentially sample the RTT with every incoming ACK. However, if an ACK contains the sequence number of a retransmitted segment, it is unknown if the ACK is the receiver's answer to a (reordered) original segment or to the retransmission. TCP must therefore use *Karn's Algorithm* [87], which ensures correct RTT estimates by ignoring ACKs to retransmitted segments. Alternatively, if both endpoints support the timestamp extension, reliable RTT estimates can be computed by including the current time as a timestamp option field [79] into TCP's header whenever a segment is sent.

Fast Retransmit

Instead of waiting a considerably long time (i.e., about one RTT) until the retransmission timer expires for a missing segment, TCP employs an additional scheme for loss recovery when congestion is mild, called *fast retransmit*. The main idea behind fast retransmit is that if segment loss can be detected early, then the lost segment can be retransmitted before an RTO passes. Since IP packets can get lost, reordered, or even duplicated in the network, a TCP receiver often observes gaps of outstanding data in the received bytestream. In this case, even though new data has arrived, the receiver cannot increase the acknowledgment number, but must anyway reply with an ACK to the sender with an unchanged acknowledgment number. An ACK with an acknowledgment number identical to a previous ACK is called a *duplicate ACK* (DupACK). Even though a DupACK seemingly conveys no information, it tells a sender that at least one segment is missing at the receiver (either lost or reordered). DupACKs cannot be piggybacked on data.

Fast retransmit assumes that every consecutively received dupACK increases the confidence that a packet is not reordered, but in fact lost. Under the assumption that IP packets are rarely reordered by more than 3 positions in a sequence, receiving 3 consecutive dupACKs can be interpreted as an indication for a loss. If the fast retransmit threshold of 3 dupACKs is reached, the TCP sender resets the retransmission timer and triggers a *fast* retransmission. Later enhancements to TCP [25] use a state variable, often referred to as *dupACK threshold* or `dupThresh`, to allow for values larger than 3.

5.1.3 Standard Congestion Control

TCP's early behavior of loss recovery was too aggressive and caused *congestion collapse*, a vicious circle of data retransmissions. When routers receive more traffic than they can forward, some packets must be dropped from their overloaded queues. Early versions of TCP noticed these missing data segments but wrongly reacted to them by triggering retransmissions and injecting even more data into the network. This behavior overloaded the router queues even further, until the data throughput on the backbone collapsed by a thousandfold from 32 Kbit/s to 40 bit/s in October 1986 [77].

In order to avoid such congestion collapses, TCP was enhanced with a *congestion control* mechanism that includes a quick backoff from sending data when congestion is detected. In other words, congestion control is TCP's attempt of not sending more data than what can be *in flight* on the one-way path from the sender to the receiver.

There are two values that limit the number of bytes a sender is allowed to send before receiving any acknowledgment back: the receive window (`rwnd`) used for flow control and the *congestion window* (`cwnd`) used to avoid network congestion. While `rwnd` is a hard limit dictated by the receiver, `cwnd` is an estimated value local to the sender. The congestion window represents the estimated number of bytes that can be sent into the network without packets being lost (i.e., without causing congestion). A TCP sender must fulfill both the `rwnd` and `cwnd` limits and this calculates its *send window* (`swnd`) as:

```
swnd = min(rwnd, cwnd)
```

When `swnd` of segments are in transit, a TCP sender must receive an ACK before it is again allowed to send new data. Analogously, new data can only be sent if the condition `cwnd` > (`snd.nxt - snd.una`) is satisfied.

TCP congestion control is documented in its most recent form in RFC 5681 [9]. The main idea is that the congestion window is gradually increased until packet loss is detected. When packet loss is detected, the congestion window is abruptly reduced in order to back off from causing congestion. This behavior causes TCP's typical *sawtooth* development of the send window and is often referred to as *additive increase/multiplicative decrease* (AIMD). Figure 5.3 illustrates an example development of the TCP congestion window.

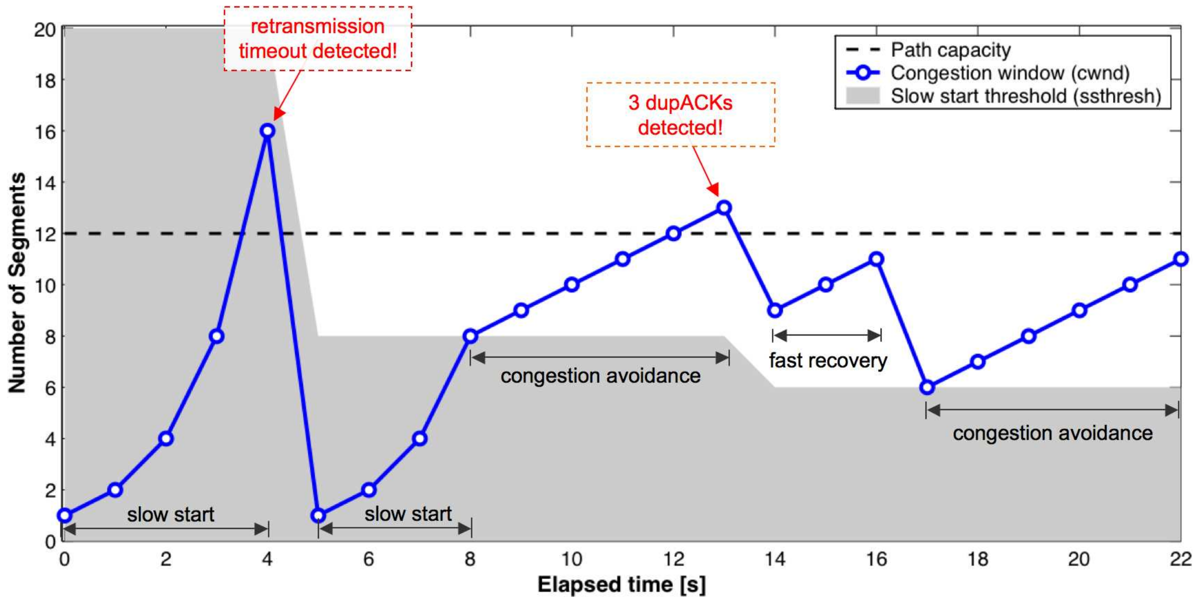


Figure 5.3: *Example TCP Congestion Window Development*

The growth of the congestion window is governed by two phases called *slow start* and *congestion avoidance*. The slow start phase is used to probe the network capacity by rapidly increasing the number of packets sent per round-trip time. Slow start ends as soon as a threshold is reached or lost packets are detected, which indicates that the current data rate is overwhelming the path capacity. When TCP is the phase of congestion avoidance, the load into the network is only gradually increased with the goal to prevent traffic congestion. The two phases are separated by the *slow start threshold* `ssthresh` variable, which can be interpreted as a conservative estimate of the current path capacity – a safe value for sending data without causing congestion. A newly established TCP connection always starts in the slow start phase, with a congestion window size equal to 1 segment, and a slow start threshold set to an arbitrarily high value. During slow start (`cwnd < ssthresh`), the congestion window is increased by 1 full-sized segment for every incoming ACK. During congestion avoidance (`cwnd > ssthresh`), the congestion window is incremented by 1 full-sized segment per RTT.

Under normal circumstances, the only event that stops the congestion window growth is when the sender detects that segments are missing (i.e., duplicate ACKs arrive at the sender or a timeout occurs). Since IP packet loss is caused by overloaded router queues on the majority of Internet paths, standard TCP interprets the loss of data segments as a sign of congestion in the network and reacts to it by retransmitting the lost segment and by reducing the congestion window and the `ssthresh`. As previously discussed in Section 5.1.2, a TCP sender attempts to recover from segment loss either after the ex-

Algorithm 3 TCP Reno Congestion Control

```
 1: if 1 or 2 consecutive dupACKs received then
 2:     Send a segment of previously unsent data                {Limited Transmit}
 3: else if 3 consecutive dupACKs received then
 4:     Retransmit the lost segment                             {Fast Retransmit}
 5:     ssthresh = cwnd[1] / 2
 6:     cwnd = ssthresh + 3
 7: else if 4 or more consecutive dupACKs received then
 8:     cwnd = cwnd + 1                                         {Fast Recovery}
 9:     Send 1 segment of previously unsent data (if possible)
10: else if finally an ACK arrives that is not a dupACK then
11:     cwnd = ssthresh (deflating the window)
12: end if
```

piration of a *retransmission timer* or after the reception of 3 duplicate ACKs (i.e., *fast retransmit*). Both cases indicate network congestion and cause a reduction of the slow start threshold to half of the congestion window, i.e., `ssthresh = cwnd / 2`. However, after a timeout, TCP is forced back into slow start by resetting the congestion window to 1, while after a fast retransmit, TCP is put into congestion avoidance by setting the congestion window equal to the slow start threshold (plus the three segment that caused the dupACKs). TCP congestion control that only includes RTO-based loss recovery and fast retransmit is typically referred to as *TCP Tahoe* [49, 77].

Fast Recovery

After a fast retransmit, modern TCP senders go into the state of fast recovery, which lasts as long as dupACKs keep arriving. During fast recovery, every incoming dupACK informs the sender that a segment has left the network and is now stored at the receiver. Without contributing to an increased network congestion, a sender is allowed to send a new segment for every dupACK and to increment `cwnd` by 1 to ensure that new data fits into the window. The fast recovery phase ends with the reception of an ACK that acknowledges new data (also called a *recovery ACK*). The `cwnd` is at this point reduced to the value of `ssthresh` that was computed when loss recovery was entered. Algorithm 3 illustrates TCP with both the fast retransmit and the fast recovery mechanisms, which is generally known as *TCP Reno* [49, 78].

Note that TCP uses the *limited transmit* algorithm for the first and second dupACK (before a possible fast retransmit occurs). This algorithm does nothing else than allowing TCP to send a new data segment for the first two dupACKs (but without incrementing the congestion window). Limited transmit was introduced in RFC 3042 [7] as an improvement for situations in which a lost segment is not followed immediately by enough duplicate ACKs to trigger the desired fast retransmit (for instance for very low-capacity paths, when the congestion window is very small, or when too many ACKs are lost).

[1]Instead of `cwnd`, RFC 5681 uses `FlightSize`, the sent but not cumulatively acknowledged data.

TCP New Reno

Due to the fast recovery mechanism, TCP Reno typically achieves a better utilization of the available path capacity than TCP Tahoe. However, TCP Reno has the disadvantage that only a single segment is retransmitted per RTT, even if multiple segments were lost from a window. TCP Reno's assumption that the fast recovery phase ends with an ACK for an entire window is not necessarily true. For example, if two consecutive segments are lost, then fast recovery will end with a *partial ACK* that acknowledges only the first of the two segments. The second lost segment, which was sent roughly one RTT ago and never retransmitted during fast recovery, will cause the retransmission timer to expire soon after the end of the fast recovery phase.

Many solutions to this drawback of TCP Reno have been proposed, most notably *TCP New Reno* [54]. TCP New Reno uses an additional variable (called `recover`) that is used to remember the highest transmitted sequence number at the time a fast retransmit occurs. TCP New Reno then stays in fast recovery until all lost segments up to `recover` are acknowledged. An alternative to TCP New Reno is the use of selective acknowledgements, explained in more detail in Section 5.1.5.

5.1.4 Enhanced Congestion Control Algorithms

Besides TCP Tahoe, Reno and New Reno, there are other congestion control algorithms that attempt to better utilize the available path capacity. They typically differ in terms of *how* congestion is detected. Not only lost packets can be interpreted as a sign of congestion, an increasing RTT, for example, can also indicate congested router queues. Another difference is that *if* congestion is detected, various TCP congestion control algorithms adjust `cwnd` and `ssthresh` in more moderate or more aggressive ways.

Most congestion control algorithms are designed for enhanced performance in special scenarios, for example in high-speed networks, over long-latency satellite links, or in lossy wireless environments. More exotic examples are TCP YeAH (Yet Another TCP) [15] by Baiocchi et al., which attempts to strike a balance between these various scenarios, and TCP-LP [99] by Kuzmanovic et al., which is tailored for low-priority, non-greedy utilization of the path capacity. Yang et al. [168] showed that a 45% majority of Web servers in the Internet use CUBIC [63] and BIC [167] congestion control, while less than 25% of Web servers still use the traditional TCP New Reno. In Table 5.1 we summarize all congestion control algorithms available in Linux (as of kernel 2.6.32) according to their basic functionality, supplemented by Yang et al.'s estimated usage in the Internet.

RTT-based Congestion Detection

TCP Vegas is an alternative to TCP New Reno proposed by Brakmo et al. [29] several years before. Instead of exclusively taking packet loss as a sign of congestion, TCP Vegas relies on accurate RTT estimates and interprets an increasing RTT as an early sign of congestion. TCP Vegas uses timestamps and samples the RTT for every segment. If a single dupACK is received in a time greater than the RTO, a retransmission is immediately triggered rather than waiting for three dupACKs. In addition, TCP Vegas uses its RTT estimates (and the bytes in flight) to calculate the difference between the current and the expected send rate. Congestion is assumed if the current rate falls below a certain

Table 5.1: Enhanced TCP Congestion Control Algorithms

TCP Version		Congestion detection		Specialized for high			Internet
Name	Ref.	loss-based	RTT-based	bandwidth	RTT	loss	usage [168]
New Reno	[54]	✓	–	–	–	–	$17 \sim 25\%$
BIC	[167]	✓	–	✓	✓	–	14%
CUBIC	[63]	✓	–	✓	✓	–	30%
H-TCP	[101]	✓	–	✓	✓	–	0.49%
Hybla	[31]	✓	–	–	✓	✓	–
Illinois	[104]	✓	✓	✓	✓	–	0.76%
Low-Priority	[99]	✓	✓	–	–	–	–
Scalable	[93]	✓	–	✓	✓	–	1.86%
Vegas	[29]	✓	✓	–	–	–	1.57%
Veno	[58]	✓	✓	–	–	✓	1.22%
Westwood+	[41]	✓	–	–	–	✓	2.82%
YeAH	[15]	✓	✓	✓	✓	✓	1.95%

threshold. Later introduced congestion control algorithms, such as TCP Veno [58] and TCP Illinois [104], often reuse ideas from TCP Vegas, but never rely as heavily on RTT estimates as a primary sign of congestion.

TCP in Wireless Environments

In wireless networks, standard TCP faces the challenge that segment loss may not be a sign of congestion. Channel errors can randomly corrupt data, causing TCP senders to reduce their window even in situations without congestion. TCP Westwood by Mascolo et al. [112] was one of the first congestion control algorithms to address this problem. Instead of a multiplicative congestion window reduction, TCP Westwood adjusts the congestion window and the slow start threshold according to the estimated path capacity, which is calculated by the rate of returning ACKs. TCP Westwood+ [41] corrects some of the inaccuracies in TCP Westwood's rate estimation in the presence of ACK compression [172] (when ACKs return in bursts).

TCP Veno is an approach proposed by Fu et al. [58], which takes its name from how it extends TCP Reno with ideas from TCP Vegas. Similar to TCP Illinois, TCP Veno interprets an unusually high RTT as a sign of congestion. In that case, standard Reno congestion control is applied. However, if the RTT is normal, TCP Veno assumes that bit errors must be the reason for loss. In that case, the congestion window is not halved, but only decreased by 20%. Caini et al. [31] introduced TCP Hybla to improve the throughput over long-latency, wireless networks, such as satellite connections. TCP Hybla's congestion window rules are scaled by the RTT, so that flows with different RTTs all experience an equal window growth. Both algorithms ignore the reality of link-layer retransmissions that translate corruption into latency.

High-Speed TCP

High-speed TCP versions try to utilize the available capacity over paths with a high bandwidth-delay product (BDP) more efficiently, while remaining fair to connections with a lower bandwidth-delay product. There are two aspects to why TCP's standard AIMD algorithm fails to fully utilize high-BDP paths. First, compared to halving a small `cwnd` after a congestion event, halving a large `cwnd` requires a much longer phase of additive increase to get the throughput back to full capacity. Second, a large RTT additionally leads to a slow window growth because TCP increments its `cwnd` only once per RTT.

A pioneering method for tackling these problems is Sally Floyd's HighSpeed TCP (HSTCP), published as RFC 3649 [53], which proposes an exponential `cwnd` growth. Leith et al. [101] introduce a similar approach called H-TCP, which makes the `cwnd` growth rate additionally dependent on the elapsed time since the last segment loss. The Scalable TCP approach suggested by Tom Kelly [93] aims at providing a capacity-independent, constant number of RTTs for doubling the window by modifying the `cwnd` growth from the usual 1 segment per RTT to 1 segment per 100 ACKs.

Many such high-speed TCP variants suffer severely from *RTT unfairness*, so that connections with a small RTT achieve a much higher share of the capacity (because their `cwnd` grows quicker). Binary Increase TCP (BIC), introduced by Xu et al. [167], tries to remedy RTT unfairness over high-speed paths by locating the congestion window through a binary search between a minimum window size (for which no loss was observed) and a maximum window size (for which loss occured). In contrast to other high-speed TCP variants, BIC TCP logarithmically reduces the `cwnd` growth rate as it approaches the available capacity. CUBIC TCP by Ha et al. [63] is an improvement over BIC TCP that makes its window growth depend only on the elapsed time between congestion events.

A special instance of high-speed optimized congestion control is TCP Illinois by Liu et al. [104], which uses packet loss to detect congestion, but adapts the window in proportion to estimated queuing delay. Similar to BIC and CUBIC, this results in a fast window growth when congestion is not imminent and a slow growth when congestion is likely.

5.1.5 Other TCP Mechanisms

There are several other improvements to TCP that can be used to further enhance congestion control algorithms and TCP's loss recovery.

Selective Acknowledgements (SACK)

With cumulative acknowledgments, segments that are received after a loss cannot be acknowledged until the loss is recovered. As a consequence, a TCP sender must either wait an entire RTT to detect a lost segment or risk an unnecessary retransmission. Selective acknowledgments (SACK) [114] enhance TCP's way of acknowledging data by providing additional information to the sender about missing segments at the receiver. Obtaining additional knowledge in the option field of ACKs, a sender can make more efficient decisions about *which* segments to retransmit, reducing the number of spurious retransmissions. A study by Fall et al. [49] shows the benefits of using SACK over non-SACK implementations of TCP Tahoe, Reno and New Reno.

The usage of SACK is negotiated between the sender and receiver in a `SYN` segment during connection establishment. If both agree to use SACK, the receiver can then acknowledge a list of non-contiguous blocks of data, while the semantics of the `ACK` field and the acknowledgement number in the header remain unchanged. Each SACK block is defined by two sequence numbers, a left and a right edge. Due to the limit of 40 bytes for TCP options, a maximum of four SACK blocks can be conveyed per ACK. As shown in Figure 5.4, the most recently received segment is always reported as the first block.

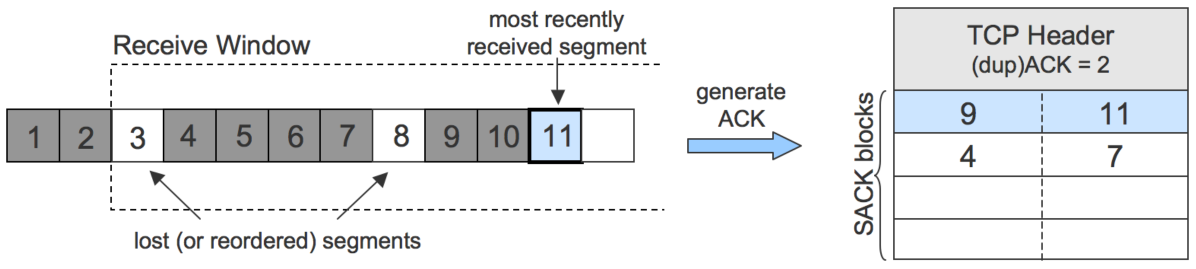


Figure 5.4: *Example of a TCP receiver generating an ACK with blocks of Selective Acknowledgements (SACK) in the option field.*

A SACK-enabled sender is expected to keep a *retransmission queue*, a list of all segments that were sent but not yet acknowledged. When receiving an ACK that includes SACK blocks, the sender uses a *scoreboard* data structure to mark segments in the retransmission queue as *SACKed*. Unmarked segments previous to the highest SACKed segment are most likely lost and therefore available for retransmission. RFC3517 [25] specifies a conservative SACK-based loss recovery algorithm for TCP that uses SACK information and a scoreboard to properly estimate the outstanding data in the network.

Forward Acknowledgements (FACK)

The Forward Acknowledgement algorithm [113] was developed as one of the first methods that try to benefit from the information provided to TCP senders in SACK blocks. It aims at more efficiently recovering in the case of multiple segment losses, which TCP Reno handles poorly. FACK takes its name from the way the most forward SACKed sequence number, denoted as `snd.fack`, is used to provide a better view of the network. With the knowledge of how many segments are missing, FACK is able to recover from large sequences of consecutive packet losses by considering all unacknowledged segments between SACK blocks as lost.

In the example of Figure 5.5(a), a burst of five segments is lost in the network and a subsequent segment carries SACK information indicating a large gap of outstanding data. In this case, although only a single dupACK is returned to the sender, FACK deduces that five segments are lost and triggers a fast retransmit. Similarly, in the more complex example of Figure 5.5(b), where several nonconsecutive segments are missing, FACK counts everything between SACKed blocks as lost.

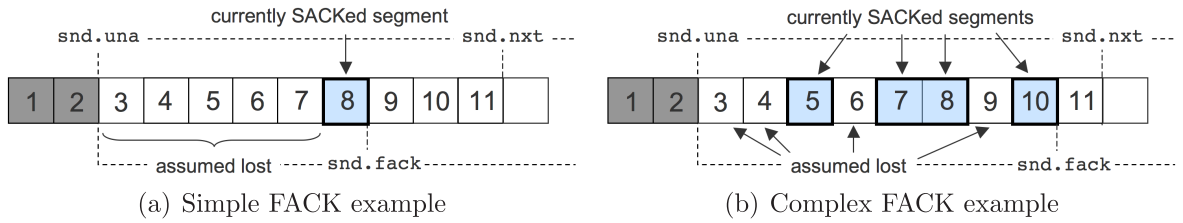


Figure 5.5: *Forward Acknowledgements (FACK) illustrated in two examples. FACK triggers a fast retransmit if the number of "holes" is larger than the dupACK threshold.*

In case the unacknowledged holes are not actually caused by packet loss, but by packet reordering, the behavior of FACK is overly aggressive. Operating systems that implement FACK, such as Linux, therefore refrain from using FACK after detecting packet reordering.

Duplicate Selective Acknowledgements (DSACK)

Although SACK is useful for reporting non-contiguous blocks of successfully received data, its specification does not cover situations when duplicate segments are received. RFC 2883 [55] fills this gap by defining DSACK, a backward-compatible extension to the SACK mechanism that allows a receiver to inform the sender about duplicate segments.

A DSACK block is specified in the same way as with SACK, always as the first block in the list. Figure 5.6 depicts an example where a duplicate segment is received that is part of a larger block of non-contiguous data. In this case, the duplicate segment is reported as a single first SACKed segment, followed by the entire block it is part of, and additional blocks may follow. Note that with DSACK, duplicate segments are reported at most once. If an ACK carrying a DSACK block is lost, the sender will never learn about the segment duplication that occurred.

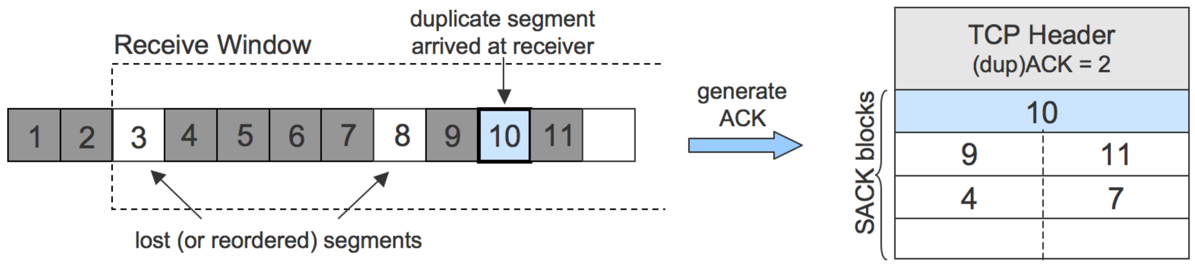


Figure 5.6: *Duplicate Selective Acknowledgements (DSACK)*

DSACK enables senders to distinguish between events that a purely SACK-enabled sender could not properly identify. However, the DSACK specification does not provide any actions a sender should implement. DSACK is able to identify the following events:

Packet replication: If a sender receives an ACK that includes a DSACK block for a segment that was never retransmitted, the DSACKed segment must have been duplicated in the network. If that ACK is in fact a dupACK, as in Figure 5.6, then the sender knows that the reason for the dupACK was packet replication, not packet loss. SACK would not provide this information, because it does not report duplicate segments.

Spurious fast retransmits: If a sender receives an ACK with a DSACK block for a segment that was already retransmitted, the DSACKed segment was most likely not lost, but reordered (causing an unnecessary fast retransmit). The SACK mechanism alone would not report the second instance of the segment as duplicate, causing the sender to wrongly believe a packet was lost.

Spurious retransmission timeouts: Retransmission timeouts can occur even in the absence of loss if the RTO is set too small. In this case, the sender will retransmit the delayed segments under the assumption that their originals were lost. Shortly after the timeout, ACKs to the original segments return without any SACK blocks included, because everything is correct from the receiver's perspective. However, the retransmissions will finally trigger DSACKs from which the sender can learn that a timeout was wrongly triggered.

Loss of all ACKs in a window: If all ACKs in a window of successfully delivered segments are lost, a timeout will occur. The first ACK received after the timeout will carry DSACK information that would not be conveyed by the SACK mechanism alone.

Note that DSACK is unable to avoid the retransmission of a full window of segments after the occurrence of a spurious timeout. A DSACK-enabled sender is only able to identify a timeout as spurious *one RTT after* the spurious retransmission occured. That is at the time when the ACK of the first (spuriously) retransmitted segment arrives at the sender. During the elapse of that RTT, the receiver is unaware that an entire window of segments is continuously causing timeouts. This drawback is addressed by the Forward RTO Recovery algorithm described in the following.

Forward RTO Recovery (F-RTO)

TCP interprets the expiration of the retransmission timer as segment loss. However, even on the most reliable links, timeouts can occur due to perfectly normal delay spikes (e.g, during cellular network handoff) and can cause TCP to enter slow start. Even worse, a *spurious* timeout typically causes the entire window of currently in-flight segments to be needlessly retransmitted, because their ACKs also arrive too late at the sender.

The F-RTO algorithm, proposed by Sarolahti et al. [143] is specified in RFC 5682 [144] and can be used by TCP senders to detect such spurious retransmission timeouts and avoid that a whole window is needlessly retransmitted. When TCP retransmits a segment after a timeout occurs, the main idea of F-RTO is to start sending new data instead of retransmitting more old segments. At the same time, F-RTO monitors the next two incoming ACKs to find out whether the timeout was spurious or not. If either of them is a dupACK, the F-RTO algorithm aborts and TCP continues with conventional RTO recovery. If the first two ACKs after the timeout both acknowledge new data, the F-RTO algorithm identifies the timeout as spurious and continues sending new data. In other words, F-RTO does not prevent a single spurious retransmissions from happening, but it prevents further unnecessary retransmissions after a spurious timeout.

5.1.6 TCP and IP Packet Reordering

The forwarding of IP packets over different paths, even though they belong to a single TCP connection, breaks two of TCP's basic assumptions: (i) that IP packet loss is an indication of network congestion, and (ii) that *severe* IP packet reordering is a rare event. Sending packets over multiple paths with differences in latency causes TCP's congestion control to misinterpret a large number of reordered packets as lost and to wrongly estimate the round-trip time and the available path capacity, potentially resulting in many needless retransmissions and a severely lowered data throughput. In order to avoid destructive implications to TCP's performance with large IP packet reordering, numerous mechanisms for increasing TCP's robustness to IP packet reordering have been proposed [102], some of them standardized by the IETF (such as the Eifel algorithm [107] and (TCP-NCR) [23]).

The Eifel Algorithm

Ludwig et al. propose the Eifel algorithm [107], an enhancement for TCP's performance in situations with frequent spurious retransmissions (covering both cases of spurious timeouts and spurious fast retransmits). The Eifel algorithm relies on the timestamp option [79] to find out if a segment is a retransmission or an original. For every *first* retransmission of a segment, the sender stores its timestamp. If later an ACK with the same sequence arrives that has a smaller timestamp, the sender can be sure that it acknowledges the original segment. This method of detecting whether TCP has unnecessarily entered loss recovery is also specified in RFC 3522 [108].

If a single retransmission is identified as spurious, the Eifel algorithm reverts the congestion window and slow start threshold to TCP's congestion control state *before* entering loss recovery. Although spurious retransmissions are not completely prevented like this, TCP's performance does not get unnecessarily reduced. These actions of reverting the congestion control state are also specified in RFC 4015 [106], but only for the case when the cause of the spurious retransmission was a false timeout.

TCP with Non-Congestion Robustness

TCP-NCR (Non-Congestion Robustness for TCP), documented in RFC 4653 [23], is an attempt to better disambiguate segment loss from reordering. Since three dupACKs may not be sufficient to distinguish loss from reordering, TCP-NCR postpones fast retransmit by dynamically adjusting the dupACK threshold from three to a congestion window worth of outstanding data in the network. Waiting for a congestion window of data after receiving a first dupACK roughly corresponds to giving TCP one RTT of time to detect reordered segments.

Using a larger dupACK threshold also increases the period of limited transmit, the time between receiving the first dupACK until a fast retransmit is triggered (see Algorithm 3). TCP-NCR specifies two alternative limited transmit algorithms, an *aggressive* and a *careful* limited transmit. The aggressive version is a simple extension of limited transmit, in which a new segment is sent for every incoming dupACK. With careful limited transmit, for every two incoming dupACKs, only one new segment is transmitted. Both methods are associated with a constant value LT_F, representing the fraction of outstanding data that must be SACKed before invoking fast retransmit: $LT_F = \frac{1}{2}$ for aggressive and $LT_F = \frac{2}{3}$ for careful limited transmit. The threshold for triggering fast retransmit, `dupThresh`, depends on LT_F and is calculated as LT_F times the number of segments currently in flight.

The ideas of a delayed congestion response and extended limited transmit were first mentioned by Blanton et. al [24]. TCP-NCR itself was first introduced by Bhandarkar et al. [22] under the name of TCP-DCR (Delayed-Congestion Response for TCP), which does not set `dupThresh` to a fraction of the outstanding data, but actively delays fast retransmit by a time interval of one estimated RTT. Using the ns-2 [126] simulator, the authors show that TCP-DCR significantly outperforms SACK-TCP in situations with "mild but persistant reordering" and also when 50% of all packets get extensively delayed due to multipath forwarding. However, the simulation configuration does not consider bandwidth heterogeneity and consistently ignores the first 100 seconds out of all 1100-second long simulation runs. Although this is usual practice for many simulated simulation and the results of TCP-DCR look very promising, reporting only TCP's steady-state data throughput does not say much about its performance in situations with severe packet reordering. As will be shown in Section 5.2.2, TCP may require many seconds to adapt to persistent packet reordering before reaching a high performance steady-state throughput.

A noteworthy approach similar to TCP-NCR and TCP-DCR was proposed by Bohacek et al. [26] as TCP for persistent packet reordering (TCP-PR). The authors propose to completely ignore DupACKs and instead rely on timers to detect packet loss. TCP-PR assumes a segment as lost if its acknowledgment does not arrive within a given time frame (a function of the RTT).

5.1.7 Linux TCP

In preparation of our experimental study in Section 5.2, this section provides an overview of how TCP is implemented in the Linux operating system. Linux TCP differs from the standard *TCP NewReno* and *SACK TCP* loss recovery schemes that are often available in simulators and commonly used as benchmarks in research papers. Besides these well-established TCP variants, Linux also supports the less conventional mechanisms of FACK, DSACK, the TCP timestamp option, F-RTO, and techniques to adapt to packet reordering and for undoing incorrect congestion window adjustments. Linux allows some of these mechanisms to be configured with immediate effect through simple configuration commands. Selected from a variety of approximately forty TCP-related parameters, Table 5.2 lists the tunable mechanisms we consider relevant within the scope of our research on multipath aggregation. In general, we use the term *Linux TCP* to refer to the full-featured version of TCP as it is implemented in Linux 2.6.28 and 2.6.32 (the kernel version used in our experiments), where all these mechanisms are enabled by default.

Table 5.2: Selected Linux TCP Parameters

Parameter name	Valid value range	Default value
Congestion control	12 algorithms ($\rightarrow$ Table 5.1)	CUBIC
SACK	{0, 1}	enabled {1}
FACK	{0, 1}	enabled {1}
DSACK	{0, 1}	enabled {1}
F-RTO	{0, 1, 2}	SACK-enhanced F-RTO {2}
TCP timestamps	boolean	enabled {1}
`reordering`$_{min}$	{1, 2, 3, ..., 127}	Standard dupACK threshold {3}

Even though the Linux operating system supports all fundamental TCP principles documented in the IETF's series of RFCs, it does not strictly follow their specification and provides additional performance enhancements that are not standardized. The most prominent example is Linux TCP's fine-grained RTT estimation with a minimum RTO limit of 200 ms, rather than the 1000 ms minimum imposed by RFC 2988 [127]. Another example is Linux TCP's measuring of the congestion window `cwnd` in units of full-sized segments, not in bytes as specified in RFC 5681 [9]. In addition to these rather simple differences, Linux is unique in its way of performing loss recovery, in its variety of supported congestion control algorithms, and how it attempts to achieve robustness against

IP packet reordering. These mechanisms are outlined in the following paragraphs, partly based on discussions by Sarolahti et al. [145] and Järvinen et al. [83].

Loss Recovery in Linux TCP

For the purpose of recovering lost data segments and for deciding how much new data can be injected into the network, Linux TCP keeps track of the number of outstanding segments in the network. The number of outstanding segments are calculated based on SACK information, which provide a clear picture of holes in the ACKed data. Linux TCP counts the number of outstanding segments (`in_flight`) by adding all sent segments (`packets_out` = `snd.nxt - snd.una`) and all currently retransmitted segments (`retrans_out`), but subtracting all SACKed segments (`sacked_out`) and all those that are considered lost (`lost_out`). In summary:

```
in_flight = packets_out + retrans_out - sacked_out - lost_out
```

Among these four values that constitute the number of data segments in flight, the three first values are easily computed when data is sent and when SACK information arrives in returning ACKs. Only the number of lost packets `lost_out` is uncertain, because it is not always clear whether holes between SACKed data blocks are actually lost or unusually delayed (reordered). The way `lost_out` is estimated depends on the loss recovery scheme. As shown in Figure 5.7, Linux TCP senders can be in three modes of loss recovery. If both SACK and FACK are enabled (the default in kernels 2.6.32), an aggressive FACK-based loss recovery is employed that counts all unacknowledged data between SACKed blocks as lost. If SACK is enabled but FACK is disabled, the conservative SACK-based loss recovery specified by RFC 3517 is used, which considers unacknowledged data as outstanding. The same conservative scheme is used if reordering is detected, because FACK fails in the presence of packet reordering. If SACK is disabled or unsupported by one of the endpoints, Linux TCP falls back to an equivalent of TCP NewReno.

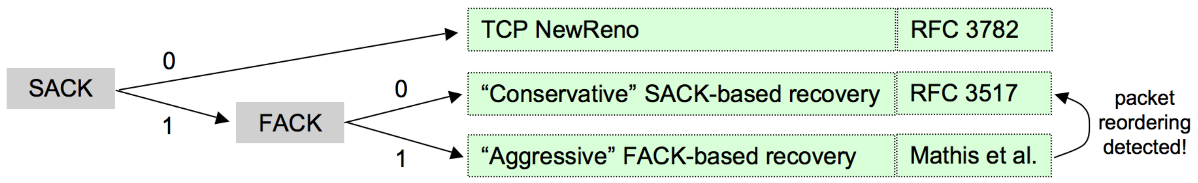

SACK
0
1
FACK
0
1
TCP NewReno
RFC 3782
"Conservative" SACK-based recovery
RFC 3517
"Aggressive" FACK-based recovery
Mathis et al.
packet reordering detected!

Figure 5.7: *The loss recovery scheme used by Linux TCP depends on whether SACK and FACK are enabled or disabled.*

The Linux TCP `reordering` Metric

Linux TCP contains several mechanisms that target an improved robustness against IP packet reordering, including SACK, DSACK, and an approach similar to the Eifel algorithm that makes use of the TCP timestamp option. When Linux TCP finds out that a presumably lost packet in fact reordered, it updates its `reordering` metric to the number of segments between the cumulatively acknowledged segment (the one detected as reordered) and the highest selectively acknowledged segment.

As its name implies, the `reordering` metric is Linux TCP's estimated magnitude of packet reordering and represents its tolerance to how many segments may be reordered without triggering a fast retransmit and entering loss recovery. It is by default set to 3 (the standard dupACK threshold) and can grow to a maximum value of 127. In Linux, it is possible to change the minimum value through a command-line instruction. We refer to this value as $\texttt{reordering}_{min}$. Whenever Linux TCP experiences a retransmission timeout, the `reordering` metric is reset to $\texttt{reordering}_{min}$.

Congestion Control in Linux TCP

Linux supports all congestion control algorithms listed in Table 5.1 with CUBIC set as default since kernel version 2.6.19. In addition to using a non-standard congestion control by default, Linux TCP employs a variant of the *rate halving* algorithm during fast recovery. Rate halving, originally proposed by Hoe et al. [68] does not inflate the congestion window during fast recovery, nor does it abruptly reduce it to half the size when fast recovery ends. During fast recovery, just as always done by Linux TCP, the congestion window is updated to represent the amount of outstanding data in the network. In addition, for every other acknowledgement received during fast recovery, a new data segment can be sent (preserving TCP's self-clocking). A positive side-effect of rate-halving is that congested router queues are drained immediately when congestion is detected, not after the loss recovery, which results in fewer multiple losses.

5.2 Analysis of TCP's Robustness to Multiple Paths

In this section, we investigate the behavior of TCP when packet-based multipath forwarding is used at the network layer. In our experiments, every TCP connection between the sender and the receiver is established with only a single IP address per endpoint. IP packet forwarding is performed transparently to TCP, no modifications are made to the TCP protocol itself. We base our study on Linux TCP, which is of particular interest to our research as it contains several performance-enhancing mechanisms that target an improved robustness against packet reordering. In addition, the Linux operating system accommodates for a convenient configuration of many TCP-related parameters, making it possible to explore the effects of a large number of settings in an automated manner.

The results of our analysis are presented in the following steps. First, we present experiments using two heterogeneous paths in a practical WLAN/HSPA-based scenario. Second, we try to reproduce the results obtained from real wireless networks by emulating the same networks in a controlled environment. Third, we analyze the available parameters that can be specified in Linux to control certain mechanisms, such as SACK, FACK, DSACK, etc. This allows us to approximate the functionality of standard TCP NewReno and SACK TCP and compare their performance to Linux TCP. Additionally, with knowledge of which parameter settings have a positive effect on TCP's multipath performance, we suggest fine-tuned Linux TCP configurations that achieve a higher aggregation benefit.

5.2.1 Linux TCP over Practical WLAN and HSPA Networks

In an initial experiment, we want to satisfy our curiosity about how TCP's data throughput is affected when IP packets are transparently forwarded over two practical access networks of different wireless technologies. As shown in Figure 5.8, we use a download scenario with a receiving machine running Linux version 2.6.32 connected simultaneously to the WLAN$_1$ (IEEE 802.11b) and the HSPA$_1$ network. As previously measured in Chapter 4, these two networks are highly heterogeneous in terms of path latency, which leads to severe packet reordering even at very low transfer rates. The RTT over the WLAN path is typically around 10 ms, while HSPA's average latency alternates over time between around 100 and 250 ms.

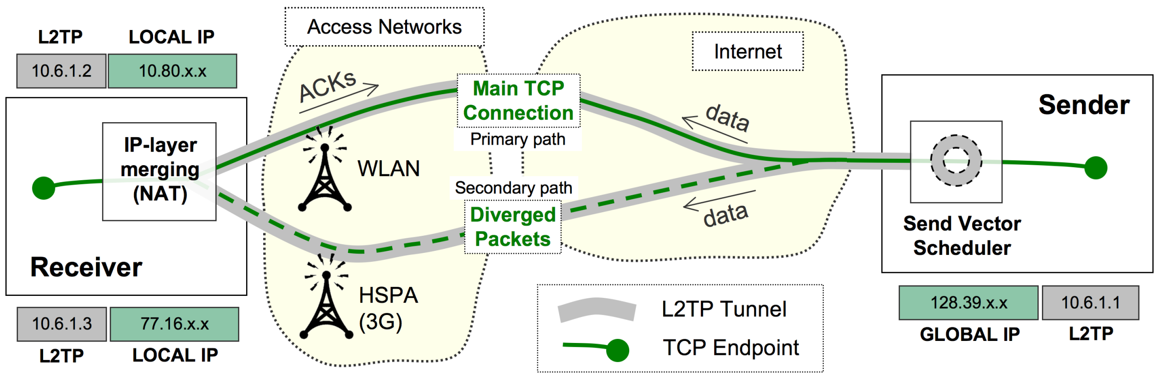


Figure 5.8: *L2TP-based testbed configuration for mulitpath TCP experiments over practical WLAN and HSPA access networks.*

For this experiment, we use a slimmed-down version of the IP-layer NAMA solution that we introduced in Chapter 4. To reduce unpredictable side-effects, we exclude NAMA's path monitor and the delay equalizer modules, which leads to the experimental testbed consisting of the following basic components:

1. **Static IP Packet Scheduling** Following the *send vector* approach introduced in Section 3.5, IP packets are scheduled in a weighted round-robin fashion, proportional to the paths' estimated throughput. TCP data segments reach the receiver on the path decided by the scheduling module, while ACKs always return to the sender over the primary path. We refer to the path corresponding to an established TCP connection as the *primary* path, while the corresponding receiver interface is called the primary interface. The terms "primary" and "secondary" do not imply any path characteristics, they only designate the reverse path for ACKs.

   Although ACKs could potentially be sent over multiple paths, too, there are several reasons why we use only a single path for them. First, ACKs consume a small share of the capacity and therefore require no multipath aggregation. Second, sending ACKs over a single path avoids reverse-path packet reordering (which, as shown by Leung et al. [102], causes additional challenges to TCP that we may consider in future work). Last, but not least, sending ACKs only over the primary path reduces the need for a scheduler at the client. If, for example, ACKs were supposed to return on the same path their corresponding data segments came from, a much more complex scheduler would be necessary that keeps track of which TCP sequence numbers came through which interface.

2. **L2TP Tunneling** After the scheduler at the TCP sender has selected path for a given IP packet, a network-layer module transparently forwards the packet by writing the IP address of the receiving interface into the destination field. At the receiver, TCP segments that arrive on the secondary interface are merged back into the original flow by a preconfigured network address translation (using `iptables` rules). In order to prevent firewalls in access networks to filter out IP packets that appear as spoofed, we encapsulate all traffic in L2TP [160] tunnels. As opposed to the IP-in-IP tunneling solutions proposed in [34] and [131], L2TP is applicable to our practical WLAN/HSPA scenario, because it uses NAT-punching to reach local interface addresses.

   Using this combination of NAT and L2TP tunneling, a single TCP connection can be sent transparently to multiple interfaces with TCP remaining completely unaware of the fact that segments travel over different paths. TCP only experiences the resulting side effect of segments getting out of order.

For comparing the performance of single- and multipath TCP, we perform an experiment that is divided into 36 rounds. Each round consists of four consecutive 60-second TCP connections (two singlepath and two multipath measurements), established with the `iperf` tool, between the sender and the multihomed receiver. The first two TCP connections in each round are individual throughput measurements of the WLAN and HSPA networks and serve as benchmarks for the comparison. In addition, these first two throughput measurements are used for configuring the scheduler's send vector to be used for the two subsequent measurements. The latter two multipath experiments, which are run using both wireless paths simultaneously, are conducted in random order. In one experiment, the WLAN interface is used as the primary interface (denoted as "WLAN$\oplus$HSPA"), while in the other, the HSPA interface is used as the primary interface (denoted as "HSPA$\oplus$WLAN").

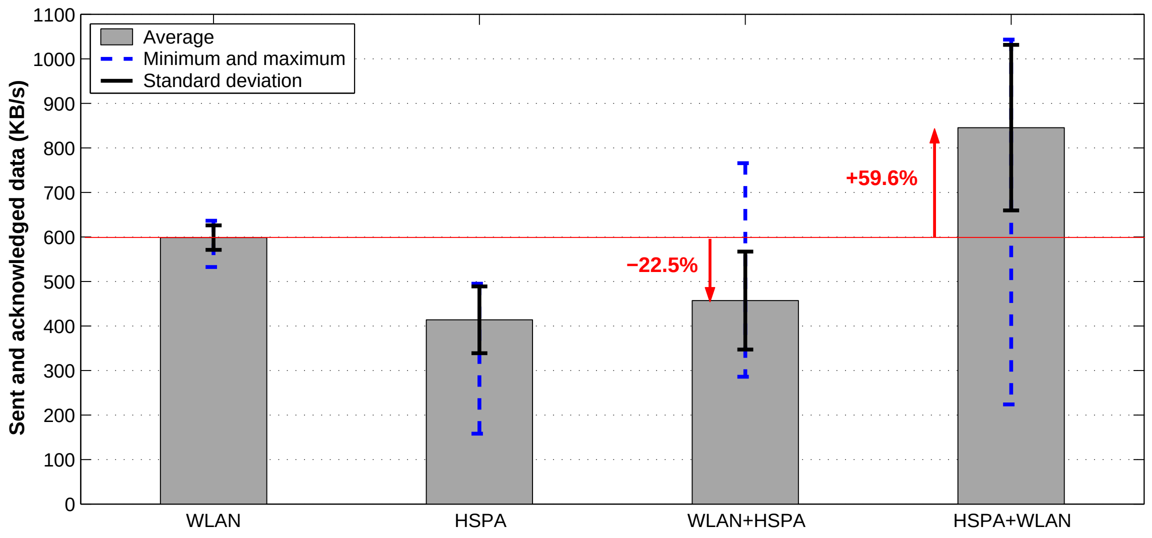


Figure 5.9: *TCP's aggregation benefit over combined WLAN and HSPA networks (the notation X+Y implies that X is the primary and Y is the secondary path).*

Figure 5.9 shows the results of this experiment with each bar representing the average of the 36 rounds of measurements. The average data throughput over a single WLAN connection was 591 KB/s, while HSPA achieved an average of 414 KB/s. With WLAN as the primary path (WLAN⊕HSPA), the aggregated throughput was only 458 KB/s, less than what a single WLAN path is able to transfer, and corresponding to an aggregation benefit of -22.5%. On the other hand, configuring HSPA as the primary interface (HSPA⊕WLAN) resulted in a significant performance increase. With HSPA as primary, the average throughput resulted in 838 KB/s, i.e., an aggregation benefit of 59.6%.

5.2.2 Linux TCP over Emulated WLAN and HSPA Networks

In order to get a better understanding of the previous results obtained from practical experiments with WLAN and HSPA networks, we attempt to reproduce the same results in a controlled, emulated network environment. Illustrated in Figure 5.10, the emulated testbed setup contains the same modules for IP packet scheduling, but manages without the use of L2TP tunneling, since the IP addresses of all involved interfaces belong to the same local test network.

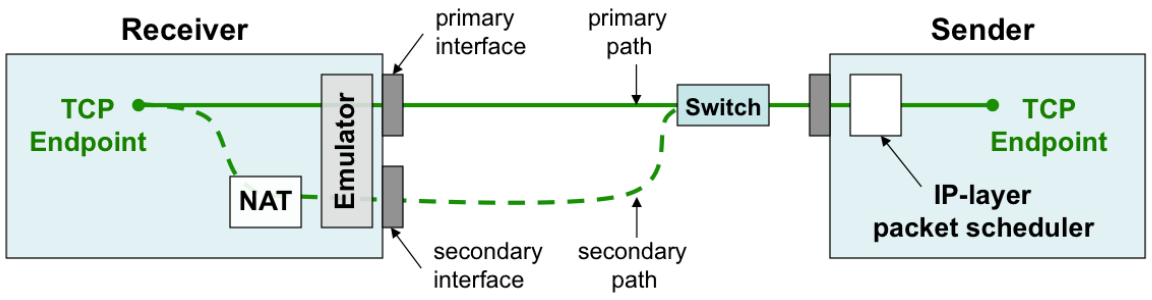


Figure 5.10: *Network emulation testbed for multipath TCP experiments.*

In this experiment, the two paths between the TCP sender and receiver are configured to the approximate properties of the real WLAN and HSPA networks. The emulated WLAN path is set to a bandwidth of 600 KB/s and a base RTT of 10 ms. The HSPA path is modeled with a bandwidth of 400 KB/s and a base RTT of 100 ms. In previous measurements, we observed excessively large RTTs under heavy load in the real WLAN and HSPA networks (some results are shown in Figure 3.5). As this is a strong indication of large buffers in the practical access networks, we intentionally use large queues of Q = 200 packets in the emulation, too. In these experiments, no additional random packet loss is introduced (i.e., $L_{pri} = L_{sec} = 0$). However, packet loss is naturally expected due to TCP probing and queue overflows.

Figure 5.11 presents the results of Linux TCP's performance when IP packets that belong to same flow are forwarded over WLAN and HSPA networks simultaneoulsy. The figure compares the practical and emulated experiments and shows very similar trends in both cases. Using the WLAN network as the primary path (WLAN⊕HSPA) results in a considerable performance loss (an average of -22.5% with the real networks and -53% in the emulated scenario). The opposite scenario of using HSPA as the primary path (HSPA⊕WLAN) is able to achieve a large aggregation benefit (an average of 59.6% with the real networks and 75% in the emulated scenario).

The large performance disparity between the two multipath scenarios, WLAN⊕HSPA and HSPA⊕WLAN, is caused by Linux TCP's ability to adjust much faster to packet

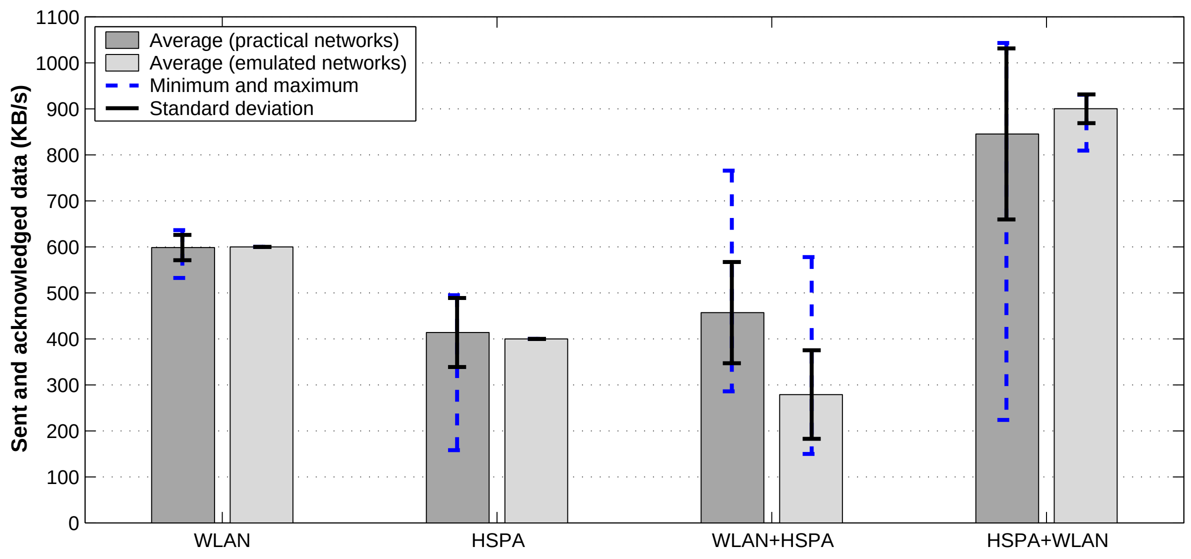


Figure 5.11: *Linux TCP-based multipath aggregation over practical and emulated WLAN and HSPA networks.*

reordering in the HSPA⊕WLAN scenario. In this case, it usually takes less than a few seconds for Linux TCP to detect the high magnitude of reordering and the aggregated throughput quickly reaches a value close to the optimum. On the other hand, in the WLAN⊕HSPA scenario, Linux TCP requires a very long time to detect packet reordering, making it unable to efficiently aggregate the two paths. Figure 5.12 illustrates this phenomenon using emulation-based experiments with much longer TCP connections that last up to 30 minutes. In the HSPA⊕WLAN scenario, Linux TCP is able to reach its full performance for connections of any duration. However, in the WLAN⊕HSPA scenario, it requires a much longer connection duration for aggregation to become beneficial. For example, a 60-second TCP connection over WLAN⊕HSPA results in an average throughput of about 250 KB/s, while close to 900 KB/s are reached in the opposite scenario. In the scenario with WLAN as the primary path, an average connection duration of about 9 minutes is necessary for the aggregated throughput to surpass the bandwidth of the slower HSPA path, while not even 30 minutes are enough to surpass the bandwidth of the faster WLAN path and to reach an aggregation benefit.

The reason for this behavior is specific to Linux TCP and lies in its ability to learn about packet reordering and to update its `reordering` metric when a new maximum reorder extent is discovered. When examining individual WLAN⊕HSPA experiments, it becomes clear that Linux TCP's adjustment to reordering does not result in a smooth growth, but in a step-wise increase of the aggregated throughput. Figure 5.13 shows this behavior based on three selected experiments. In the case of HSPA⊕WLAN, we always observed the same development of the aggregated throughput reaching about 900 KB/s within a few seconds and sustaining this level for the entire 30-minute duration in a somewhat instable but consistent manner. In the case of WLAN⊕HSPA, the progress of the aggregated throughput differs with every new connection. It generally increases at seemingly random points in time, in multiple occasions, until finally the same maximum level of about 900 KB/s is reached and sustained for the rest of the connection. In the most extreme cases that we measured, Linux TCP required hours to reach a steady state.

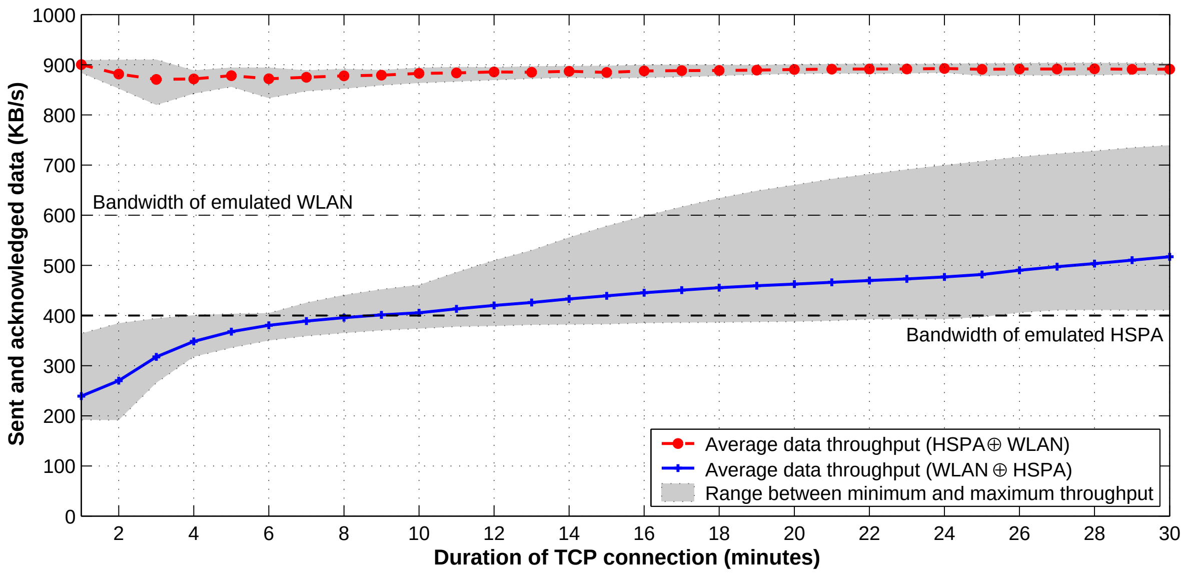


Figure 5.12: *The performance of TCP-based multipath aggregation as a function of connection duration (averages of ten 30-minute runs).*

As depicted by Figures 5.12 and 5.13, Linux TCP's learning process and adaptation to multipath forwarding is very quick in the HSPA⊕WLAN scenario, but lasts for many minutes in the case of WLAN⊕HSPA. Due to the definition of a primary path, over which all ACKs are returned, the two scenarios are asymmetric in their latency characteristics. If WLAN is primary, it takes an RTT of either 10 or 55 ms for a data segment to get acknowledged, because the one-way delays on the WLAN and HSPA paths are 5 ms and 50 ms, respectively. If HSPA is primary, the RTT is either 100 or 55 ms. This asymmetry results in performance differences between the scenarios WLAN⊕HSPA and HSPA⊕WLAN, for which we have the following explanations:

1. **Hindered `cwnd` growth** As shown in Table 5.3, Linux TCP fails to grow its `cwnd` in the WLAN⊕HSPA scenario. On average, the congestion window remains an order of magnitude smaller than in the HSPA⊕WLAN and single-path scenarios. We attribute this difference to the fact that in the WLAN⊕HSPA scenario, segments that are sent over the low-latency WLAN path result in "immediate" ACKs at the sender, leading to a severe underestimation of the RTT and the available capacity in the network. On the other hand, in the HSPA⊕WLAN scenarios, segments that are sent over the low-latency WLAN path take a considerable time (about 55 ms) until they get ACKed.

2. **Hindered reordering quantification** When Linux TCP detects packet reordering, the `reordering` metric is increased to a value that depends on the current number of packets in flight (Linux TCP's equivalent to the congestion window). In the WLAN⊕HSPA scenario, where the congestion window is very small on average, the `reordering` metric thus hardly manages to grow. According to Equation 4.2, an estimated reordering of about 120 packets can be expected in either of these multipath experiments. However, Table 5.3 shows that the average `reordering` metric is far below this value in the WLAN⊕HSPA scenario.

3. **ACKs overtaking original transmission** The WLAN⊕HSPA scenario allows for

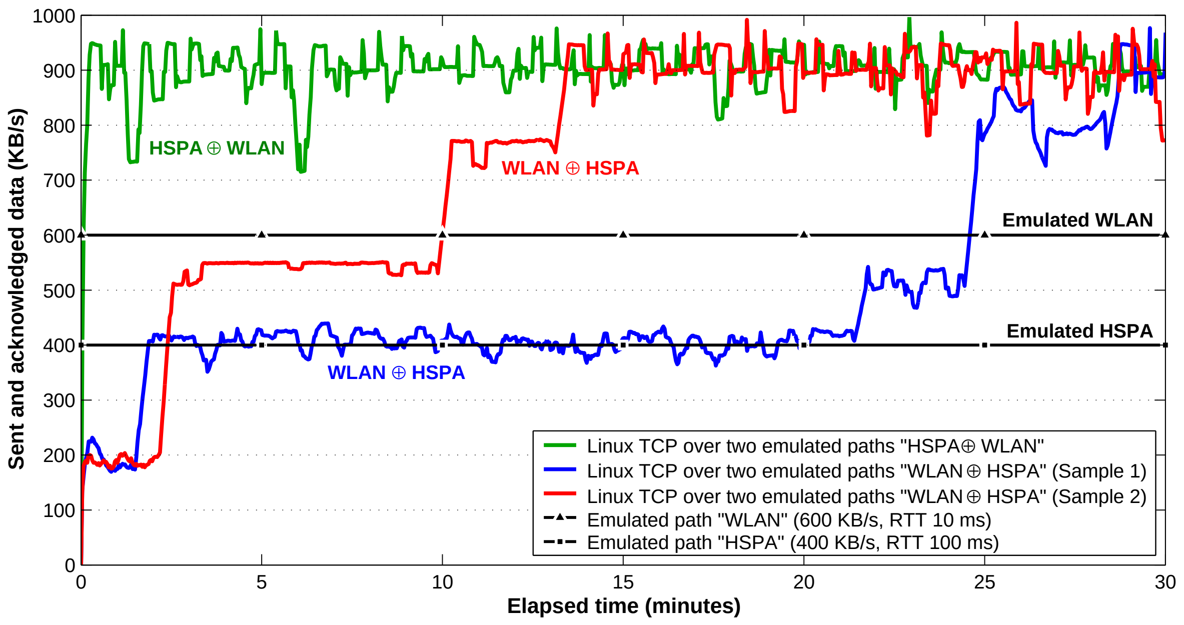


Figure 5.13: *Linux TCP's step-wise adjustment to IP packet reordering caused by packet-based forwarding over emulated WLAN and HSPA paths (independent 30-minute samples).*

a peculiar sequence of events that results in a retransmitted packet to overtake its original transmission. During the time when a segment A travels over the high-latency HSPA path, it is possible that segments B, C, and E get quickly (dup)ACKed over the WLAN path (segment D is also scheduled over HSPA). As soon as the initial dupACK threshold of 3 is reached, segment A gets retransmitted as A_{ret} and arrives at the receiver before A. This unusual event results in *SACK reneging*, which is treated by the sender similarly to an RTO timeout. As a side-effect, the `reordering` metric is set back to 3 and the same series of events begins all over again, making it difficult for TCP to grow its congestion window and increase its estimate of reordering. On the other hand, the same series of events is highly unlikely to happen in the HSPA⊕WLAN scenario, because no one-way trip from the sender to the receiver is longer than any of the possible round trips, meaning that no retransmission can overtake its original.

We made two interesting observations that support the above explanations. First, by disabling SACK, and preventing SACK reneging to occur, the stepwise behavior in the WLAN⊕HSPA scenario is removed. Without the SACK mechanism Linux TCP manages to quickly aggregate both paths just like in the HSPA⊕WLAN case. Second, manually increasing $\texttt{reordering}_{min}$ during an ongoing connection over WLAN⊕HSPA causes Linux TCP to immediately reach its stable state in one large step.

5.2.3 The Multipath Performance of Various TCP Flavors

After the previous experiments showed the performance of Linux TCP in practical and emulated scenarios with WLAN and HSPA networks, it is important to explore a much larger variety of emulated path combinations. In this section, we search for an answer to the question of how a modern, full-featured TCP implementation compares to the traditional TCP algorithms when faced with severe IP packet reordering. To this end,

Table 5.3: Average Numbers from the HSPA/WLAN Multipath Experiments

	`cwnd`	`reordering`	`srtt`	`rttvar`	`rto`	time in loss recovery
	(packets)		(milliseconds)			
WLAN (practical)	246	3	576	26	920	5%
HSPA (practical)	267	3	965	18	1295	11%
WLAN⊕HSPA (pract.)	34	30	95	9	312	27%
WLAN⊕HSPA (emul.)	6	12	53	3	253	31%
HSPA⊕WLAN (pract.)	155	105	287	11	512	16%
HSPA⊕WLAN (emul.)	236	92	404	7	609	13%

we configured Linux TCP in a way that approximates the standard TCP versions commonly refered to as *TCP NewReno* or *SACK TCP*. Our understanding, and the general conception in the research community [102, 9], is that standard TCP performs “poorly” in a multipath scenario, especially if IP packets are persistently reordered more than the standard dupACK threshold of 3. In addition, it is widely accepted that enhanced mechanisms, such as DSACK and TCP timestamps, allow a “better” performance in situations with high IP packet reordering [102]. However, there are no measurement-based studies that scrutinize these claims and that try to quantify the terms “poorly” and “better”. Our following experiments are an attempt to fill this lack of knowledge.

The many possible combinations of congestion control algorithms, loss recovery schemes and performance enhancing mechanisms result in large variety of different TCP versions, also referred to as TCP *flavors*. Common examples include TCP Tahoe, Reno and NewReno, which we described in detail in Section 5.1.3, and SACK TCP, which was introduced in Section 5.1.5. By a proper configuration of the available TCP parameters in Linux, it is possible to model some of these TCP flavors. Table 5.4 lists the TCP parameter settings that result in the approximate behavior of the well-known flavors of TCP NewReno and SACK TCP. As mentioned earlier, *Linux TCP* represents the default mode of operation in the current Linux operating system, with all mechanisms enabled by default. In addition, we define *Eifel TCP* as TCP NewReno with a timestamp-based packet reordering detection, but no SACK.

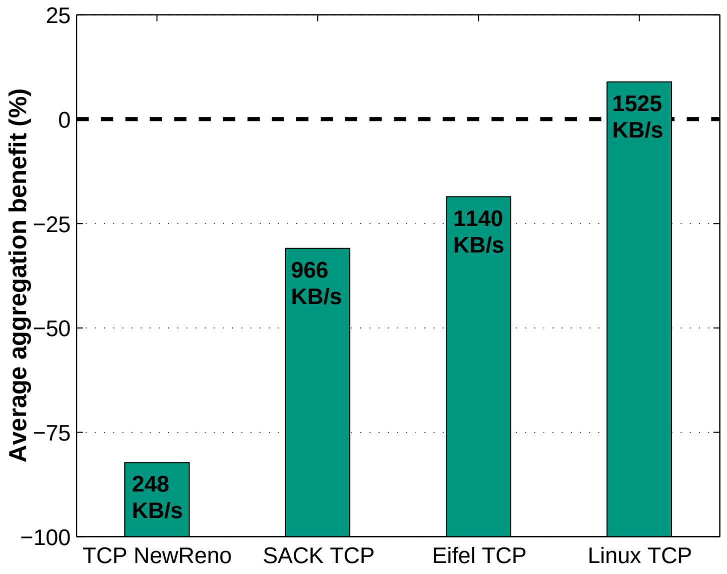
Table 5.4: TCP Flavors Modeled by Linux TCP Parameters

	TCP NewReno	SACK TCP	Eifel TCP	Linux TCP
Congestion control	NewReno	NewReno	NewReno	CUBIC
SACK	0	1	1	1
FACK	0	0	0	1
DSACK	0	0	0	1
F-RTO	0	0	0	2
TCP timestamps	0	0	1	1
$\texttt{reordering}_{min}$	3	3	3	3

In order to answer the questions of how well a full-featured TCP flavor, such as Linux TCP, handles severe IP packet reordering, and how it compares to TCP NewReno, SACK TCP and Eifel TCP, we carried out an extensive experiment including four sets of 10000

TCP connections. Each connection lasted for 60 seconds and IP packets were transparently scheduled over two paths that were arbitrarily configured using network emulation. While the queue length was kept at 1000 packets and almost no additional packet loss was introduced, the bandwidth values of BW_{pri} and BW_{sec} were randomly and independently chosen from a range of 13 equally spaced values between 300 and 2100 KB/s, resulting in scenarios with a wide variety in terms of bandwidth heterogeneity "$BW_{pri} \oplus BW_{sec}$". To avoid certain scenarios of path heterogeneity to occur more frequently than others, we ensured that the frequency of all combinations of $BW_{pri} \oplus BW_{sec}$ is approximately equal. Analogously, the RTT characteristics RTT_{pri} an RTT_{sec} were selected from a range of values between 10 and 250 milliseconds.

Figure 5.14 contains a summary of the parameters used in the emulation and the results of this experiment. Each bar represents the average multipath performance of a TCP flavor, fairly based on the same 10000 network emulation configurations. These results obviously show how standard TCP NewReno fails to adapt to IP packet reordering caused by multipath forwarding compared to more advanced loss recovery mechanisms. In the majority of emulated multipath scenarios, TCP NewReno was unable to achieve a throughput higher than $max(BW_{pri}, BW_{sec})$ and resulted in an average aggregation benefit of -81%. Our results support the claim of "poor" behavior of standard TCP NewReno under severe path heterogeneity.



Emulation Parameters	
BW_{pri}	300, ..., 2100 KB/s
BW_{sec}	300, ..., 2100 KB/s
RTT_{pri}	10, 30, ..., 250 ms
RTT_{sec}	10, 30, ..., 250 ms
L_{pri}	2^{-28} %
L_{sec}	2^{-28} %
Q	1000 packets

Figure 5.14: *The multipath performance of various TCP flavors.*

Compared with TCP NewReno, the performance of SACK TCP and TCP Eifel resulted in much higher average aggregation benefits of -28% and -19%, respectively. In both cases, Linux manages to detect packet reordering and adapt its `reordering` metric, which allows an improved performance. With all performance-enhancing mechanisms enabled, Linux TCP is able to achieve a slightly positive aggregation benefit of 9%, even under such a wide variety of randomly chosen path characteristics.

5.2.4 Exploring the Entire Linux TCP Parameter Space

In the previous section, we configured TCP-related parameters in Linux to approximate the behavior of well-known TCP flavors, such as TCP NewReno. Our results indicated that default Linux TCP significantly outperforms standard TCP. Linux' possibility of tuning TCP parameters raises the question of which parameters are relevant for multipath performance, and whether there are unknown parameter combinations that lead to enhanced performance. In this section, we therefore expand the parameter space of our network emulation to as many parameters as possible. By looking at the effects of less conventional mechanisms supported in Linux, such as FACK, DSACK, and F-RTO, we hope to improve Linux TCP's overall multipath performance.

Table 5.5 lists the complete set of parameters used in this study, divided into seven network emulation parameters and seven TCP-related parameters. The value range of the TCP parameters are given by the Linux operating system. However, certain combinations of TCP mechanisms are redundant. For example, if SACK is disabled, then DSACK, FACK, and SACK-based F-RTO are ineffective, even if these mechanisms are all enabled. We therefore made sure to avoid such invalid configurations in our experiments.

Table 5.5: Parameters for the Orthogonal Design of Experiment

	Parameter name	Test levels
Network	Primary capacity BW_{pri} (KB/s)	300, 450, 600, ..., 1950, 2100
	Secondary capacity BW_{sec} (KB/s)	300, 450, 600, ..., 1950, 2100
	Primary round-trip time RTT_{pri} (ms)	10, 30, 50, ..., 230, 250
	Secondary round-trip time RTT_{sec} (ms)	10, 30, 50, ..., 230, 250
	Primary loss probability L_{pri} (%)	2^{-x}, with $x \in \{4, 6, 8, ..., 26, 28\}$
	Secondary loss probability L_{sec} (%)	2^{-x}, with $x \in \{4, 6, 8, ..., 26, 28\}$
	Emulation queue length Q (packets)	$x * BDP + 5$, with $x \in \{1, 2, 3, ..., 13\}$
TCP	TCP congestion control	12 algorithms (→ Table 5.1)
	SACK	0, 1
	DSACK	0, 1
	FACK	0, 1
	F-RTO	0, 1, 2
	TCP Timestamp option	0, 1
	`reordering`$_{min}$	13 log-scaled values from 3 to 127

Another issue in the design of our experiments is that the value range of the network emulation parameters are unbounded and have to be carefully chosen. In order to limit the parameter space of network emulation, we set the path bandwidth levels BW_{pri} and BW_{sec} according to commonly observable wireless data rates, from 300 - 2100 KB/s, under the additional assumption that a path bandwidth ratio smaller than 1:7 is of limited interest for aggregation. The RTT levels RTT_{pri} and RTT_{sec} are also set to a range of values observable in wireless networks, with values as low as 10 ms in local wireless access networks and RTTs beyond 200 ms in 3G data networks. For the probability of randomly induced IP packet loss L_{pri} and L_{sec} we opt for covering a large range from a practically negligible loss of 0.0000004% to very high levels of up to 6.25%.

We define a sample experiment as a 60-second TCP connection with an arbitrary configuration of the listed parameters. Considering the large number of parameters that define a sample, it is practically impossible to run the full set of experiments in acceptable time. Even by limiting all parameters to only three different value levels, a runtime of two years would be necessary. In addition, even in an emulated network, the data throughput of a TCP connection is not purely deterministic and it is therefore recommended to run multiple experiments for each parameter configuration, which would require an even longer runtime. We therefore resort to an orthogonal design of experiments, a method well described by Taguchi et al. [156] that allows a dramatic reduction in the number of experiments, while still providing sufficient information to draw statistically sound conclusions about the relevance of each parameter.

Orthogonal Design of Experiments

The essence of an orthogonal design of experiments is to cleverly select only a small number of sample experiments instead of exploring all possible parameter combinations. The chosen subset of experiments ensures that each level of a given parameter is equally often combined with all levels of any other parameter. For example, in a well-defined orthogonal design, the parameter level 10 ms of the parameter BW_{pri} is combined equally often with each level of BW_{sec}, each level of RTT_{pri}, each level of RTT_{sec}, and so forth. All other parameter levels of BW_{pri}, and all levels of any other parameters, follow the same principle. After the experiments are constructed using an orthogonal design and the aggregation benefit of each experiment is collected using the network emulation testbed, the performance of each parameter level can be computed by averaging the results of all experiments containing this parameter level. For example, the average aggregation benefit of having the SACK mechanism enabled can be calculated by considering the results of all experiments where SACK is equal to 1. Analogously, the expected multipath performance of TCP without SACK can be computed. The correctness of this procedure is based on the argument that all possible side-effects are fairly averaged out and accounted for. In other words, the orthogonal design of experiments guarantees an equal number of experiments with SACK=1 and with SACK=0.

An orthogonal design of experiment can be created as follows. Let P be the number of parameters under test and L the number of levels each parameter can take. For the moment we assume that all parameters allow the same number of levels. Let (p, l) denote a level setting l of a parameter p ($p \in \{1, ..., P\}, l \in \{1, ..., L\}$). The idea of an orthogonal design is to define a set of experiments such that for each pair of parameters, p_1 and p_2 ($p_1 \neq p_2$), and for any choice of corresponding levels (p_1, l_1) and (p_2, l_2) an equal number of experiments combining the two parameter levels occurs. The construction of such a set of experiments is straightforward if L is a prime power, i.e., $L = q^n$, where q is a prime number and $n \geq 1$ an integer. In this case there exists a complete set of $L - 1$ mutually orthogonal latin squares (MOLS) [65, 100]. From these $L - 1$ latin squares of size $L \times L$, an orthogonal array with L^2 rows (experiments) and $L + 1$ columns (parameters) can be readily obtained as decribed by Hedayat et al. [65].

The number of our parameters listed in Table 5.5 is $P = 14$, exactly 1 larger than the prime number 13. We therefore choose $L = 13$ as the appropriate number of test levels for each parameter. However, a small adjustment to the described construction is necessary, because most of the TCP parameters have less than 13 levels available. This circumstance is dealt with by replacing all level settings exceeding the predefined maximum by allowed levels in a random and balanced fashion. This procedure does not meet a strict orthogonality criterion, but for a large amount of experiments conducted, a good randomization is achieved. Algorithm 4 summarizes the steps for the definition of the experimental runs to be performed.

Algorithm 4 Orthogonal Design of Experiments

```
1: Generate a set of $L - 1$ mutually orthogonal Latin squares
2: Use the Latin squares to build an orthogonal array $A$ of size $L^2 \times (L + 1)$
3: while more experiment configurations shall be created do
4:     Randomly permute the $L + 1$ columns of $A$ (to create a new level allocation)
5:     Concatenate $A$ to the final set of experiments
6: end while
```

In our specific case with $P = 14$ and $L = 13$, an orthogonal array defines $L^2 = 13^2 = 169$ experiments. In order to increase the confidence in our results, we created a total of 400 different orthogonal arrays, resulting in 67600 different multipath TCP experiments with a combined runtime of about 47 days. In the following, we analyze the individual effect of each TCP parameter under the randomly generated network characteristics. Whenever error bars are displayed, they represent a 95% confidence interval around the averages obtained from the 400 sets of experiments.

The Effect of TCP Congestion Control

Figure 5.15 compares the multipath performance of all TCP congestion control algorithms that are available in Linux. The x-axis is sorted by the average aggregation benefit that is achieved when all levels of randomly added packet loss are included.

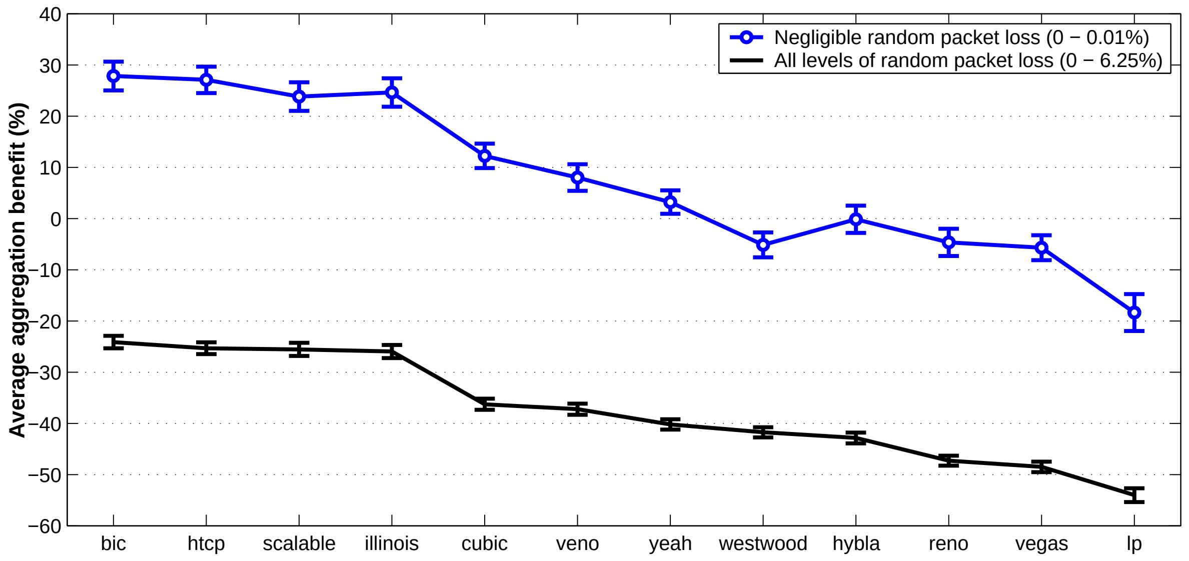


Figure 5.15: *Ranking of TCP congestion control algorithms when faced with a large variety of multipath scenarios.*

The four high-speed congestion control algorithms BIC, HTCP, Scalable TCP and Illinois are leading with an advance of more than 10% in aggregation benefit ahead of

Linux' default CUBIC. All other congestion control algorithms are consistently worse, with Low-Priority TCP at the end of the ranking list.

The Effect of TCP Loss Recovery Mechanisms

Figure 5.16 displays all reasonable combinations of the SACK, FACK, DSACK, and F-RTO mechanisms, ranked by their average achieved aggregation benefit. Interestingly, the default parameter setting of Linux TCP ranks somewhere in the middle of the ranking list. The main reason for this is the negative effect of the FACK algorithm, which fails to operate correctly in the presence of packet reordering. Even though Linux automatically disables FACK when reordering is detected, it is better to turn off this mechanism right from the start of a multipath TCP connection. This observation is clearly underlined by the fact that the best five parameter combinations all contain FACK=0.

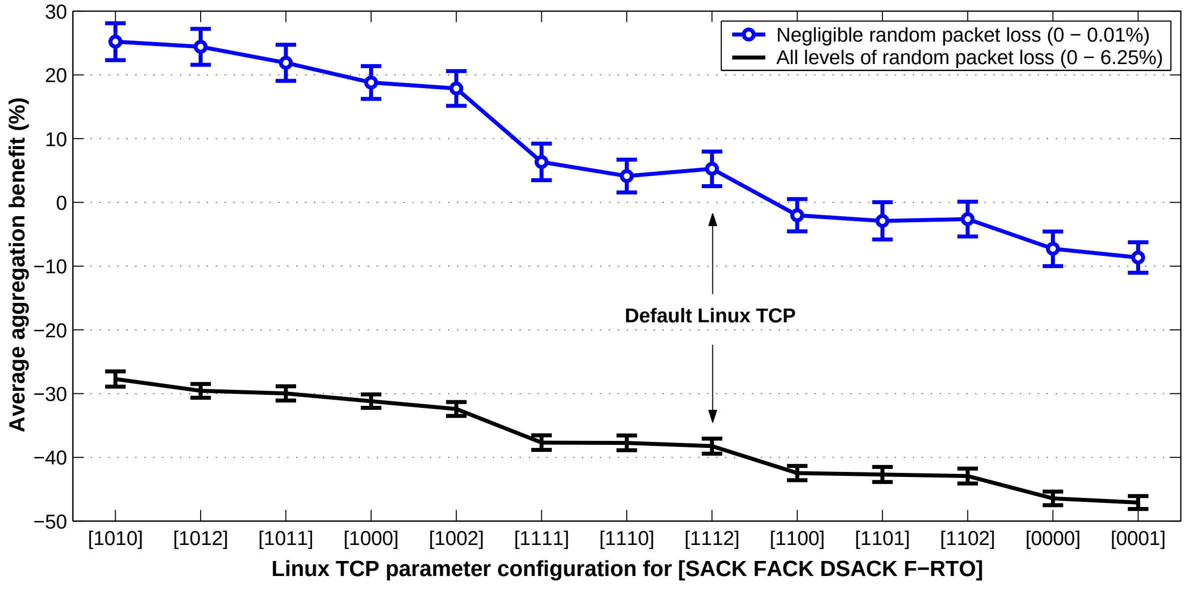


Figure 5.16: *Ranking of TCP loss recovery mechanisms. The example notation of [1112] stands for a default Linux TCP configuration [SACK=1, FACK=1, DSACK=1, F-RTO=2].*

Trailing at the end of the ranking list are the scenarios with SACK disabled, which emphasizes the importance of using SACK information for improved multipath performance. In fact, enabling only SACK, but disabling all other mechanisms results in a significanly higher aggregation benefit than using Linux TCP with every mechanism enabled by default. The impact of DSACK and the F-RTO algorithms are less obvious at a first glance. At a closer look, however, DSACK clearly brings a benefit when enabled. The top three configurations all contain DSACK=1, and every configuration with DSACK=0 improves by switching it to DSACK=1. On the other hand, F-RTO has no obvious relevance on TCP's multipath performance. Switching F-RTO between the allowed values of 1, 2, and 3, never results in a significant change in performance. In summary, we draw the following conclusions for an optimized configuration of Linux TCP over multiple paths: (1) SACK must be enabled, (2) FACK must be disabled, (3) DSACK should be enabled, and (4) F-RTO may be enabled. We observed that with a careful setting of these parameters, a significant aggregation benefit can be achieved in networks with small packet loss.

The Effect of the `reordering`$_{min}$ Variable

Figure 5.17 shows the impact of increasing the standard dupACK threshold of 3 to values up to Linux' maximum of 127. The trend of our experiments clearly implies an advantage of raising the number of required dupACKs to trigger fast retransmit in multipath scenarios. However, the positive effect of increasing the reordering metric decreases logarithmically and we did not find much benefit in adjusting `reordering`$_{min}$ to values larger than 50. These observations equally apply scenarios with and without packet loss, although the effect is more beneficial when packet loss is low.

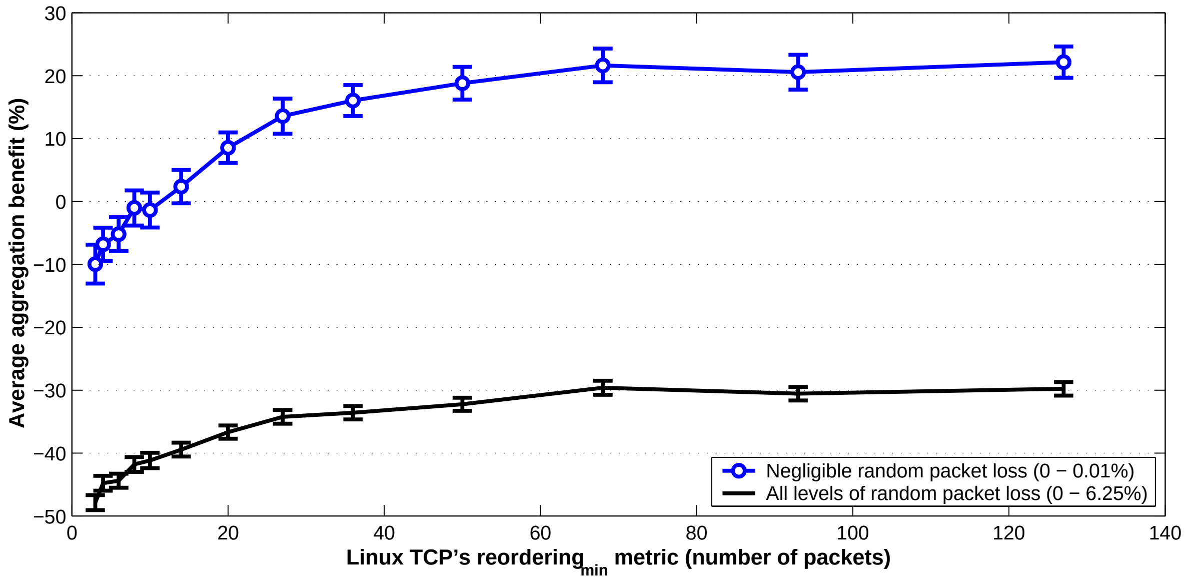


Figure 5.17: *The effect of the* `reordering`$_{min}$ *variable.*

5.3 Tuning Linux TCP for Improved Multipath Performance

In the previous section, we applied an orthogonal design of experiments to explore parameter settings that result in an improved multipath performance. In this section, we first use these settings to create new TCP flavors that are tuned for enhanced performance over multiple paths. Second, we use our best tuned Linux TCP to compete against the TCP-DCR [22] algorithm.

Comparing Tuned Linux TCP to Linux TCP

Through a large set of experiments, we have identified several parameters that can be modified in Linux to increase the aggregation benefit of TCP over multiple paths. In particular, the default congestion control algorithm in Linux, CUBIC, was outperformed by several high-speed TCP variants, with BIC leading by a narrow margin. Among the TCP loss recovery mechanisms, FACK has clearly shown negative side-effects, suggesting that it should be turned off before establishing a connection. In addition, increasing

$\texttt{reordering}_{min}$ (the dupACK threshold) has shown a consistent improvement in performance. Based on these results, we define a set of tuned Linux TCP flavors whose parameter configurations are summarized in Table 5.6.

Table 5.6: COMPARISON OF LINUX TCP AND OUR TUNED TCP FLAVORS

	Linux TCP	Tuned TCP	R50-Tuned TCP
Congestion control	CUBIC	BIC	BIC
SACK	1	1	1
FACK	1	0	0
DSACK	1	1	1
F-RTO	2	2	2
TCP timestamps	1	1	1
$\texttt{reordering}_{min}$	3	3	50

Figure 5.18 shows the aggregation benefit arising from TCP parameter tuning. For a fair comparison, the results are based on the same 10000 experiment configurations that we used for testing the other well-known flavors, such as TCP NewReno. While default Linux TCP reached an average aggregation benefit of 9%, our tuned versions performed much more efficiently and resulted in aggregation benefits of 27% and 39%, respectively. While default Linux TCP gained from using multiple paths in 56% of all tested network scenarios, R50-Tuned TCP reached a positive aggregation benefit in 77% of the experiments.

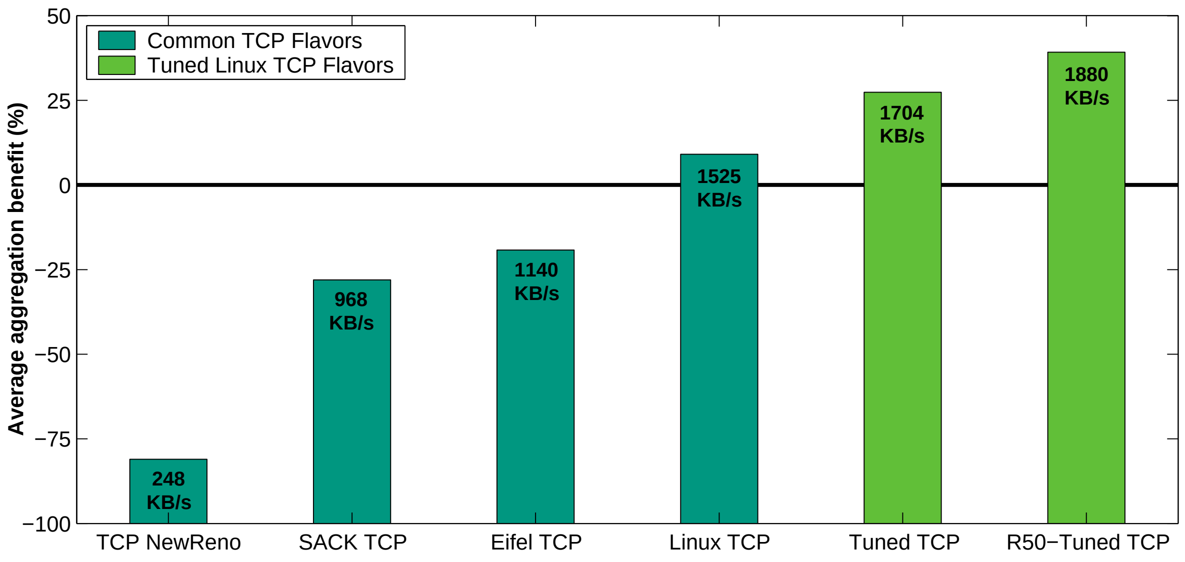
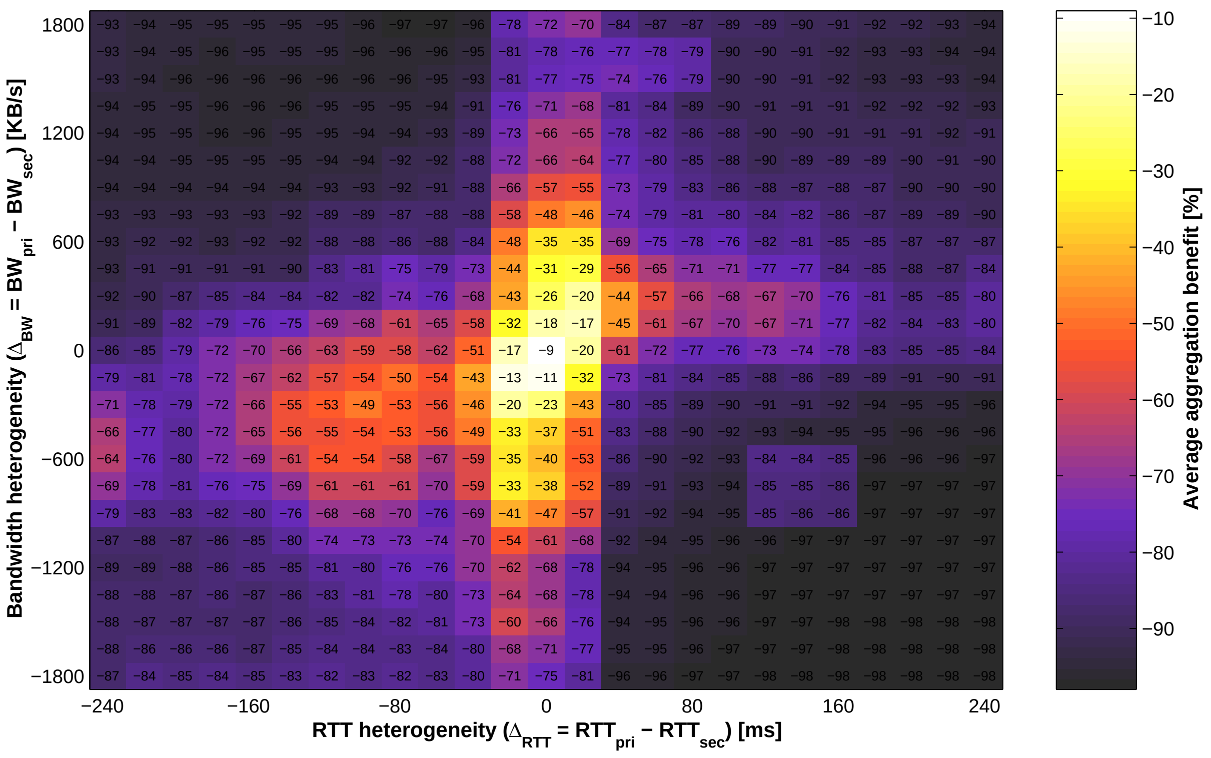
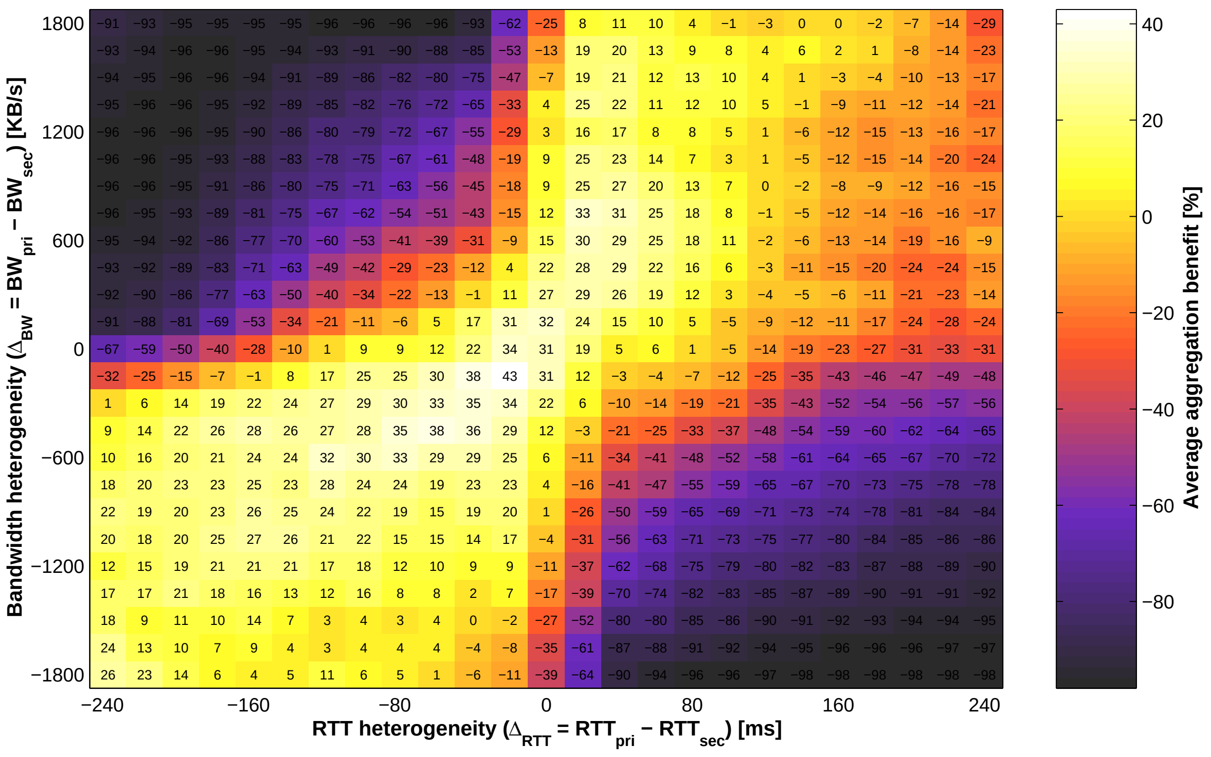


Figure 5.18: *The multipath performance of tuned Linux TCP.*

Figures 5.19 and 5.20 summarize how bandwidth and RTT heterogeneity affect the aggregation benefit of various TCP flavors in the absence of randomly introduced packet loss. As noted in the axis labels, we define the *heterogeneity* Δ as the difference in absolute values between the primary and secondary paths. For example, an RTT heterogeneity of 50 ms implies that $\Delta_{RTT} = RTT_{pri} - RTT_{sec} = 50\,\text{ms}$. Analogously, a bandwidth

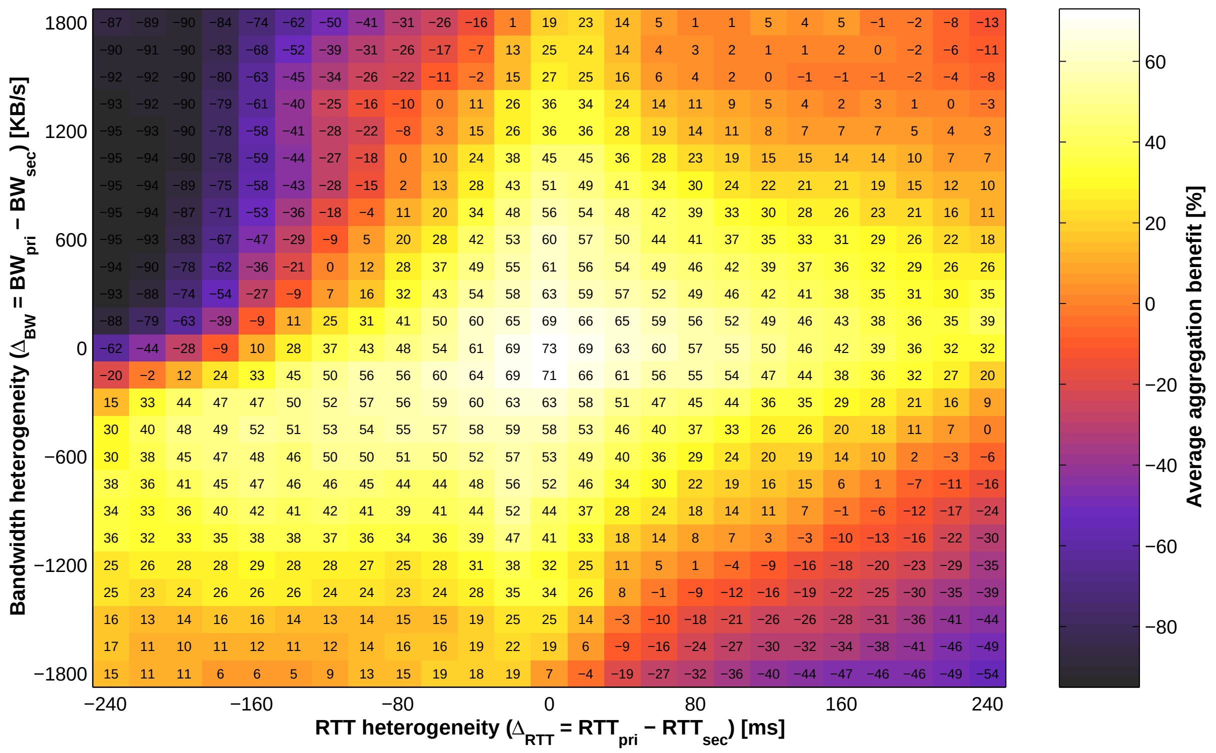


(a) TCP New Reno

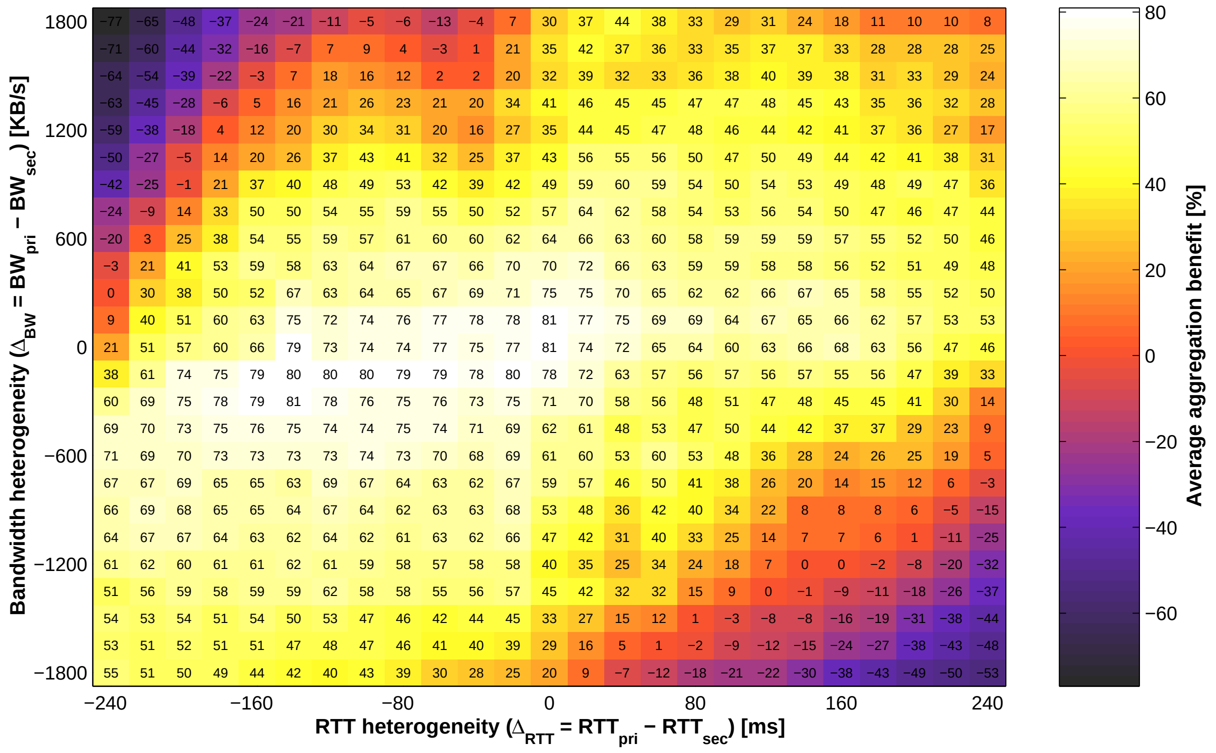


(b) SACK TCP

Figure 5.19: *The effect of path heterogeneity on the aggregation benefit of (a) TCP New Reno and (b) SACK TCP.*



(a) Default Linux TCP



(b) R50-Tuned Linux TCP

Figure 5.20: *The effect of path heterogeneity on the aggregation benefit of (a) default Linux TCP and (b) tuned Linux TCP.*

heterogeneity of -1200 KB/s implies that $\Delta_{BW} = BW_{pri} - BW_{sec} =$ -1200 KB/s. While Δ is not the only factor (the absolute values are also important), it is an intuitive measure of heterogeneity that allows a quick comparison of different TCP flavors and leads to some interesting observations:

- Standard TCP New Reno consistently fails to benefit from multipath forwarding, even under homogeneous path conditions (center of the plot). On the contrary, Linux TCP is fairly robust to small and moderate deviations from equal path characteristics. After fine-tuning Linux TCP, an average aggregation benefit of over 50% and up to 80% is achieved for a large variety of multipath configurations. Only the most heterogeneous path conditions result in a severe performance degradation below the bandwidth of the faster path.

- In general, a good aggregation benefit can be achieved if the sign of both heterogeneity measures are the same, i.e., $\{\Delta_{BW} < 0$ and $\Delta_{RTT} < 0\}$ or $\{\Delta_{BW} > 0$ and $\Delta_{RTT} > 0\}$. If one path is larger in bandwidth *and* RTT than the other, then there is a positive aggregation benefit almost over the entire range considered (except in the case of TCP New Reno). Especially in cases where Δ_{BW} or Δ_{RTT} are very negative, this implies that it is preferable to have paths with a high difference in *bandwidth-delay product* ($\Delta_{BW} * \Delta_{RTT}$), i.e., a path with a higher number of bytes "in flight".

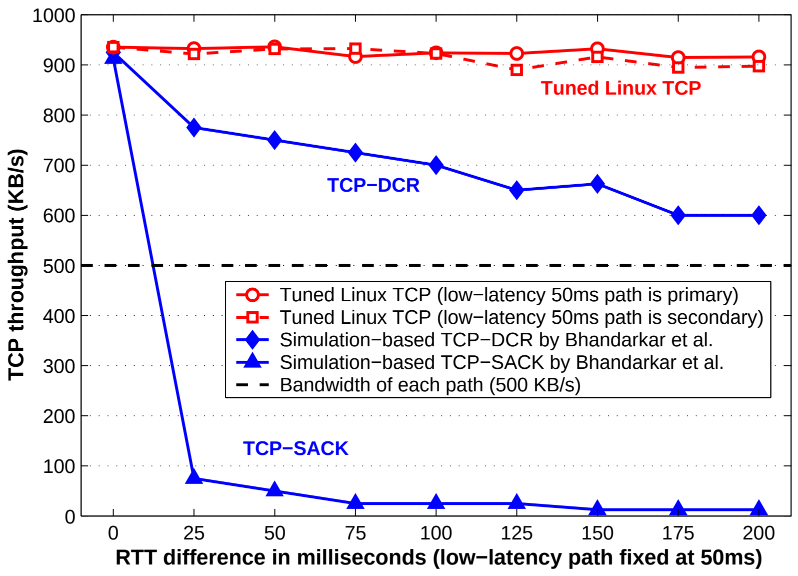
Comparing Tuned Linux TCP to TCP-DCR

As a final experiment, we bring our results into context of previous research. To the best of our knowledge, there exists only one research paper with results that are directly comparable to ours. Although a large variety of reorder-robust TCP flavors have been published, all algorithms known to us were tested with artificially introduced IP packet reordering, typically by randomly delaying certain packets that travel over a single path. The only authors known to us, who tested their TCP-DCR [22] algorithm in a scenario of packet-based multipath forwarding, are Bhandarkar et al. In their paper, the authors evaluate TCP-DCR under a variety of situations, including one simple example of multipath forwarding, in which a round-robin scheduler distributes IP packets over two paths with an aggregated bandwidth of 1000 KB/s (i.e., $BW_{pri} = BW_{sec} = 500$ KB/s). While one of the paths is configured with a fixed RTT of 50 ms, the other path's RTT is increased from 50 ms up to 250 ms in steps of 25 ms. In their experiment, the queue size Q is set to 50 packets and neither cross-traffic nor random loss is introduced (i.e., $L_{pri} = L_{sec} = 0$).

We have translated the simulation scenario of Bhandarkar et al. into our network emulation testbed with only two remaining uncertainties. First, the authors of TCP-DCR based their results on TCP connections lasting for 1100 seconds, but ignoring the first 100 seconds of the connection (to ensure that a steady state is reached). We consider this kind of truncating as inappropriate and instead use TCP connections lasting exactly 60 seconds. Second, since it is not clear which path the returning ACKs take in their simulation, we run two sets of experiments, one with a fixed $RTT_{pri} = 50$ ms and variable secondary RTT, and another with a fixed $RTT_{sec} = 50$ ms and variable primary RTT.

The resulting comparison of TCP-DCR and tuned Linux TCP is shown in Figure 5.21. The lower two curves (marked with diamonds and triangles) are copied from Figure 3 in [22], while the upper two curves were obtained from our emulation testbed with Linux

TCP configured as listed in the side table. In particular, we have turned off the FACK mechanism, changed the congestion control algorithm to BIC, and modified the minimum `reordering` metric to 30. While TCP-DCR slowly loses in its aggregation benefit with increasing RTT heterogeneity, our tuned Linux TCP manages to achieve close to perfect multipath aggregation.



Tuned Linux TCP	
Cong. control	BIC
SACK	1
DSACK	1
FACK	0
F-RTO	2
Timestamps	1
$\texttt{reordering}_{min}$	30

Figure 5.21: *A comparison of tuned Linux TCP to TCP-DCR.*

5.4 Discussion

In this chapter, we made several interesting observations from evaluating TCP's performance when IP packets are forwarded over two paths with heterogeneous characteristics. A first experiment showed that the modern TCP implementation of the Linux operating system is able to sustain a considerable aggregation benefit over combined WLAN and HSPA networks, both in a practical environment and also with emulated paths. In a second experiment, we expanded the network heterogeneity to a wide variety of combined path properties and numerous TCP-related parameters, including different congestion control algorithms and loss recovery mechanisms. Through a large set of emulation-based experiments, we identified several Linux TCP parameters that lead to improved multipath performance. By correctly tuning these parameters, we finally achieved a 30% increased aggregation benefit compared to Linux TCP's default parameter settings. In addition, our tuned Linux TCP significantly outperformed TCP-DCR in a direct comparison.

While our experiments confirm the commonly accepted notion that standard TCP suffers a drastic loss in performance due to multipath forwarding, our results show that a full-featured TCP is not easily defeated by even some of the most extreme cases of path heterogeneity. Linux TCP achieves a consistent aggregation benefit in moderately heterogeneous configurations and operates well within a large range of path heterogeneity. Even though the criteria of achieving an aggregation benefit under heterogeneous path characteristics (C1 to C3) can not be fully guaranteed, they were met in the majority of the tested scenarios. Since our experiments did not introduce protocol modifications, criterion C4 is also satified. In addition, by moving the scheduling to a proxy, the criteria C5 and C6 could both be met at the cost of breaking criterion C7.

Linux TCP's robustness to reordering indicates a great potential of a multipath-robust TCP and it cannot be ruled out that TCP implementations in other operating systems perform even better. Performing a comparison of various operating systems would therefore be a worthwhile endeavor in the future. However, even if other TCP implementations achieved a consistent aggregation benefit and outperformed our fine-tuned Linux TCP, this would only reinforce our belief that further improving TCP's robustness to reordering could be a viable alternative to designing a new protocol from scratch. The insights gained from the analysis in this chapter thus lead us to several conclusions and possible future research directions:

Clean Slate From a radical point of view, we feel tempted to conclude that it is time for TCP to finally be extended to fully cope with IP packet reordering. The IP layer should be allowed to return back to its principle of "best effort" delivery, while routers and switches should be freed from their resource-intensive responsibility of maintaining in-order transport connections. Full trust in TCP's robustness against reordering would help to offload routers and allow convergence toward a truly multipath-enabled Internet.

Multipath TCP A less far-reaching way forward is the design of a new TCP-compatible protocol with support for multiple interfaces. The MPTCP working group in the IETF follows this approach by specifying an experimental protocol that opens simultaneous TCP subflows, similar to existing application-layer solutions, which establish a number of concurrent TCP connections (such as pTCP [70]). Preliminary tests by Barré et at. [16] already show promising results and time will tell whether MPTCP will find wide acceptance.

Multipath Proxy From a less academic and more commercially oriented viewpoint, the deployment of multipath-enabling proxies appears like the most promising option for a short-term support of multiple interfaces. A solution similar to the NAMA solution presented in Chapter 4 could be employed to optimize TCP flows, detect situations in which multipath aggregation is highly beneficial, and refrain from using multiple paths when no gain is expected.

Chapter 6

Multipath Aggregation at the Application Layer

This chapter presents an application-layer solution for multipath aggregation, which builds upon the idea of file segmentation in a somewhat comparable way as the BitTorrent/ECMP-based approach described in the introductory Section 1.3. However, instead of opening a large number of transport flows and counting on a routing protocol to spread the workload evenly across multiple paths, a single connection per client interface is sufficient for the solution presented in this chapter. The main idea is to use an existing protocol that allows parts of a resource to be retrieved, so that the client's interfaces can collaboratively download fragments of the same resource and reassemble them locally, therefore achieving higher data throughput.

The solution presented here is based on the Hypertext Transfer Protocol (HTTP) [51], the fundamental communication protocol in the World Wide Web. Relying on the HTTP-based functionality provided by standard Web servers, multipath aggregation can be achieved in a lightweight and purely client-based manner, with the benefit of easy deployment and interoperability with existing infrastructure. Although generally applicable to file download (and other protocols, such as FTP), we present our solution in the context of multimedia streaming. With the introduction of HTTP-based progressive download and services like Youtube[1], multimedia streaming has rapidly risen in popularity and has become one of the dominating services on the Internet today. There are even claims that in 2013, almost 64% of the world's mobile data traffic will be video [37]. Above all, multimedia streaming is a very resource-intensive application and presents itself as a promising candidate for bandwidth aggregation.

In a first step (→ Section 6.2), we propose and evaluate an HTTP-based approach for transferring a single byte stream over multiple access networks. Hereby, we focus on maximizing the in-order data throughput over multiple interfaces, assuming that every received in-order byte can immediately be played back by the client. Application-specific restrictions, such as video format and frame length, are deliberately ignored, because an increase in the average data throughput is equivalent to a potential improvement of the user-perceived multimedia quality. In addition, we consider a startup latency and client-side buffer requirements as important metrics that need to be minimized.

In a second step (→ Section 6.3), we embed our algorithm for multipath aggregation

[1]http://www.youtube.com

into DAVVI, a video streaming platform developed by Johansen et al [85]. DAVVI already follows the concept of video HTTP-based file segmentation, but uses it for quality adaptation and content search functionality. Implementing a *multipath request scheduler* into DAVVI, which additionally allows video segments to be simultaneously pulled over multiple interfaces, is therefore a very promising extension for enhancing the video streaming performance. Our results show that such a client-side modification of DAVVI leads to higher data throughput that might result in a significant boost in video quality.

6.1 Relevant Features of the HTTP Protocol

This section briefly introduces the basics of the Hypertext Transfer Protocol (HTTP) and explains some of its core functionality that is used in subsequent sections to enable multipath aggregation.

6.1.1 Mode of Operation

HTTP is an application-layer communication protocol that allows clients to request remotely stored content or otherwise interact with a server. Sessions between a client (commonly a Web browser) and an HTTP server are typically[2] established based on TCP connections. The client communicates with the server by sending text-based requests containing a basic command. For example, the command `GET` is used to retrieve data from the server, `HEAD` to obtain metainformation (e.g., the content length), and `PUT` to upload data (usually information filled into Web forms). Requests from the client also contain a Uniform Resource Identifier (URI) [19] of the desired file and a `Host` header field with the server's name:

```
GET /~kaspar/example-movie.mpg HTTP/1.1
Host: www.simula.no
```

Upon receiving a request, the server returns a response message containing a similarly formatted header with a status code (such as `200 OK` in case of successful operation), followed by an empty line and the requested content. In the following example, the server indicates its support for byte-range requests by an additional `Accept-Ranges` header field:

```
HTTP/1.1 200 OK
Accept-Ranges: bytes

[PAYLOAD]
```

The latest protocol specification of HTTP is defined in RFC 2616 [51] and is often referred to as HTTP/1.1. The main difference to its previous version, HTTP/1.0 [20], is that it allows a single transport connection to be reused for many requests to the same server. With TCP, the most common example, connection establishment involves a three-way handshake that introduces an overhead of two RTTs, followed by a conservative slow-start phase, in which the path capacity is not fully utilized. Particularly when requesting

[2]TCP is no requirement for HTTP. For example, Funasaka et at. [59] propose a UDP-based HTTP.

a large number of small files (or file segments), a persistent connection thus considerably improves performance compared to opening and closing a new TCP connection for each request. Along with the concept of persistent connections, HTTP/1.1 also supports the features of partial content requests and request pipelining, which are the foundation of the multipath solution presented here.

6.1.2 HTTP Range Retrieval Requests

Most HTTP servers support so-called *range retrieval requests* (also referred to in short as *range requests*), which were designed with the intention of allowing interrupted downloads to proceed with the outstanding part at a later time [123]. Another common purpose of range requests is to download specific parts of an object, for example when only a few pages of a large document are relevant. For example, the following HTTP request asks for the first 100-kilobyte segment of a large movie:

```
GET /~kaspar/example-movie.mpg HTTP/1.1
Host: www.simula.no
Range: bytes=0-99999
```

In our solution, we use range requests for the novel purpose of achieving multipath aggregation. By dividing a file into many logical segments, multiple network interfaces can collaboratively download a single file at a higher speed. A similar approach is frequently used by *download managers* to achieve a larger throughput by opening multiple transport flows over a single network interface. Searching the Internet for existing download managers, we have found over forty such tools [166]. Most of them are advertised with a multitude of features, such as the ability to connect to multiple mirror sites and multi-protocol support. However, to the best of our knowledge, not a single existing download manager actively supports data transfer over multiple network interfaces at the client.

6.1.3 HTTP Request Pipelining

According to the protocol specification, HTTP request pipelining is a method that “allows a client to make multiple requests without waiting for each response, allowing a single TCP connection to be used much more efficiently, with much lower elapsed time” [51]. Figure 6.1 illustrates the difference between the typical way of sequential request processing and the pipelined method.

In the absence of pipelining, each request must be fully processed by the server and all requested data must be transmitted before the client is allowed to send the next request. Thus, for each request, an average time overhead of one RTT is incurred. When the number of requests is large, for example when a website is composed of elements from many sources or when a file is logically segmented into numerous small pieces, this overhead significantly impairs the throughput of high-latency connections (such as satellite or cellular data networks). HTTP request pipelining eliminates most periods of idle waiting time at the client and leads to a better utilization of the available path capactiy. Servers that support pipelining must process requests in the same order they were received. Ensuring correct order only requires that the requests are buffered in a queue. HTTP pipelining is supported by most Web servers.

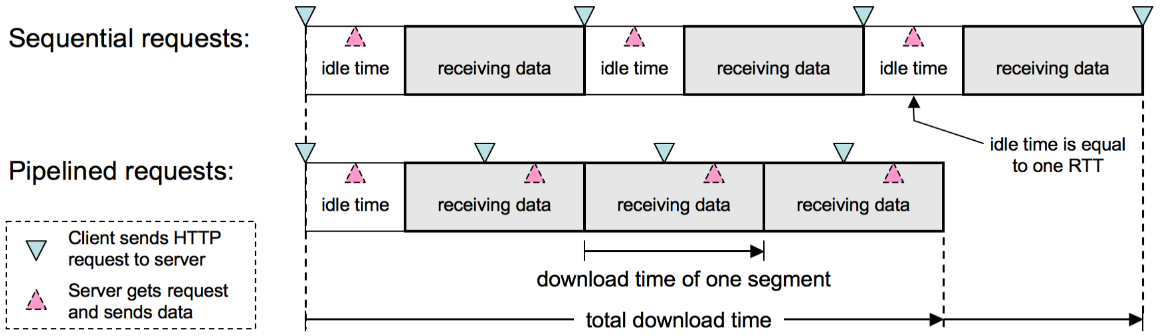


Figure 6.1: *From a client-side perspective, HTTP request pipelining eliminates the time overhead incurred by a sequential processing of requests and leads to higher data throughput.*

6.2 Multipath Progressive Download

The term *progressive download* describes the method of allowing data playback on the client while the download is still in progress. Initially coined as "Fast Start" in Apple's QuickTime Streaming Server [11], progressive download is also supported by several other commercial streaming products, including Microsoft's Internet Information Services [171]. Although these systems are quality-adaptive and based on file segmentation principles, they fail to enhance the quality of multimedia streaming by supporting multiple network interfaces at the client.

The novelity of our multipath aggregation solution lies in the combination of HTTP's features of range retrieval requests and pipelining. As illustrated in Figure 6.2, rather than using range requests for the purpose they were originally designed for (i.e., resuming partially delivered files), we use them for inverse multiplexing. In other words, data segments are requested from the server over multiple paths and the received data is assembled into correct order at the client.

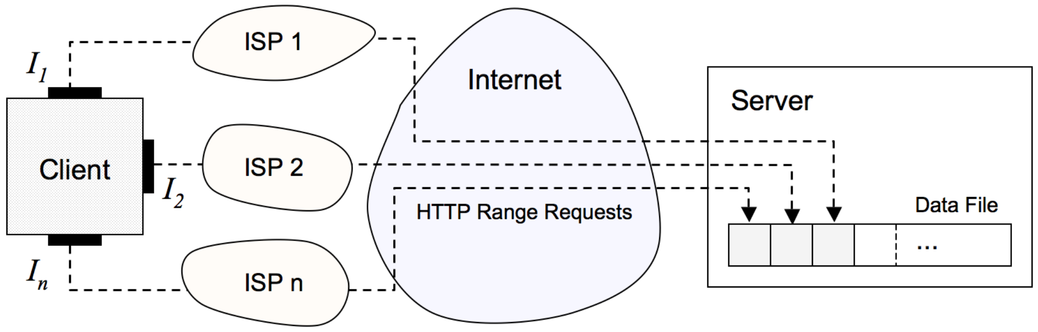


Figure 6.2: *Scenario of a client connected to n Internet service providers (ISPs). The client sends HTTP range requests over its multiple network interfaces I_1, ..., I_n to simultaneously download a single file that is logically split into many fixed-size segments.*

A fundamental property of progressive download is the use of client-side buffering to compensate for variable throughput and to achieve a smooth playback. If the received and buffered data is consumed too fast by the client application, such as a video player, the playback pauses until sufficient data is available again. Such phases of rebuffering are perceived by users as service disruptions and should be avoided. A common approach to mitigate interruptions in the video experience is to introduce a brief waiting time for filling the buffer prior to playback. The tolerable startup latency is subjective and dependent on the media content. For example, when starting a full-length motion picture, users might be willing to wait a minute before show time, but for shorter video clips, the startup delay should be perceived as instantaneous.

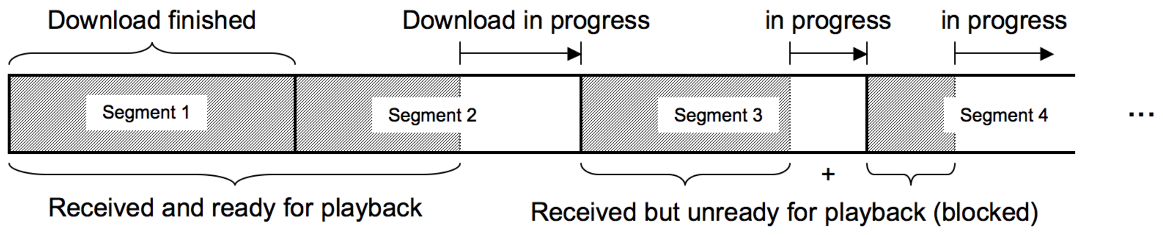


Figure 6.3: *Head-of-line blocking illustrated as a snapshot of received byte ranges during a progressing download with three interfaces simultaneously retrieving a segment each. Data blocked by a gap is unready for playback and must be buffered.*

Progressive download over multiple paths raises additional challenges for achieving a smooth playback. Even if segments are requested sequentially over the available interfaces, incoming packets at the client may contain data that is not linear with respect to the playback order. As illustrated in Figure 6.3, receiving data over paths with different transfer rates leads to gaps in the received file and causes head-of-line blocking.

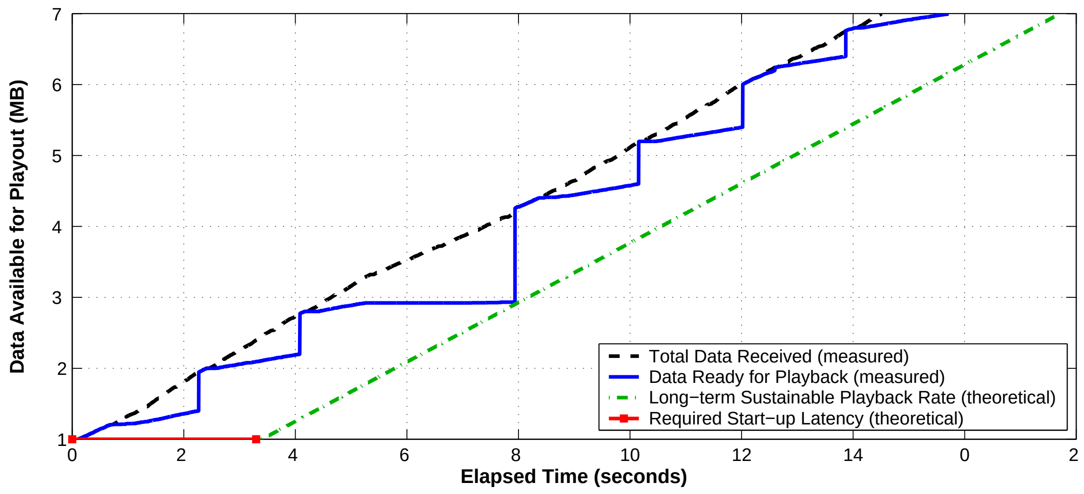


Figure 6.4: *Playback timeline of progressive download over multiple paths: the use of heterogeneous access networks (in this case WLAN$_1$ and HSPA$_1$) causes head-of-line blocking and results in a step-wise increase of data ready for playback.*

In contrast to typical progressive download over a single path, buffer space is required to store out-of-order byte ranges and to compensate for head-of-line blocking. Thus, a larger startup latency is necessary to allow for a smooth playback at a bitrate equivalent to the total aggregated throughput. Based on a field experiment with our solution and the WLAN$_1$ and HSPA$_1$ networks, Figure 6.4 illustrates the interaction between several performance values associated with progressive download over multiple interfaces. While the total amount of received data (dashed curve) is smoothly increasing, the actual in-order received data that can be played back forms a step-wise increasing curve. The steps occur when received but blocked data gets released, for example when the download of Segment 2 in Figure 6.3 is completed and suddenly a large amount of data is unblocked.

Under the assumption that every received in-order byte is immediately consumable by a receiver application, the theoretical value of a *long-term sustainable playback rate* can be represented as a line parallel to the total aggregated data received, with an x-axis offset equal to the startup latency. Although unknown in advance, the optimal long-term sustainable playback rate has the same slope as the average aggregated data throughput and covers all instances of head-of-line blocking, forming a tangent to the worst-case step of the "data ready for playback" curve. In the example shown in Figure 6.4, if the receiver application would pre-buffer data for 3.2 seconds before starting playback, an uninterrupted playback could be provided to the user. With a startup latency smaller than 3.2 seconds, the client application would face situations of lacking data for playback, resulting in rebuffering lags later on. On the other hand, a startup latency larger than 3.2 seconds would waste buffer space and force the user to unnecessarily wait before playback.

In other words, there is a tradeoff between the startup latency and the long-term sustainable playback rate. Reducing the startup latency implies either that playback lags are introduced or that the video quality must be lowered. However, nothing is gained by increasing the startup latency beyond the point where all worst-case instances of head-of-line blocking are covered. Therefore, the primary target of this work is to maximize the long-term sustainable playback rate, while providing the smallest possible startup latency.

6.2.1 Playback Metrics

For the analysis of our solution, this sections describes the informally introduced playback metrics in more detail. Due to network heterogeneity and dynamically changing path conditions, the shortest startup latency that achieves a long-term sustainable playback rate is a priori unknown. Any network interface involved in the multipath progressive download might suddenly experience a reduction in throughput, cause a worst-case instance of head-of-line blocking and therefore stretch the required startup latency. Only after downloading the entire content and logging the timestamps of all received packets, it becomes possible to define the long-term sustainable playback rate and calculate what the optimal startup latency would have been.

The Long-term Sustainable Playback Bitrate

Following our primary goal of maximizing data throughput, the long-term sustainable playback rate r is the most important performance metric in our experiments. It represents the optimal transfer rate that an instance of multipath progressive download is able to support. This is generally unknown at the beginning the data transmission. In retrospect, however, we can calculate the long-term sustainable playback rate r as the average aggregated throughput by dividing the total content length by the transfer duration:

$$r = \frac{\text{total number of downloaded bytes}}{\text{elapsed time since first HTTP request}} \tag{6.1}$$

After a download is finished and the target r is defined, the optimal startup latency and the required buffer requirements can be calculated from the download history of received bytes. The download history of an experiment is created by logging the timestamp and byte range for each received data packet.

The Startup Latency

The *startup latency* is the time period between sending the first request and before playback starts. It has the purpose of pre-buffering data and preventing lagging playback in the future. At the beginning of a multipath progressive download, the shortest startup latency that guarantees compensation for later throughput variances and path heterogeneity is unknown. In the following experiments, the optimal startup latency is therefore deduced after all data transmission is finished.

To compute the startup latency, the logged download history is virtually replayed to reconstruct the state of the client-side buffer at any time t during the download. With the complete view of downloaded file pieces, as shown in Figure 6.5, it becomes straightforward to determine the number of bytes that are ready for consumption by a player, $bytes\_ready(t)$, and the number of bytes affected by head-of-line blocking, $bytes\_blocked(t)$, at any time t.

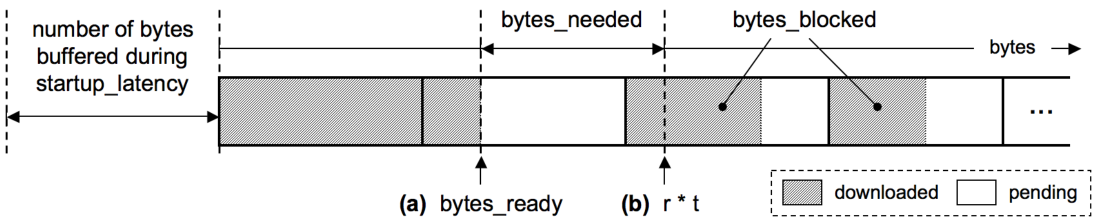


Figure 6.5: *The required startup latency can be illustrated as a race between (a) the number of bytes ready for playback and (b) the number of bytes $r * t$ that would be played back at a constant, long-term sustainable rate. The startup latency must be able to compensate for the number of bytes between the two pointers (a) and (b).*

With additional knowledge of the optimal bitrate r, the number of bytes $r * t$ that would be played back at a constant, long-term sustainable rate, can be calculated at every moment t. This allows the computation of the number of bytes that are needed, $bytes\_needed(t)$, for achieving an uninterrupted and sustainable playback at a given time t:

$$bytes\_needed(t) = r * t - bytes\_ready(t) \tag{6.2}$$

In other words, $bytes\_needed(t)$ is the number of pre-fetched, in-order bytes that must be present in the buffer at time t to achieve a constant playback rate r. For filling the buffer prior to playback with the number of needed bytes, a startup latency of $bytes\_needed(t)/r$ is required. The optimal startup latency, which guarantees that the maximum number of $bytes\_needed(t)$ are pre-fetched at any time t during the entire download, can therefore be calculated as follows:

$$startup\_latency = \frac{\max_t(bytes\_needed(t))}{r} \tag{6.3}$$

The Required Buffer Size

In order to store received data that is not ready for playback, buffering is needed. From a user's perspective, the required buffer size is of little relevance, but for devices with scarce memory, the maximum required buffer capacity represents an important design factor.

The required buffer size is normally understood as the maximum capacity needed to store those packets that were received during the startup waiting time (and during times of rebuffering). However, in a scenario of sending file segments over multiple links, the buffer size is additionally influenced by received data that is not in correct playback order ($bytes\_blocked(t)$). Similar to the reorder buffer occupancy density that was introduced in Section 2.2.2, the required buffer size for multipath progressive download depends on the amount of received out-of-order data.

In other words, the buffer occupancy at a given point in time t is the bytes pre-fetched during the startup phase plus the $blocked\_bytes(t)$. For guaranteeing a smooth, sustainable playback rate r, the pre-fetched data must be equal to the $bytes\_needed(t)$. Therefore, the required buffer size buf_req buf_req can be calculated as the maximum buffer occupancy during the course of a download:

$$buf\_req = \max_t(bytes\_blocked(t) + bytes\_needed(t)) \tag{6.4}$$

In consideration of the $startup\_latency$ and buf_req metrics, the following sections will explore the potential of using multiple network interfaces for enhancing the performance of progressive download.

6.2.2 The Impact of the Segment Size

Although a data resource at the server must be split into logical segments to be progressively downloaded over multiple paths, it is not obvious how the segments should be sized. For a deeper understanding of the effect of the segment size, we tested our solution in two scenarios of multipath progressive download with combined WLAN and HSPA networks[3] (with two and three network interfaces at the client). Figure 6.6 illustrates the long-term sustainable playback bitrate over heterogeneous access networks using a wide range of segment sizes. For each segment size, a 25.2 MB large file was transferred 50 times.

Two main observations can be made from Figure 6.6. First, both scenarios of using two and three aggregated interfaces achieve a significantly higher throughput than using the fastest interface (WLAN$_1$) alone. Second, an optimal segment size seems to exist for which the average long-term sustainable playback rate reaches a maximum. The scenario with three aggregated interfaces clearly shows a peak in performance with segment sizes between 250 and 1000 KB/s. The scenario with two interfaces does not exhibit a distinct peak, but follows the same trend of lowest performance for the smallest and largest segment sizes.

The segment size does not only affect the long-term sustainable playback bitrate, but also influences the startup latency and the required buffer size. Based on the same experimental data, Figure 6.7(a) depicts the average startup latency that satisfies the long-term achievable playback rate. When the segment size is small, it takes less time for transferring a full segment, which reduces the startup latency. From a consumer's

[3]The measured characteristics of the used WLAN and HSPA networks are summarized in Table 3.1.

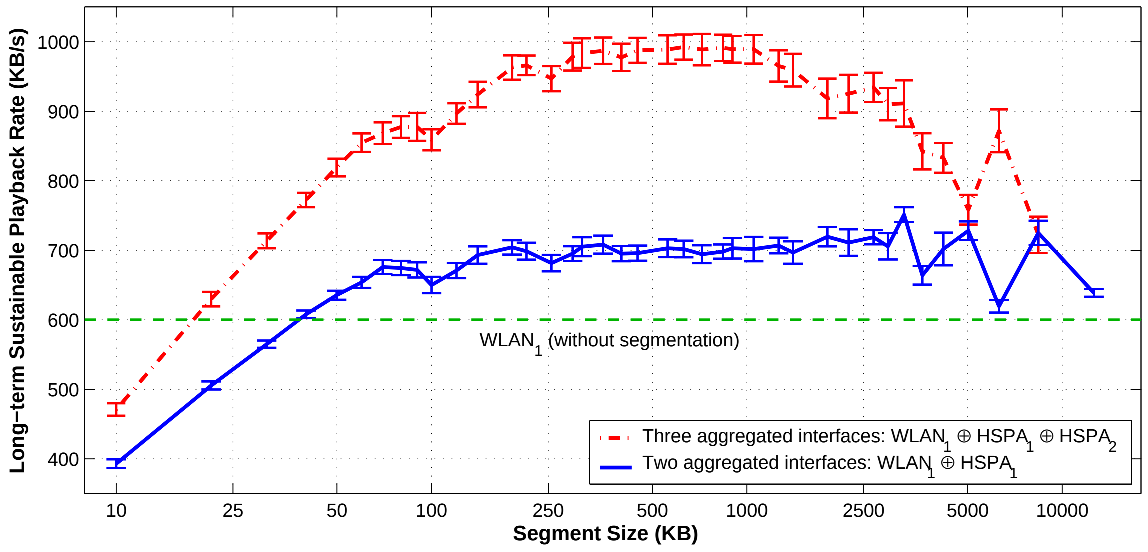


Figure 6.6: *The efficiency of multipath aggregation heavily depends on the segment size.*

perspective, choosing a smaller segment leads to a shorter startup latency at the cost of a lower playback quality. In addition, Figure 6.7(b) shows that the average required buffer size grows with increasing segment sizes and is in close correlation to the required startup latency. The only major difference between the two measures is their dependence on the number of used interfaces. As the network interface with the lowest throughput is the main contributor to head-of-line blocking, the simultaneous usage of additional interfaces with higher throughput has no significant impact on the startup latency. The buffer capacity, on the other hand, increases with the number of used network interfaces, because each interface causes additional out-of-order data to be blocked.

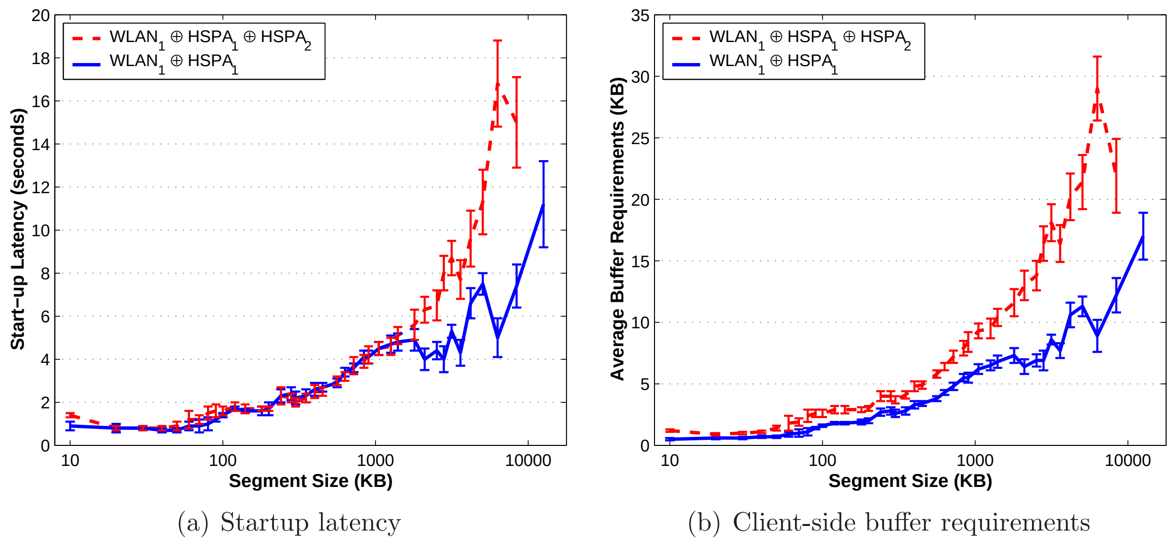


Figure 6.7: *HTTP-based multipath aggregation and the effect of different segment sizes on (a) the startup latency, and (b) the client-side buffer requirements.*

The Last Segment Problem

A common observation in Figures 6.6 and 6.7 is the seemingly random performance of multipath progressive download at large segment sizes. This phenomenon becomes most evident when dividing a file in only two, equally sized segments. If the used network paths are heterogeneous, one interface is bound to download its allocated part first and remain idle, while the remaining data transfer is completed by the slower interface(s). Even though this inefficiency is most apparent when using large segments, it cannot be avoided. There must always be one segment whose transfer ends last, no matter how small the segments are. With small segments, the *last segment problem* becomes negligible.

Summarized Conclusions from the Segment Size Analysis

Through this segment size analysis, we have acquired several insights indicating that using small segments are preferable over large segments:

1. With small segments, a short startup latency can be achieved.
2. Choosing a small segment size reduces the required buffer capacity.
3. The use of small segments mitigates the last segment problem.

A remaining drawback of using small segments is that they increase the number of HTTP `GET` requests to the server, which leads to an inefficient channel utilization and a reduction in the long-term sustainable playback rate. The extreme case of asking for only a single byte in each request clearly indicates the existence of a lower bound of the segment size. In this case, the metadata in each request exceeds the payload by a large margin, wasting most of the available bandwidth. In the next section, we attempt to optimize multipath progressive download by the use of HTTP request pipelining, including a more in-depth discussion of how to properly adjust the segment size.

6.2.3 The Benefit of Request Pipelining

As the results in Section 6.2.2 were collected in the absence of request pipelining, a round-trip time was "wasted" for each request, causing the server to often wait for a new request instead of transmitting data. For very small segment sizes, the server therefore remains in a state of idle waiting for most of the time, which can reduce the aggregation efficiency to a point where it becomes beneficial to only use a single interface.

HTTP request pipelining can be used to eliminate this idle time at the server, ensuring that each client-side network interface receives data at the connection's full capacity at all times. A small problem with pipelining, however, is to decide *when* to send the next/pipelined request (these decision points are illustrated with small triangles in Figure 6.1). Requesting the next segment too late results in idle time at the server and degrades the performance of pipelining, and requesting it too early reduces the robustness to throughput fluctuations. Since our main goal is to fully utilize the available paths, we avoid late requests by a predefined pipelining pattern at the very beginning of a multipath progressive download, the *interleaved startup phase*.

Interleaved Startup Phase

To guarantee that the server always has at least one pipelined segment ready for processing, each multipath progressive download begins with two range requests per client interface. As shown in Figure 6.8, the byte ranges of these initial requests are alternately assigned to the involved interfaces in an interleaved pattern.

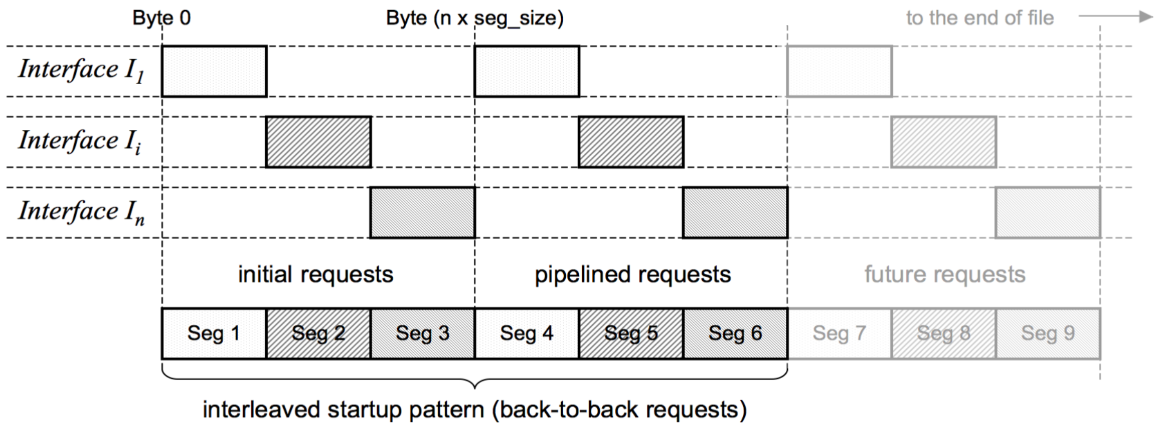


Figure 6.8: *Our pipelining mechanism is initiated by requesting the first segments over interfaces I_1, ..., I_n in an interleaved startup pattern.*

After the interleaved startup phase, the dilemma of when to send the next request is eliminated. Each interface may simply issue a new request right after completing a segment and the server's pipeline always contains at least one request to be processed. Implementing such a pipelining mechanism leads to a consistent increase in the long-term sustainable playback rate. Figure 6.9 illustrates the benefit of request pipelining through an experiment of simultaneously downloading a 50 MB large file multiple times over the $HSPA_1$ and $WLAN_1$ networks.

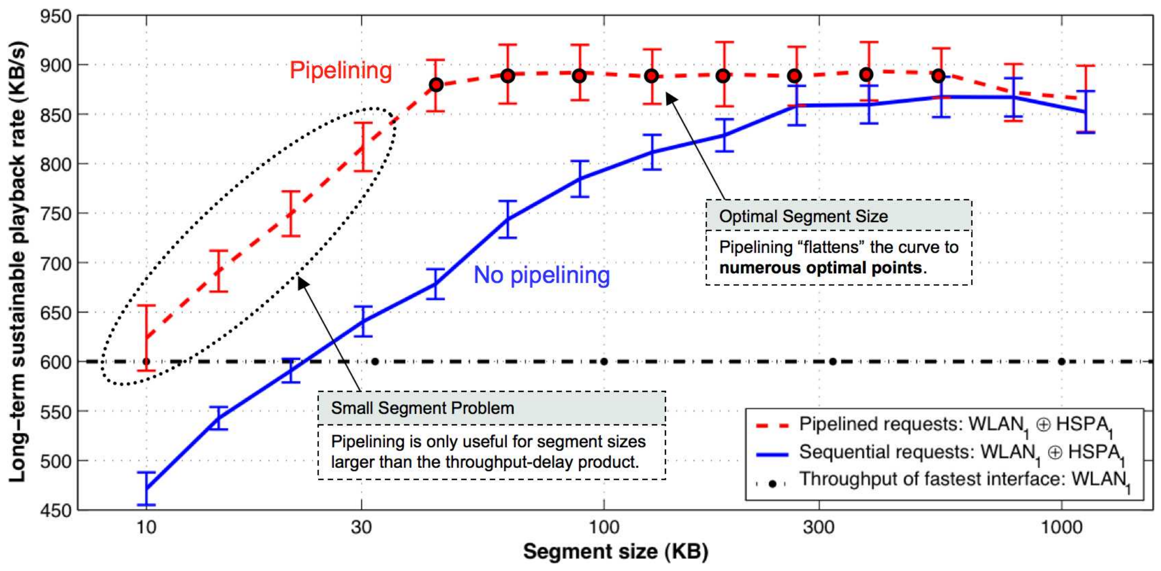


Figure 6.9: *Using a pipelining mechanism eliminates the waiting time between successive requests and effectively increases the aggregated throughput.*

Especially when using small segments, the benefit of pipelining becomes evident. In the absence of pipelining, Figure 6.6 suggested segment sizes of up to a megabyte for achieving maximum throughput. With the use of pipelining, as pointed out by Figure 6.9, the range of optimal segment sizes is stretched to values as small as 50 KB. These observations imply that pipelining can be used to guarantee a short startup latency and minimal buffer requirements, while achieving close to ideal multipath aggregation.

Even with a pipelining mechanism, however, multipath aggregation sharply turns inefficient when using segment sizes below a certain threshold. We refer to avoiding this threshold as the *minimum segment size problem*.

The Minimum Segment Size Problem

In the attempt of achieving a smooth playback experience by employing a very fine-grained segmentation, there exists a risk of choosing the segment size too small. The encircled results in Figure 6.9 are examples of sizing the segments so small that even pipelining is unable to continuously keep the server busy sending data. In these cases, the server spends such a short instant to process each request that the client has no time to push the next request into the pipeline. The minimum segment size problem takes effect when the one-way transmission delay d_i over interface I_i is larger than the time t_i it takes for the HTTP server to fully process a request. If the server is able to process requests faster than the client is able to send them, then pipelining loses its effect. The time t_i is given by the segment size S and the average throughput Φ_i experienced over interface I_i:

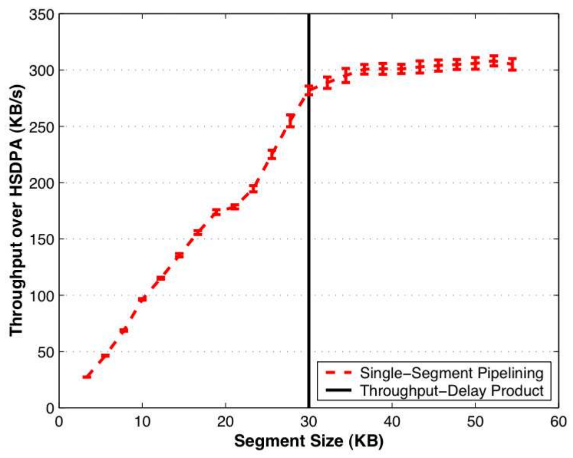
$$t_i = S/\Phi_i \tag{6.5}$$

As an example, for a 10 KB file segment over the HSPA$_1$ interface (which has an average throughput of 300 KB/s), the server requires 33 ms until the last IP packet of a requested file segment is transmitted. These 33 ms are much less than the average one-way transmission delay $d_i = 110\,\text{ms}$ experienced over the HSPA$_1$ interface (in other words, t_i is smaller than d_i). Hence, the efficiency of pipelining is diminished, because the server spends only 33 ms sending data over the HSPA$_1$ network and staying idle for the remaining 77 ms. In other words, the segments should be big enough to keep the server busy processing each request for the duration of a one-way path delay. The minimum segment size S_{min} required for efficient pipelining can thus be calculated as follows:

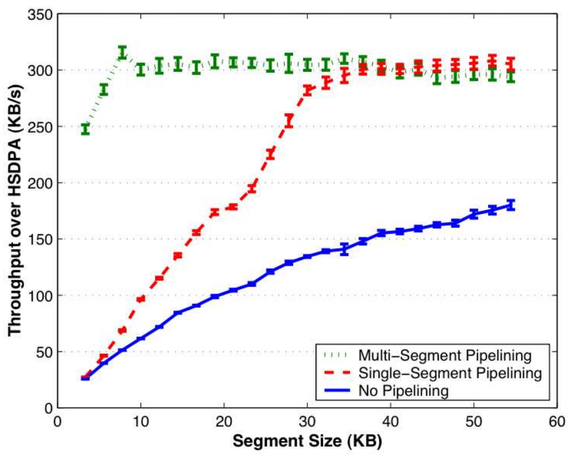
$$S_{min} = d_i * \Phi_i \tag{6.6}$$

For a full utilization of the HSPA$_1$ interface, segments should therefore be sized to at least $110\,\text{ms} * 300\,\text{KB/s} = 30\,\text{KB}$, which is equivalent to the throughput-delay product of the path. Figure 6.10(a) illustrates our attempt to experimentally determine this value of $S_{min} = 30\,\text{KB}$. Without trying to achieve path aggregation, only the HSPA$_1$ network was used in this test, but the target resource was anyway split into fixed-size segments. Almost precisely at 30 KB, the graph reaches the typically experienced throughput of 300 KB/s, while segments smaller than 30 KB cause a significant loss in throughput.

The minimum segment size problem can be alleviated by either enlarging the segments or, equivalently, by extending the interleaved startup phase to include more than just one pipelined request. The performance gain of pipelining multiple segments during the startup phase is shown in Figure 6.10(b). While single-segment pipelining incurs



(a)



(b)

Figure 6.10: *The minimum segment size problem takes effect at segment sizes smaller than the throughput-delay product → (a). A possible solution is to pipeline not only one, but multiple segments → (b).*

a rapid decrease in throughput for segment sizes below the bandwidth-delay product, multi-segment pipelining achieves close to optimal throughput with file segments sizes in the order of a few IP packets. A potential negative effect of multi-segment pipelining may become apparent with severe throughput fluctuations. For example, the throughput may suddenly drop or stall during the download of a segment, while several subsequent segments are already pipelined. In this case, pipelining only a single segment would incur less head-of-line blocking, because an interface with a more stable throughput could take over the task of downloading the prematurely requested data. As an alternative to pipelining multiple segments, a dynamic segment size could be used – a similar method will be presented as part of a quality-adaptive streaming platform in Section 6.3.

Before diving into the details of a practical video streaming implementation, we first evaluate the combined advantages of HTTP range requests and HTTP request pipelining from a more abstract point of view. For simplicity (i.e., to avoid dependence on video content and codecs), we assume that a video player is a constant-bitrate file reader. This allows our solution to be judged by a simple but demanding performance metric: a multipath progressive download without any playback interruption is counted as a success, while a download with only a single occurrence of the constant-bitrate file reader seeking ahead of the received data is counted as a failure.

Figure 6.11 compares the performance of the WLAN$_1$ interface versus the additional use of the HSPA$_1$ interface. Each resulting bar is based on 100 downloads of a 50 MB large file, divided into fixed-size 100 KB segments. After a startup delay of 3 seconds the received bytes are read sequentially at a constant bitrate. Clearly, the simultaneous use of both paths increases the probability of an interruption-free playback experience at a targeted bitrate. For example, the WLAN network used in these experiments, which does not perform reliably at target bitrates above 400 KB, can be made 100% reliable for streaming up to 600 KB/s by using an additional HSPA interface. Another way to interpret these results is that for a target constant-bitrate stream of 800 KB/s, the multipath solution achieves a completely smooth playback in 65% of the time, which is simply impossible using the WLAN_1 interface alone.

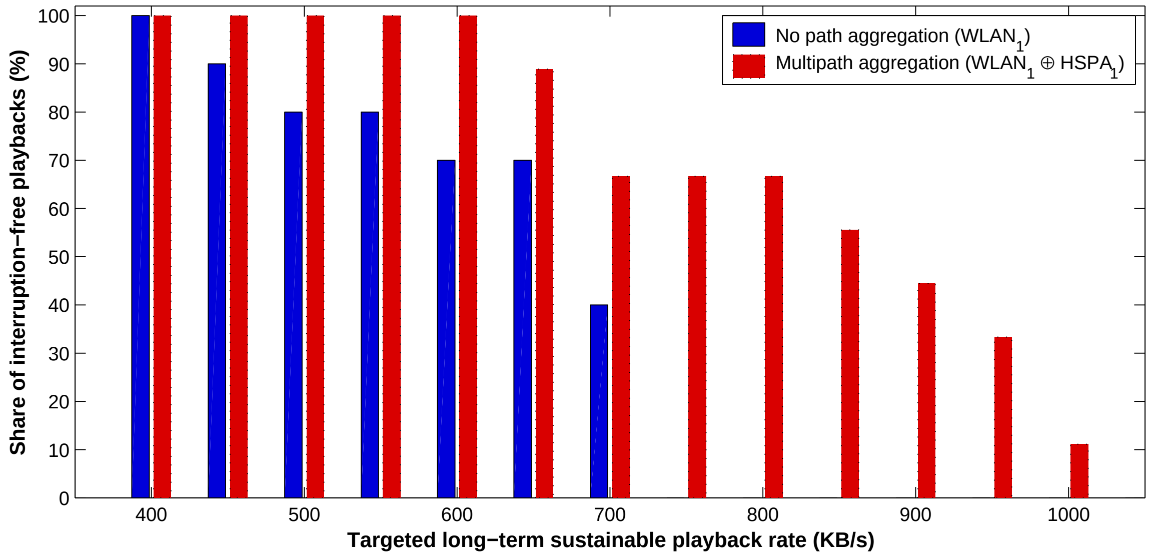


Figure 6.11: *Pipelining-based download over multiple heterogeneous interfaces achieves a higher quality of progressive video playback.*

6.3 Quality-Adaptive Streaming over Multiple Paths

In this section, we validate the combined methods of file segmentation and request pipelining by implementing them into DAVVI, a video streaming platform originally developed by Johansen et al. [85]. After a description of DAVVI's architecture and its extension for multihoming in Section 6.3.1, we verify in Section 6.3.2 whether our insights from an abstract model of multipath progressive download are applicable in a practical system. We evaluate how the video quality over multiple paths is affected by bandwidth and latency heterogeneity.

6.3.1 The DAVVI Streaming Platform

Similar to systems like Move Networks [121], Microsoft's Smooth Streaming [171] and Apple's HTTP Live Streaming [12], DAVVI uses file segmentation for adapting the video quality according to the experienced data transfer speed. High-quality segments are downloaded while the client observes a fast connection, but as soon as the system detects a throughput decrease, it switches to segments of lower quality (which are smaller and hence require less time for transmission). As shown in Figure 6.12, DAVVI makes every full-length video available in different quality encodings (in this case, four quality levels $Q_1, ..., Q_4$). Additionally, each video is split into small movie clips with a length of two seconds. This duration is motivated by a study by Ni et al. [122] that identified period lengths below two seconds as likely to introduce flickering. Since all segments are independent video files, they can be arbitrarily merged for smoothly switching between quality levels as the network throughput changes over time.

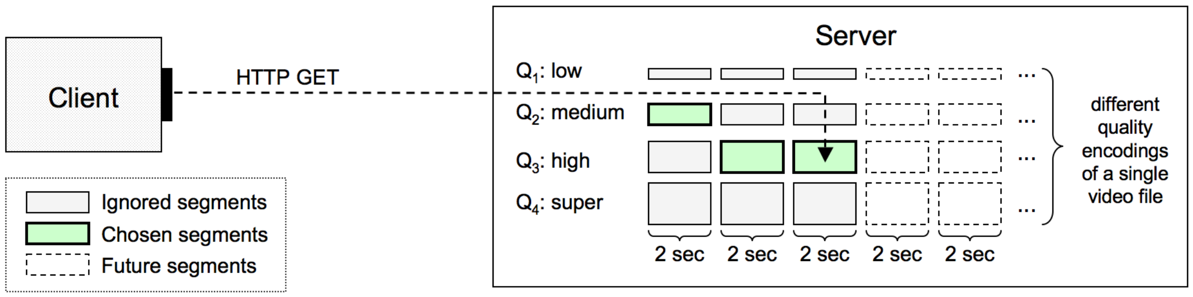


Figure 6.12: *DAVVI allows the client to dynamically adapt the video streaming quality by downloading independent video segments stored in different quality levels.*

Extending DAVVI with Support for Multiple Client Interfaces

DAVVI's pull-based mode of operation and its use of HTTP as an underlying network protocol allow a fairly straightforward integration of our ideas of pipelined request multiplexing over more than a single access network. However, DAVVI's quality-adaptive method of operation is more demanding than simply downloading a file and sequentially parsing it. This therefore increases the complexity of adding support for multiple interfaces. Particularly, DAVVI already uses predefined, two-second blocks of "playable data", which can be as large as several megabytes, much bigger than the 100 KB segments we used in Section 6.2.3 to optimize the performance of multipath progressive download. In order to overcome this limitation, we establish a new tier of segmentation by dividing the two-second video clips in even smaller *subsegments*. Figure 6.13 illustrates our extended concept of subsegments that allows multiple client interfaces to collaboratively download a video segment of a given quality level. Algorithm 5 provides pseudo-code of the extended request scheduler.

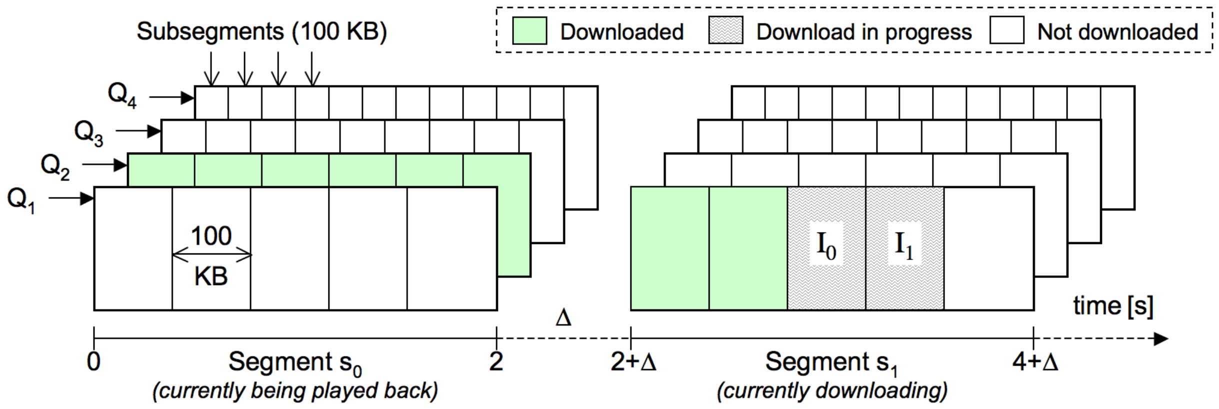


Figure 6.13: *In this example, two client interfaces I_0 and I_1 have finished downloading segment s_0 at quality level Q_2. As a drop in throughput was measured, they currently collaborate on downloading a lower-quality segment.*

DAVVI's two-second granularity also has implications on the startup latency and data buffering. Due to the fact that only fully downloaded video segments are useable by a video player, an entire segment must initially be downloaded before playback can commence. Furthermore, after a quality level has been decided, it is not possible to adjust it during the download of a segment[4]. On the upside, the fixed-size length of the video clips allows us to intuitively quantify the client-side buffer requirements in terms of segments, instead of bytes. Under DAVVI's assumption that the client always fills its buffer prior to playback, an example buffer size of five segments implies a startup latency of ten seconds.

Algorithm 5 DAVVI's extended request scheduler

```
 1: Initialize quality_level = 1 (use lowest quality to reduce the startup latency)
 2: Request a subsegment over each interface
 3: Request a "pipelined" subsegment over each interface
 4: while data is received do
 5:     Write data to buffer
 6:     if data == complete subsegment then
 7:         if data == complete segment then
 8:             Forward the completed segment to the video player
 9:             Update quality_level based on measured transfer speed
10:         end if
11:         Request next subsegment at current quality_level
12:     end if
13: end while
```

6.3.2 The Performance of Multipath-enabled DAVVI

As an initial experiment, the multipath-enabled extension of DAVVI was tested in a practical environment using the wireless access networks $WLAN_1$ and $HSPA_2$, with a client-side buffer holding a maximum of two segments, thus introducing four seconds of startup delay. The video streaming involved a sample movie with a length of 104 minutes (3127 segments), encoded in four quality levels with properties as listed in Table 6.1.

Table 6.1: Quality Levels and Bitrates of the Sample Movie

Quality level	Low (Q_1)	Medium (Q_2)	High (Q_3)	Super (Q_4)
Minimum bitrate (Kbit/s)	524	866	1491	2212
Average bitrate (Kbit/s)	746	1300	2142	3010
Maximum bitrate (Kbit/s)	1057	1923	3293	4884

As the video segments are encoded at a variable bitrate, they all consist of a different number of bytes, which creates a challenge to fairly compare the performance of different test runs. However, designing an objective quality metric is out of the scope of this thesis, and we therefore opt for plotting the distribution of the received segments quality levels. The difference between the quality levels is sufficiently large to justifiably interpret an increase in the number of super-quality segments as an improvement of the user-perceived playback experience.

Figure 6.14 depicts two video quality distributions and clearly illustrates how the DAVVI's performance in terms of perceived video quality might increase when a WLAN

[4]unless one is willing to discard partially received data (a tradedoff we leave for future work).

interface is supplemented by an HSPA interface. Compared to the case of exclusively using the WLAN interface, the resulting distribution of quality levels shows a 98% increase in the number of super-quality segments when the WLAN and HSPA interfaces are used in collaboration. These results confirm that multipath aggregation by logical segmentation and request pipelining is able to enhance the quality of video streaming.

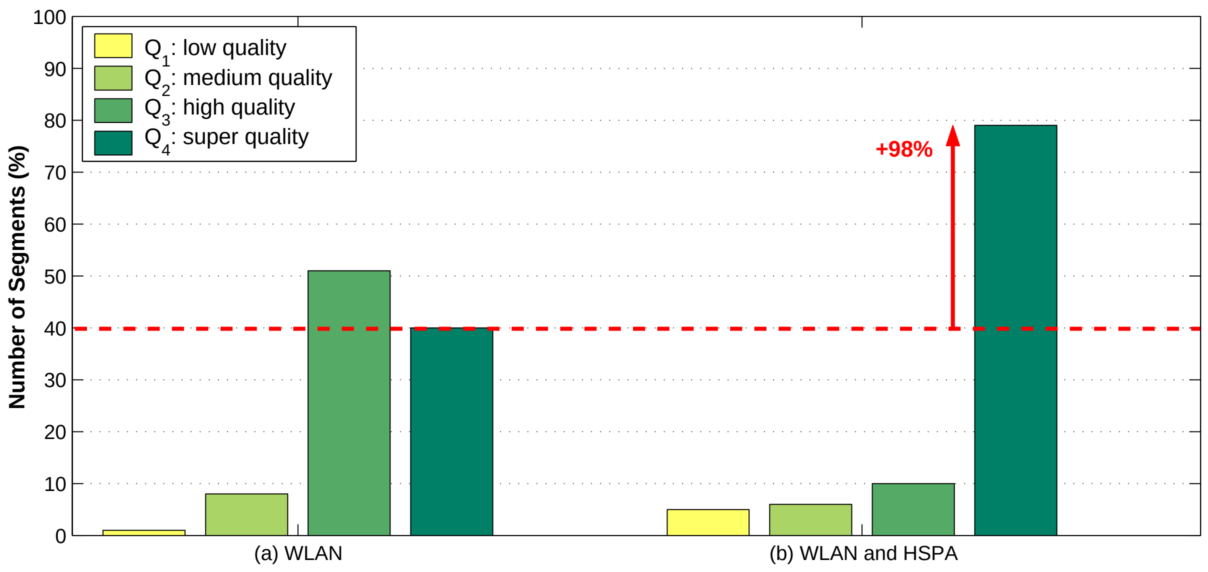


Figure 6.14: *The average quality distribution of downloaded video segments when the DAVVI client is connected to (a) WLAN only, and (b) WLAN and HSPA simultaneously.*

The Last Segment Problem (revisited)

Although the wireless networks used in this experiment are subject to frequent variations in throughput and latency, they normally experience only moderate differences in throughput. On average, the throughput over WLAN_1 is about 30% higher than the one observed over HSPA_2. For a larger throughput disparity between the used paths, it is forseeable that the last segment problem (previously discussed in Section 6.2.2) comes back into effect. In the context of DAVVI, this problem should more appropriately be called the "last *sub*segment problem", because it can repeatedly occur for the last subsegment of each video segment. Figure 6.13 serves as an illustrative example of the problem at hand: two interfaces are in progress of downloading, while only a single, last subsegment is outstanding. Assuming that interface I_0 is much faster than I_1, a significant drop in performance is expected if this last subsegment happens to be assigned to the slow interface I_1. If the entire movie consists of thousands of segments, a performance loss at every segment may accumulate to a major reduction in the overall aggregation benefit.

Tests of DAVVI in our emulation testbed (→ Section 3.3), which allows experiments under controlled path conditions, have indeed confirmed a loss in performance when the transfer speed of the involved interfaces is significantly different. Based on an experiment in which a bandwidth of 3 Mbit/s was allocated in an 80:20 ratio to two paths, Figure 6.15 shows a 37% performance drop in using both paths, as opposed to using only the faster of the two. A possible solution to this deficiency is to dynamically adjust the subsegment sizes according to the observed throughput, instead of using fixed-size segments. In this example, the strategy of dynamic subsegments restores the previously observed multipath aggregation benefit – a 44% increase in the number of super-quality segments is achieved, despite the severe bandwidth heterogeneity.

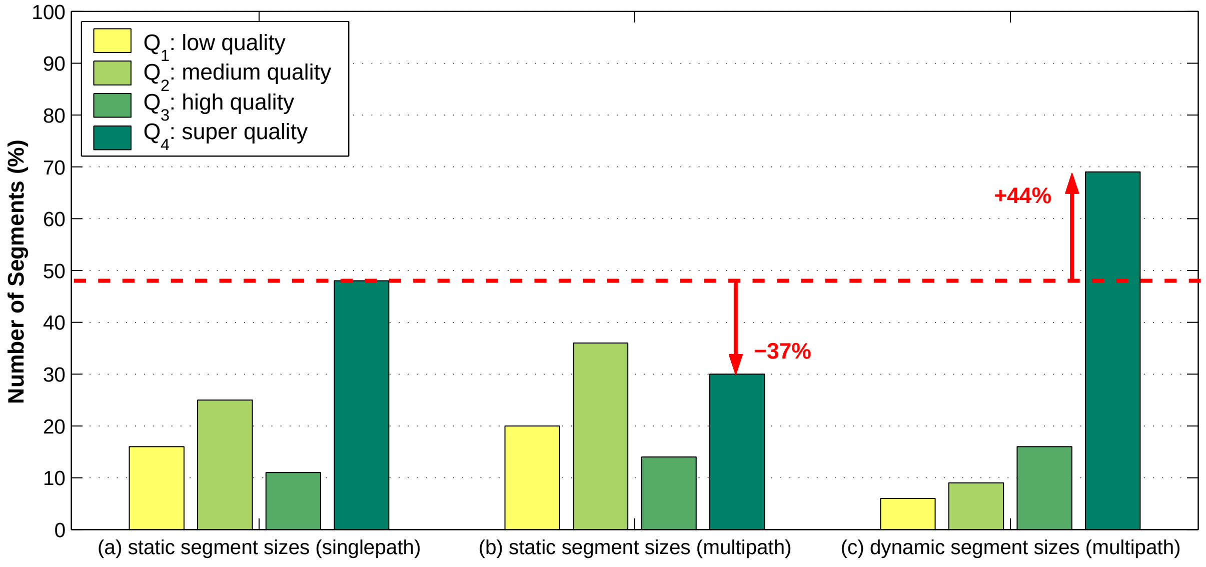


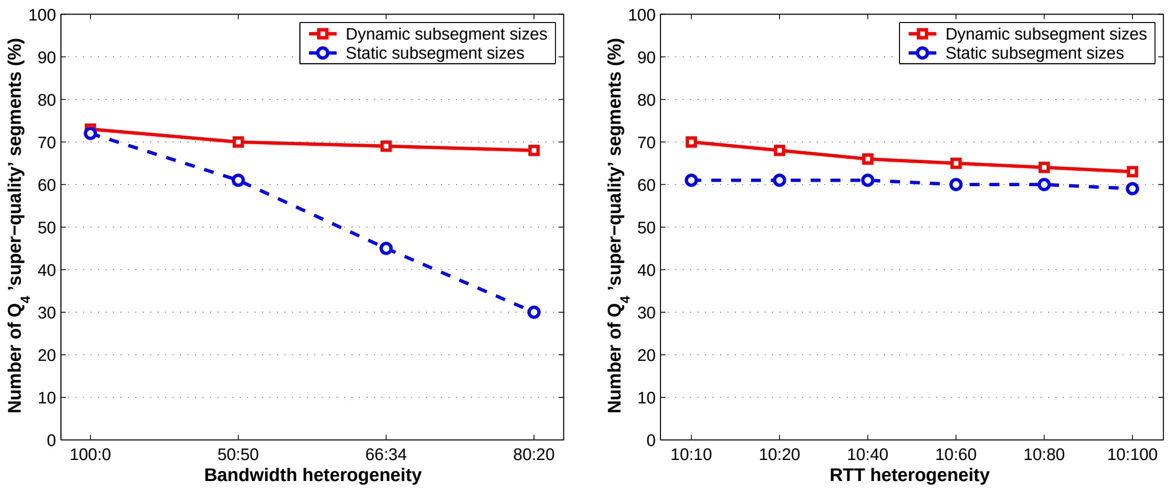
Figure 6.15: *In this experiment with two paths of 2.4 and 0.6 Mbit/s bandwidth (a 80:20 ratio), it is preferable to (a) use the fastest path alone over (b) aggregating them both. However, (c) with a dynamic subsegment approach, an aggregation benefit can be achieved.*

The Dynamic Subsegment Approach

The main idea behind the dynamic subsegment approach is to assign the data transfer workload in proportion to the available interfaces' throughput. However, simply splitting a video segment in as many pieces as there are interfaces might render the performance susceptible to throughput variances and measurement errors. In our approach, subsegments are therefore proportionally sized to sum up to $n * 100$ KB, where n is the number of interfaces. For example, if the current throughput ratio of two interfaces A and B is 1:3, then interface A downloads 50 KB subsegments and interface B requests 150 KB.

Figure 6.16 provides more evidence of how dynamically adjusted subsegments are able to mitigate the adverse effects of the last segment problem. In Figure 6.16(a), DAVVI was tested over two paths configured to share a total bandwidth of 3 Mbit/s in the given $X{:}Y$ ratio. Evidently, compared to using fixed-size 100 KB subsegments, the dynamic subsegment approach is much less prone to bandwidth heterogeneity and only experiences a slight loss in performance as the heterogeneity increases. Even at a bandwidth ratio of 80:20, the number of streamed super-quality segments is similar to the performance of a single path with the full bandwidth (i.e., the 100:0 case).

For completeness, we also ran the same streaming experiment with different levels of RTT heterogeneity ($\rightarrow$ Figure 6.16(b)). In this case, DAVVI was tested over two paths with an equal bandwidth of 1.5 Mbit/s each, but with respective emulated round-trip times of X and Y milliseconds (also denoted as $X{:}Y$). As an expected result of request pipelining, different RTTs have little impact on the video streaming performance and the benefit of using dynamic segment sizes is only marginal.



(a) Bandwidth heterogeneity (b) RTT heterogeneity

Figure 6.16: *DAVVI's video streaming performance over two interfaces, with emulated path hetergeneity in terms of (a) bandwidth, and (b) round-trip time.*

6.4 Discussion

In this chapter, an application-layer solution to multipath aggregation was introduced, which satisfies the majority of the evaluation criteria listed in Section 2.1.1. Our results show that if file segmentation and request pipelining are applied properly, an aggregation benefit is achieved even in highly heterogeneous multipath scenarios (fulfilling criteria C1 to C3). In addition, relying only on standard features of the widely used HTTP protocol, allows the solution to be deployed without any server-side modifications and without introducing additional infrastructural devices (satisfying the criteria C5 and C7).

However, although HTTP is the dominant protocol on the World Wide Web, there are many applications that are not based on HTTP. The criterion C4 of being compatible with all applications is thus only partially fulfilled by the presented solution. In addition, the criterion C6 of leaving client software unchanged is clearly violated, because the client application is responsible for issuing requests and selecting the appropriate interface and subsegment size. With the DAVVI streaming platform as an example, extending an existing application with efficient support for multiple interfaces may lead to new tradeoffs and challenges. We have shown that these issues are resolvable and that our solution is a worthwhile enhancement to existing applications. Nevertheless, there are some remaining aspects the presented solution does not address and for which a brief discussion is therefore provided in the following.

Robustness to Network Failures

Although the use of small segments is an effective method of adapting to throughput variances, it is not robust to significant drops in throughput or to connections that stall for an indefinite amount of time. If some data loss is tolerable (such as in video streaming), rescheduling of the stalled segments may be avoided. Otherwise, if data integrity is required (for example in file transfers), parts of a segment and all pipelined request must be rescheduled over another interface. Since HTTP does not allow pipelined requests to be cancelled, this case requires the underlying TCP connection to be aborted and re-established. Alternatively, it might be advantageous to redundantly request missing data over an interface with stable throughput. We did not consider such tradeoffs in our solution, although failure detection and finding the appropriate moment of data recovery is a crucial aspect of a fully functional system.

A related problem is to detect whether the additional use of an interface is beneficial to the overall performance. If the path heterogeneity is very high, the gain in throughput may be outweighed by the head-of-line blocking introduced through the additional use of an interface. A complete solution should thus be able detect and ignore interfaces that do not contribute to the overall performance.

Live Streaming

The ideas presented in this chapter were tested in scenarios of progressive download without considering the timeliness of data. In live streaming solutions, content is typically recorded by a camera and made available for download on one or multiple servers. A perfect *liveness* would be achieved by a player capable of immediately playing back the content recorded by the camera, without introducing any additional delay. In reality, a certain loss of liveness is unavoidable, because data transfer, storing and encoding add to the overall delay between the real event and the displayed picture on a user's screen.

In the example of DAVVI, the recorded content is stored in two-second segments, which means that two seconds of delay are already lost through the procedure of storing them on the server. After the first segment is available on the server, the client needs additional time to download the segments and start its playback. DAVVI therefore achieves the highest degree of liveness by buffering only a single segment (the one currently being downloaded). However, this increases the risk of lagging playback, because every segment must be downloaded within a strict limit of two seconds. If this deadline is missed, the video player does not have any data available and a playback interruption is unavoidable. In order to reduce the number of deadline misses, a larger buffer could be used. However, filling up a larger buffer normally requires a longer startup latency, which in turn reduces the liveness. Hence, there exists a tradeoff between the number of deadline misses and the liveness of a stream. In addition, if deadlines are missed numerous times, the video stream will gradually lose its liveness with each playback interruption. If the total lag surpasses the segment duration of two seconds, DAVVI may decide to skip a segment in order to "catch up" to the live stream.

In [48], we provide a more in-depth discussion of live streaming, taking into account deadline misses and skipped segments as part of the user-perceived quality of experience. Our main conclusion is that the simultaneous use of multiple network interfaces helps to reduce both deadline misses and skipped segments in live streaming scenarios. However, for a solution to optimally solve these tradeoffs, the subjective user preferences must be fully understood. Some viewers may be easily disturbed by playback lags and skipping, while others may wish to be as close to the live event as possible.

A Lower Bound on the Segment Size

While presenting our solution and evaluating its performance, we have not fully discussed the existence of a lower bound for the number of bytes requested with each HTTP request.

In fact, there are several reasons why segments (or in DAVVI's terminology: subsegments) should not be smaller than a certain threshold:

1. As discussed in Section 6.2.3, pipelining loses its efficiency when the requested data is smaller than the throughput-delay product of the path.

2. When dynamically dividing a segment into subsegments proportional to the observed throughput, as performed in Section 6.3.2, requests for unreasonably small amounts of data may be issued. This may occur if the path heterogeneity is very large or when the throughput is estimated inaccurately. In order to avoid "silly" small segments, our implementation currently uses a lower limit of 10 KB for the subsegment size. However, this is only a temporary solution and should be optimized in future versions.

3. The absolute lower bound of the segment size depends on the metadata overhead, most notably on the number of characters in the hostname and the path statement of the requested resource. For large segments, this constant overhead is negligible, but its weight increases with the use of small segments. For example, if the request overhead is 200 bytes, and no more than 1% loss in aggregation efficiency is desired, then segments should be at least 20 KB large.

Chapter 7

Conclusions

In this thesis, we examined the problem of accelerating data transmission by using multiple network interfaces of different technologies simultaneously. Due to the growing number of devices that are sold today with built-in network interfaces of heterogeneous technologies (such as WLAN and 3G), this problem of *multipath aggregation* has recently become an interesting research topic. We explored various layers of the network protocol stack for efficient solutions to use such different technologies in parallel. We analyzed the multipath performance of existing systems and introduced several mechanisms for aggregating the available bandwidth of two or more Internet connections on a single host. In order to allow for a quick deployment, we targeted practical methods that avoid modifications to servers, third-party equipment and existing network protocols. As our proposed mechanisms and performance evaluations are based on network emulation and practical implementation, we believe that our work prompts and stimulates the deployment of multipath solutions.

7.1 Summary

Routers in the Internet forward data traffic on a per-flow basis. During normal operation, this implies that all IP packets belonging to the same end-to-end communication flow over a single path. This rule of keeping each data flow on the same path minimizes IP packet reordering, a phenomenon known for its negative effects on higher-layer protocols and applications. The main difficulty of the multipath aggregation problem is that it breaks this rule and introduces reordering. As we have measured in Section 4.1, scheduling IP packets over access networks with different latencies can result in IP packet reordering that is an order of magnitude higher than observed on typical end-to-end paths. For solving this dilemma between host-based multipath aggregation and IP packet reordering, we have pursued the following three approaches:

1. Actively reducing packet reordering using a delay-equalizing buffer.

In Chapter 4, we reasoned that multipath aggregation would be easy if there was a way to fully eliminate its side-effect of IP packet reordering. To this end, we introduced NAMA, a <u>n</u>etwork-layer <u>a</u>daptive <u>m</u>ultipath <u>a</u>ggregation proxy, which forwards packets over multiple paths, while trying to compensate for packet reordering. NAMA constantly monitors the network characteristics and actively equalizes the end-to-end path latencies in the attempt of achieving in-order delivery at the receiver. Under stable path conditions, NAMA achieves efficient multipath aggregation without introducing packet reordering. When the path latencies vary over time, the precision of NAMA's path monitoring is insufficient to fully eliminate packet reordering, but a significant reduction is achieved.

2. Performing multipath forwarding and letting TCP cope with reordering.

In Chapter 5, we challenged the unwritten law of flow-based forwarding on the IP layer. Even though standard TCP's fast retransmit algorithm fails when a TCP connection is diverted over paths with heterogeneous bandwidth and latency characteristics, a modern operating system might perform much better. Due to a lack of published results on TCP's actual behavior in practical multipath scenarios, we evaluated a full-featured and widely used Linux TCP implementation for its robustness against packet reordering and its capability to aggregate multiple paths.

In a first experiment with practical and emulated multipath scenarios, we showed that Linux TCP achieves a considerable aggregation benefit when IP packets of a single TCP connection are forwarded over two heterogeneous networks. Our example scenario of diverting a TCP flow over HSPA and WLAN networks resulted in a large average aggregation benefit of 59.6% in practical networks and 75% in the emulated scenario.

In a second experiment, we expanded the network heterogeneity to a wide variety of combined path properties and numerous TCP-related parameters, including different congestion control algorithms and loss recovery mechanisms. Through a large set of emulation-based experiments, we identified several Linux TCP parameters that lead to improved multipath performance. By correctly tuning these parameters, we managed to show a 30% increased aggregation benefit compared to Linux TCP's default parameter settings. In addition, our tuned Linux TCP significantly outperformed TCP-DCR in a direct comparison.

These results tempt us to conclude that it is time for TCP to finally be extended to fully cope with IP packet reordering. Some performance-enhancing features (such as TCP timestamps, SACK, and an adaptive dupACK threshold) already exist, but they are not normally used for achieving multipath aggregation. We therefore believe that a dedicated set of enhancements, which specifically target TCP's performance in multipath scenarios, might lead to a consistent aggregation benefit with any type of path heterogeneity.

3. Using an independent end-to-end connection per path.

In Chapter 6, we presented a method for downloading a single resource from a standard Web server over multiple client interfaces, without introducing neither modifications to the server nor a proxy. While this sounds impossible at first, we achieved efficient multipath aggregation by exploiting HTTP's range retrieval requests, a feature that is available on all modern Web servers.

The main idea behind our approach is for the client to open independent connections to the same server over each network interface. These multiple connections then collaborate by downloading different segments of the same resource, sharing the workload and resulting in a high aggregation benefit. In addition, we applied HTTP request pipelining to further improve the performance and verfied these algorithms by implementing them into DAVVI, an existing video streaming platform.

7.2 Future Research Directions

We have presented and evaluated a variety of techniques to efficiently aggregate the available bandwidth of heterogeneous access connections. However, the research field related to multipath aggregation is still in its early stage and there are several interesting topics remaining for future work:

Generic Multilink Framework From the experiments with NAMA in Section 4.2, we learned that compensating for packet reordering at the sender or at a proxy is not a viable approach to achieve efficient multipath aggregation if the path heterogeneity is large and rapidly changing over time. A more promising method could involve the receiver to restore order in the packet requence, similar to the approach followed by MPTCP [56]. An improved version of the NAMA prototype that has already picked up momentum is our generic and modular MULTI [46] framework, which targets easy deployment and multipath support for all transport protocols and applications.

Reorder-robust TCP The promising results from our multipath experiments with TCP in Section 5.2 lead us to believe that a reorder-robust transport layer is feasible. A truly reorder-robust TCP would lead to more freedom to the network layer and imply a significant relief of workload to routers.

Vertical Handover The methods proposed in this thesis were tested in static scenarios. In cases when mobile devices move between the coverage areas of heterogeneous access networks, it is desirable for multipath aggregation to automatically take into consideration newly discovered networks. While handover between networks of the same technology and vertical handover between networks of different technologies is already well studied, the combination with multipath aggregation is a promising new idea. Our first step into this direction is presented as a practical demo in [47].

Energy Efficiency An important issue with the concurrent usage of multiple Internet connections is the increased energy required for powering several network interfaces. Especially for users of mobile devices, a long battery life is crucial. Evaluating the tradeoffs between multipath aggregation and power consumption, as well as finding techniques for prolonging battery life, would increase the popularity and aid the deployment of our solutions. Last but not least, combining our work with energy-efficient ESP-based forwarding, as documented in RFC 5984 [116], should be considered.

Appendix A

List of Publications

This chapter lists all the scientific contributions that emerged in the course of this PhD project. Each piece of work is shortly described and the individual contributions of the authors are explained.

A.1 Refereed Proceedings

MMSys 2011 K. R. Evensen, D. Kaspar, C. Griwodz, P. Halvorsen, A. F. Hansen, and P. E. Engelstad. "Improving the Performance of Quality-Adaptive Video Streaming over Multiple Heterogeneous Access Networks", In: Multimedia Systems (MMSys), 2011 [48].

This paper is an extension of the Nossdav 2010 work [45]. The major contribution to both writing and performance measurements was made by Kristian Evensen. Pål Halvorsen and Dominik Kaspar contributed to a smaller extent by writing and providing illustrative figures. All authors were equally involved in regular discussions on the content and the structure of the paper.

Nossdav 2010 K. R. Evensen, T. Kupka, D. Kaspar, P. Halvorsen, and C. Griwodz. "Quality-Adaptive Scheduling for Live Streaming over Multiple Access Networks", In: International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV), 2010 [45].

This paper describes and evaluates a practical implementation of the discoveries made in our previous papers on HTTP-based progressive download over multiple paths [90, 88]. Kristian Evensen and Tomas Kupka were the driving forces behind programming the multihoming support for DAVVI, while Kristian Evensen carried out all the performance measurements. Dominik Kaspar contributed by writing the section on related work and parts of the architectural section. Pål Halvorsen was the main initiator and motivated this work by providing a detailed description of DAVVI. All authors took part in technical discussions and provided feedback on the textual content and structure.

ICC 2010 D. Kaspar, K. R. Evensen, P. E. Engelstad, and A. F. Hansen. “Using HTTP Pipelining to Improve Progressive Download over Multiple Heterogeneous Interfaces”, In: International Conference on Communications (ICC), IEEE, 2010 [88].

This paper is a continuation of the CCNC 2010 paper [90]. Dominik Kaspar and Kristian Evensen equally contributed to extending our previous prototype with code for pipelining support. The performance analysis was carried out by Dominik Kaspar, who was also responsible for all the text and figures. All authors provided feedback on the paper's content and involved in numerous technical discussions.

CCNC 2010 D. Kaspar, K. R. Evensen, P. E. Engelstad, A. F. Hansen, P. Halvorsen, and C. Griwodz. “Enhancing Video-on-Demand Playout over Multiple Heterogeneous Access Networks, In: Consumer Communications and Networking Conference (CCNC)”, 2010 [90].

This paper describes a method of downloading files over multiple paths using HTTP range retrieval requests. To our best knowledge, this idea came from Kristian, who also programmed a prototype tool for testing. With the use of this tool, Dominik Kaspar conducted the performance easurements and analysis. All text and figures were made by Dominik Kaspar. All authors frequently met for technical discussions and were equally involved in providing feedback on the text.

LCN 2009 K. R. Evensen, D. Kaspar, P. E. Engelstad, A. F. Hansen, C. Griwodz, and P. Halvorsen. “A Network-Layer Proxy for Bandwidth Aggregation and Reduction of IP Packet Reordering, In: The 34rd Annual IEEE Conference on Local Computer Networks (LCN)”, 2009 [44].

This publication includes several concepts and ideas (e.g., the send vector approach and the delay equalizer), that were developed and tested together by Kristian Evensen and Dominik Kaspar. Kristian Evensen was responsible for implementing the code, while Dominik Kaspar was in charge for writing the text and creating all the figures. The other coauthors took part in the evaluation of the results and provided feedback on the paper's content and structure.

ISCC 2009 D. Kaspar, K. R. Evensen, A. F. Hansen, P. E. Engelstad, P. Halvorsen, and C. Griwodz. “An Analysis of the Heterogeneity and IP Packet Reordering over Multiple Wireless Networks”, In: IEEE Symposium on Computers and Communications (ISCC), 2009 [91].
[Runner-up for the best paper award]

This paper contains an analysis of how IP packets are reordered when subjected to heterogeneous network paths. In addition, first algorithmic ideas are introduced that we later referred to as the “delay equalizer” and the “send vector”. The performance evaluation and writing were done by Dominik Kaspar. The other authors contributed with discussions on the text and implementation advice.

A.2 Patent Applications

US12/713,939 D. Kaspar, K. Evensen, P. Engelstad, A. Hansen, C. Griwodz, and P. Halvorsen, "Data Segmentation, Request and Transfer Method", US Patent application (number 12/713,939), filed February 2010 [89].

The text of this patent application is based on the CCNC 2010 [90] and ICC 2010 [90] publications, which were written mainly by Dominik Kaspar. In several iterations, patent experts Adam Piorowicz and Tom Ekeberg translated the scientific content into "patent language", consulting Dominik Kaspar for feedback and technical correctness. All co-inventors were involved in meetings and discussions with the patent engineers.

A.3 Poster Presentations at International Venues

Trilogy 2009 K. R. Evensen, and D. Kaspar. "Bandwidth Aggregation over Heterogeneous Wireless Links", Poster, Trilogy Summer School, Louvain-la-Neuve, Belgium, 2009 [43].

This work summarizes 18 months of work on bandwidth aggregation over multiple heterogeneous wireless networks, including ideas from two conference publications. Kristian Evensen took full responsibility in writing this two-page abstract. Dominik Kaspar prepared the poster version and presented it at the seminar in Belgium.

ICNP 2008 D. Kaspar, A. F. Hansen, and C. Griwodz. "Multilink Transfer over Heterogeneous Networks", In: IEEE International Conference on Network Protocols (ICNP), Poster Session, ed. by Kevin Almeroth and K. K. Ramakrishnan, IEEE (ISBN: 987-1-4244-2507-5), 2008 [92].

This short paper describes the basic challenges of using multiple interfaces and presents preliminary measurement results on network heterogeneity. This work was initiated by Audun F. Hansen, while Dominik Kaspar had the main responsibility for the network measurements and the poster design. All authors contributed to the writing and discussions.

Nomenclature

3G	Third generation of mobile telecommunication standards, page 2
ACK	Acknowledgement, page 51
AIMD	Additive increase/multiplicative decrease, page 54
ARPANET	Advanced Research Projects Agency Network, page 1
ATM	Asynchronous Transfer Mode, page 11
DSACK	Duplicate Selective Acknowledgements, page 60
ECMP	Equal-cost Multipath (routing), page 5
F-RTO	Forward Retransmission Timeout, page 61
FACK	Forward Acknowledgements, page 59
FTP	File Transfer Protocol, page 15
HIP	Host Identity Protocol, page 12
HSPA	High Speed Packet Access, page 2
HTTP	Hypertext Transfer Protocol, page 86
ICMP	Internet Control Message Protocol, page 27
IEEE	Institute of Electrical and Electronics Engineers, page 11
IETF	Internet Engineering Task Force, page 15
IFB	Intermediate function block, page 29
IP	Internet Protocol, page 6
ISDN	Integrated Services Digital Network, page 11
ISP	Internet Service Provider, page 88
Kbit/s	1000 bit per second, page 1
LISP	Locator/Identifier Separation Protocol, page 12

MAC	Media Access Control, page 11
Mbit/s	1000000 bits per seconds, page 1
MIF	Multiple Interfaces, page 15
MOLS	Mutually Orthogonal Latin Squares, page 75
MPTCP	Multipath TCP, page 15
NAMA	Network-layer Adaptive Multipath Aggregation, page 43
NAT	Network Address Translation, page 43
PPP	Point-to-Point Protocol, page 11
RBD	Reorder Buffer Density, page 21
RD	Reorder Density, page 20
RE	Reorder Entropy, page 20
RFC	Request For Comments, page 9
RTO	Retransmission Timeout, page 52
RTP	Realtime Transport Protocol, page 16
RTT	Round-Trip Time, page 27
SACK	Selective Acknowledgements, page 58
SCTP	Stream Control Transmission Protocol, page 14
SRR	Surplus Round Robin, page 11
SRTT	Smoothed Round-Trip Time, page 52
TCP	Transmission Control Protocol, page 50
TPRRS	Type-P Reordered Ratio Stream, page 20
UDP	User Datagram Protocol, page 38
UMTS	Universal Mobile Telecommunications System, page 23
URI	Universal Resource Identifier [19], page 86
USB	Universal Serial Bus, page 26
WLAN	Wireless local access network, page 2
WWAN	Wireless Wide-Area Network, page 11

Bibliography

[1] Abd El Al, A., Saadawi, T., and Lee, M. LS-SCTP: A Bandwidth Aggregation Technique for Stream Control Transmission Protocol. *Elsevier Computer Communications* (2004).

[2] Abley, J., Lindqvist, K. E., Davies, E. B., Black, B., and Gill, V. IPv4 Multihoming Practices and Limitations. IETF RFC 4116, July 2005.

[3] Adhari, H., Dreibholz, T., Becke, M., Rathgeb, E. P., and Tuxen, M. Evaluation of Concurrent Multipath Transfer over Dissimilar Paths. In *Proceedings of the 1st International Workshop on Protocols and Applications with Multi-Homing Support (PAMS), Singapore* (March 2011).

[4] Adibi, S., and Erfani, S. A Multipath Routing Survey for Mobile Ad-Hoc Networks. In *IEEE CCNC* (2006).

[5] Adiseshu, H., Parulkar, G., and Varghese, G. A Reliable and Scalable Striping Protocol. *ACM SIGCOMM* (1996).

[6] Akamai. The State of the Internet. 3rd Quarter, 2010 Report, 2011.

[7] Allman, M., Balakrishnan, H., and Floyd, S. Enhancing TCP's Loss Recovery Using Limited Transmit. IETF RFC 3042, January 2001.

[8] Allman, M., Kruse, H., and Ostermann, S. An Application-Level Solution to TCP's Satellite Inefficiencies. In *Proceedings of the First International Workshop on Satellite-based Information Services (WOSBIS)* (Rye, New York, USA, 1996).

[9] Allman, M., Paxson, V., and Blanton, E. TCP Congestion Control. IETF RFC 5681, September 2009.

[10] Appenzeller, G., Keslassy, I., and McKeown, N. Sizing Router Buffers. In *SIGCOMM* (2004), ACM.

[11] Apple Inc. Mac OS X Server – QuickTime Streaming and Broadcasting Administration. Technical Report, 2007.

[12] Apple Inc. HTTP Live Streaming Overview. Technical Report, 2010.

[13] Association, I. S. Link aggregation for local and metropolitan area networks. IEEE Standard 802.1AX-2008, November 2008.

[14] Atheros. Super G – Maximizing Wireless Performance. White Paper, No. 991-00006-001, March 2004.

[15] Baiocchi, A., Castellani, A. P., and Vacirca, F. YeAH-TCP: Yet Another Highspeed TCP. In *5th PFLDnet Workshop* (2007).

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

[17] Barré, S., Ronan, J., and Bonaventure, O. Implementation and evaluation of the Shim6 protocol in the Linux kernel. *Computer Communications* (2011).

[18] Bennett, J. C. R., Partridge, C., and Shectman, N. Packet reordering is not pathological network behavior. *IEEE/ACM Transactions on Networking* (1999).

[19] Berners-Lee, T. Universal Resource Identifiers in WWW – A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web. IETF RFC 1630, June 1994.

[20] Berners-Lee, T., Fielding, R. T., and Nielsen, H. F. Hypertext Transfer Protocol – HTTP/1.0. IETF RFC 1945, May 1996.

[21] Best, M. L. Can the Internet be a Human Right? *Human Rights & Human Welfare 4* (2004), 23 – 31.

[22] Bhandarkar, S., and Reddy, A. L. N. TCP-DCR: Making TCP Robust to Non-Congestion Events. *NETWORKING 2004* (2004), 712–724.

[23] Bhandarkar, S., Reddy, A. L. N., Allman, M., and Blanton, E. Improving the Robustness of TCP to Non-Congestion Events. IETF RFC 4653, August 2006.

[24] Blanton, E., and Allman, M. On Making TCP More Robust to Packet Reordering. *ACM SIGCOMM Computer Communication Review* (2002).

[25] Blanton, E., Allman, M., Fall, K., and Wang, L. A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP. IETF RFC 3517, April 2003.

[26] Bohacek, S., Hespanha, J. P., Lee, J., Lim, C., and Obraczka, K. A new TCP for Persistent Packet Reordering. *IEEE/ACM Trans. Netw. 14* (April 2006), 369–382.

[27] Borgnat, P., Dewaele, G., Fukuda, K., Abry, P., and Cho, K. Seven Years and One Day: Sketching the Evolution of Internet Traffic. In *INFOCOM* (April 2009), IEEE.

[28] Bradner, S. Key words for use in RFCs to Indicate Requirement Levels. IETF RFC 2119, March 1997.

[29] Brakmo, L. S., O'Malley, S. W., and Peterson, L. L. TCP Vegas: New Techniques for Congestion Detection and Avoidance. *SIGCOMM Comput. Commun. Rev. 24* (October 1994), 24–35.

[30] Budzisz, L., Ferrus, R., Casadevall, F., and Amer, P. On Concurrent Multipath Transfer in SCTP-Based Handover Scenarios. In *IEEE International Conference on Communications (ICC)* (2009).

[31] Caini, C., and Firrincieli, R. TCP Hybla: a TCP Enhancement for Heterogeneous Networks. *International Journal of Satellite Communications and Networking 22* (2004).

[32] Cerf, V., Dalal, Y., and Sunshine, C. Specification of Internet Transmission Control Program. IETF RFC 675, December 1974.

[33] Cerf, V. G., and Kahn, R. E. A Protocol for Packet Network Intercommunication. *IEEE Transactions on Communications* (1974).

[34] Chebrolu, K., Raman, B., and Rao, R. R. A network layer approach to enable TCP over multiple interfaces. *Wireless Networks 11*, 5 (2005), 637–650.

[35] Chebrolu, K., and Rao, R. R. Bandwidth aggregation for real-time applications in heterogeneous wireless networks. *IEEE Transactions on Mobile Computing 5*, 4 (2006), 388–403.

[36] Chesterfield, J., Chakravorty, R., Pratt, I., Banerjee, S., and Rodriguez, P. Exploiting diversity to enhance multimedia streaming over cellular links. In *INFOCOM* (March 2005).

[37] Cisco Systems, I. Cisco visual networking index: Forecast and methodology, 2008–2013. Tech. rep., Cisco Systems, Inc., June 9 2009.

[38] Cohen, B. The BitTorrent Protocol Specification. http://bittorrent.org, February 2008.

[39] Crocker, S. D., Carr, C. S., and Cerf, V. G. New HOST-HOST Protocol. IETF RFC 33, February 1970.

[40] Davie, B., Charny, A., Bennett, J., Benson, K., Boudec, J.-Y. L., Courtney, B., Davari, S., Firoiu, V., and Stiliadis, D. An Expedited Forwarding PHB (Per-Hop Behavior). IETF RFC 3246, March 2002.

[41] Dell'Aera, A., Grieco, L. A., and Mascolo, S. Linux 2.4 Implementation of Westwood+ TCP with rate-halving: A Performance Evaluation over the Internet. In *IEEE International Conference on Communications (ICC)* (June 2004), vol. 4, pp. 2092 – 2096.

[42] Dreibholz, T., Becke, M., Rathgeb, E. P., and Tuxen, M. On the Use of Concurrent Multipath Transfer over Asymmetric Paths. In *Proceedings of the IEEE Global Communications Conference (GLOBECOM), ISBN 978-1-4244-5637-6* (December 2010).

[43] Evensen, K., and Kaspar, D. Bandwidth Aggregation over Heterogeneous Wireless Links. Poster at the Trilogy Summer School, Louvain-la-Neuve, Belgium, August 2009.

[44] Evensen, K., Kaspar, D., Engelstad, P., Hansen, A. F., Griwodz, C., and Halvorsen, P. A network-layer proxy for bandwidth aggregation and reduction of IP packet reordering. In *IEEE Conference on Local Computer Networks (LCN)* (2009).

[45] Evensen, K., Kupka, T., Kaspar, D., Halvorsen, P., and Griwodz, C. Quality-Adaptive Scheduling for Live Streaming over Multiple Access Networks. In *The 20th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)* (2010).

[46] Evensen, K., Petlund, A., Kaspar, D., Griwodz, C., and Halvorsen, P. Mobile Video Streaming Using Location-based Network Prediction and Transparent Handover. In *NOSSDAV* (2011), C. Krasic and K. Li, Eds., ACM, pp. 21 – 26.

[47] Evensen, K., Petlund, A., Riiser, H., Vigmostad, P., Kaspar, D., Griwodz, C., and Halvorsen, P. Demo: Quality-Adaptive Video Streaming With Dynamic Bandwidth Aggregation on Roaming, Multi-Homed Clients. In *International Conference on Mobile Systems, Applications, and Services (MOBISYS)* (2011), A. Agrawala, Ed., pp. 355–356.

[48] Evensen, K. R., Kaspar, D., Griwodz, C., Halvorsen, P., Hansen, A. F., and Engelstad, P. E. Improving the Performance of Quality-Adaptive Video Streaming over Multiple Heterogeneous Access Networks. In *Multimedia Systems (MMSys)* (2011).

[49] Fall, K., and Floyd, S. Simulation-based comparisons of Tahoe, Reno and SACK TCP. *SIGCOMM Computer Communications Review 26* (July 1996), 5–21.

[50] Farinacci, D., Fuller, V., Meyer, D., and Lewis, D. Locator/ID Separation Protocol (LISP). IETF Internet Draft (draft-ietf-lisp), expires: October 28, 2011, April 2011.

[51] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. Hypertext transfer protocol – HTTP/1.1. IETF, 1999.

[52] Fiore, M., Casetti, C., and Galante, G. Concurrent multipath communication for real-time traffic. *Comput. Commun. 30*, 17 (2007), 3307–3320.

[53] Floyd, S. HighSpeed TCP for Large Congestion Windows. IETF RFC 3649, December 2003.

[54] Floyd, S., Henderson, T., and Gurtov, A. The NewReno Modification to TCP's Fast Recovery Algorithm. IETF RFC 3782, April 2004.

[55] Floyd, S., Mahdavi, J., Mathis, M., and Podolsky, M. An Extension to the Selective Acknowledgement (SACK) Option for TCP. IETF RFC 2883, July 2000.

[56] Ford, A., Raiciu, C., Handley, M., Barre, S., and Iyengar, J. Architectural Guidelines for Multipath TCP Development. IETF RFC 6182, March 2011.

[57] Fracchia, R., Casetti, C., Chiasserini, C.-F., and Meo, M. Wise: Best-path selection in wireless multihoming environments. *IEEE Transactions on Mobile Computing 6*, 10 (2007), 1130–1141.

[58] Fu, C. P., and Liew, S. C. TCP Veno: TCP Enhancement for Transmission Over Wireless Access Networks. *IEEE Journal on Selected Areas in Communications 21*, 2 (February 2003), 216 – 228.

[59] Funasaka, J., Takemoto, Y., and Ishida, K. Parallel Downloading Method using HTTP over UDP for High Loss Rate and Delay Networks. In *Proceedings of the Eighth International Symposium on Autonomous Decentralized Systems* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 555–561.

[60] Gettys, J. Bufferbloat – Dark Buffers in the Internet. Presentation (http://www.bufferbloat.net), February 2011.

[61] Gharai, L., Perkins, C., and Lehman, T. Packet reordering, high speed networks and transport protocol performance. *ICCCN* (2004).

[62] Griwodz, C. *Wide-area True Video-on-Demand by a Decentralized Cache-based Distribution Infrastructure*. PhD thesis, Darmstadt University of Technology, 2000.

[63] Ha, S., Rhee, I., and Xu, L. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating System Review 42*, 5 (July 2008), 64–74.

[64] Habib, A., Christin, N., and Chuang, J. Taking advantage of multihoming with session layer striping. In *INFOCOM* (2006).

[65] Hedayat, A. S., Sloane, N. J. A., and Stufken, J. *Orthogonal Arrays: Theory and Applications*. Springer, 1999.

[66] Hemminger, S. Network Emulation with NetEm. In *linux.conf.au* (2005).

[67] Hemminger, S. TCP Testing (tcp_probe). The Linux Foundation – http://devresources.linuxfoundation.org/shemminger/tcp, Accessed 2011.

[68] Hoe, J. C. Start-up Dynamics of TCP's Congestion Control and Avoidance Schemes. Master's thesis, Massachusetts Institute of Technology (MIT), 1995.

[69] Hopps, C. Analysis of an equal-cost multi-path algorithm. IETF RFC 2992, November 2000.

[70] Hsieh, H.-Y., and Sivakumar, R. pTCP: An End-to-End Transport Layer Protocol for Striped Connections. In *IEEE International Conference on Network Protocols (ICNP)* (2002).

[71] Huang, C.-M., and Tsai, C.-H. WiMP-SCTP: Multi-path transmission using stream control transmission protocol (SCTP) in wireless networks. In *AINA Workshops (1)* (2007), pp. 209–214.

[72] IETF. MIF Status Pages. Online: http://tools.ietf.org/wg/mif.

[73] IETF. MPTCP Status Pages. Online: http://tools.ietf.org/wg/mptcp.

[74] ITU. Measuring the Information Society: The ICT Development Index. ISBN 9261128319, 2009.

[75] Iyengar, J., Shah, K., Amer, P., and Stewart, R. Concurrent multipath transfer using sctp multihoming. In *SPECTS* (2004).

[76] Iyengar, J. R., Amer, P. D., and Stewart, R. Concurrent Multipath Transfer using SCTP Multihoming over Independent End-to-End Paths. *IEEE/ACM Trans. Netw. 14*, 5 (2006).

[77] Jacobson, V. Congestion Avoidance and Control. In *SIGCOMM* (1988).

[78] Jacobson, V. Modified TCP Congestion Avoidance Algorithm. Online at ftp://ftp.isi.edu/end2end/end2end-interest-1990.mail, April 30 1990.

[79] Jacobson, V., Bradan, R., and Borman, D. TCP Extensions for High Performance. IETF RFC 1323, May 1992.

[80] Jaiswal, S., Iannaccone, G., Diot, C., Kurose, J., and Towsley, D. Measurement and Classification of Out-of-Sequence Packets in a Tier-1 IP Backbone. *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurements (IMW)* (2002).

[81] Jaiswal, S., Iannaccone, G., Diot, C., Kurose, J., and Towsley, D. Measurement and Classification of Out-of-Sequence Packets in a Tier-1 IP Backbone. In *INFOCOM* (2003).

[82] Jaiswal, S., Iannaccone, G., Diot, C., Kurose, J., and Towsley, D. Measurement and Classification of Out-of-Sequence Packets in a Tier-1 IP Backbone. *IEEE/ACM Transactions on Networking 15* (February 2007), 54–66.

[83] Järvinen, I., and Kojo, M. Improving Processing Performance of Linux TCP SACK Implementation. In *International Workshop on Protocols for Future, Large-Scale and Diverse Network Transports (PFLDNeT)* (May 2009).

[84] Jayasumana, A., Piratla, N., Banka, T., Bare, A., and Whitner, R. Improved packet reordering metrics. IETF RFC 5236, 2008.

[85] Johansen, D., Johansen, H., Aarflot, T., Hurley, J., Kvalnes, Å., Gurrin, C., Sav, S., Olstad, B., Aaberg, E., Endestad, T., Riiser, H., Griwodz, C., and Halvorsen, P. DAVVI: A Prototype for the Next Generation Multimedia Entertainment Platform (demo). In *ACM Multimedia* (2009).

[86] Kandula, S., Katabi, D., Sinha, S., and Berger, A. Dynamic Load Balancing without Packet Reordering. *ACM SIGCOMM Computer Communication Review 37*, 2 (April 2007), 51 – 62.

[87] Karn, P., and Partridge, C. Improving Round-Trip Time Estimates in Reliable Transport Protocols. *SIGCOMM* (1987).

[88] Kaspar, D., Evensen, K., Engelstad, P., and Hansen, A. F. Using HTTP Pipelining to Improve Progressive Download over Multiple Heterogeneous Interfaces. In *International Conference on Communications (ICC)* (2010).

[89] Kaspar, D., Evensen, K., Engelstad, P., Hansen, A. F., Griwodz, C., and Halvorsen, P. Data Segmentation, Request and Transfer Method. US Patent Application 12/713,939, February 2010.

[90] Kaspar, D., Evensen, K., Engelstad, P., Hansen, A. F., Halvorsen, P., and Griwodz, C. Enhancing Video-on-Demand Playout over Multiple Heterogeneous Access Networks. In *Consumer Communications and Networking Conference (CCNC)* (2010).

[91] Kaspar, D., Evensen, K., Hansen, A. F., Engelstad, P., Halvorsen, P., and Griwodz, C. An Analysis of the Heterogeneity and IP Packet Reordering over Multiple Wireless Networks. In *IEEE Symposium on Computers and Communications (ISCC)* (2009).

[92] Kaspar, D., Hansen, A. F., and Griwodz, C. Multilink Transfer over Heterogeneous Networks. In *IEEE International Conference on Network Protocols (ICNP), Student Poster Session* (2008).

[93] Kelly, T. Scalable TCP: Improving Performance in Highspeed Wide Area Networks. *ACM SIGCOMM Computer Communication Review 33* (2002), 83–91.

[94] Kerrisk, M. Linux Man Pages. http://www.kernel.org/doc/man-pages/.

[95] Keshav, S. The packet pair flow control protocol. Tech. Rep. 91-028, International Comp. Sci. Institute, Berkeley, CA 94704, May 1991.

[96] Kim, K.-H., and Shin, K. G. Improving TCP performance over wireless networks with collaborative multi-homed mobile hosts. In *MobiSys* (2005), ACM.

[97] Kim, K.-H., and Shin, K. G. PRISM: Improving the Performance of Inverse-Multiplexed TCP in Wireless Networks. *IEEE Transactions on Mobile Computing* (2007).

[98] Krasic, C., and Walpole, J. Priority-Progress Streaming for Quality-Adaptive Multimedia. In *Proceedings of the ninth ACM international conference on Multimedia* (New York, NY, USA, 2001), MULTIMEDIA '01, ACM, pp. 463–464.

[99] Kuzmanovic, A., and Knightly, E. W. TCP-LP: Low-Priority Service via End-Point Congestion Control. *IEEE/ACM Transactions on Networking 14*, 4 (August 2006), 739 – 752.

[100] Laywine, C. A., and Mullen, G. L. *Discrete Mathematics using Latin Squares.* John Wiley and Sons, 1998.

[101] Leith, D. J., and Shorten, R. N. H-TCP: TCP for High-Speed and Long-Distance Networks. In *Proc. PFLDnet* (2004).

[102] Leung, K.-C., Li, V., and Yang, D. An overview of packet reordering in transmission control protocol (tcp): Problems, solutions, and challenges. *Parallel and Distributed Systems, IEEE Transactions on 18*, 4 (April 2007), 522–535.

[103] Liao, J., Wang, J., and Zhu, X. cmpSCTP: An extension of SCTP to support concurrent multi-path transfer. *ICC* (2008).

[104] Liu, S., Başar, T., and Srikant, R. TCP-Illinois: A Loss and Delay-Based Congestion Control Algorithm for High-Speed Networks. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools* (New York, NY, USA, 2006), Valuetools '06, ACM.

[105] Loguinov, D., and Radha, H. End-to-End Internet Video Traffic Dynamics: Statistical Study and Analysis. pp. 723–732.

[106] Ludwig, R., and Gurtov, A. The Eifel Response Algorithm for TCP. IETF RFC 4015, February 2005.

[107] Ludwig, R., and Katz, R. H. The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions. *ACM SIGCOMM Computer Communication Review 30* (2000), 30 – 36.

[108] Ludwig, R., and Meier, M. The Eifel Detection Algorithm for TCP. IETF RFC 3522, April 2003.

[109] Ma, L., Yu, F., Leung, V., and Randhawa, T. A new method to support umts/wlan vertical handover using sctp. *IEEE Wireless Communications 11* (August 2004), 44 – 51.

[110] Mao, S., Bushmitch, D., Narayanan, S., and Panwar, S. Mrtp: A multi-flow real-time transport protocol for ad hoc networks. *IEEE Transactions on Multimedia* (2006).

[111] Martin, R., Menth, M., and Hemmkeppler, M. Accuracy and Dynamics of Multi-Stage Load Balancing for Multipath Internet Routing. In *IEEE International Conference on Communications (ICC)* (Glasgow, Scotland, June 2007).

[112] Mascolo, S., Casetti, C., Gerla, M., Sanadidi, M. Y., and Wang, R. TCP westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. In *Proceedings of the 7th annual international conference on Mobile computing and networking* (New York, NY, USA, 2001), MobiCom '01, ACM, pp. 287–297.

[113] Mathis, M., and Mahdavi, J. Forward Acknowledgment: Refining TCP Congestion Control. In *SIGCOMM* (1996).

[114] Mathis, M., Mahdavi, J., Floyd, S., and Romanow, A. Tcp selective acknowledgment options. IETF RFC 2018, October 1996.

[115] Mogul, J. C. Observing TCP Dynamics in Real Networks. In *In Proc. SIGCOMM Symposium on Communications Architectures and Protocols* (August 1992), p. 305.

[116] Moller, K.-M. Increasing Throughput in IP Networks with ESP-Based Forwarding: ESPBasedForwarding. IETF RFC 5984, April 1 2011.

[117] Mondal, A., and Kuzmanovic, A. Removing Exponential Backoff from TCP. *ACM SIGCOMM Computer Communication Review 38*, 5 (October 2008).

[118] Morton, A., Ciavattone, L., Ramachandran, G., Shalunov, S., and Perser, J. Packet reordering metrics. IETF RFC 4737, November 2006.

[119] Moskowitz, R., Nikander, P., Jokela, P., and Henderson, T. R. Host identity protocol. IETF RFC 5201, April 2008.

[120] Move Networks. Internet Television: Challenges and Opportunities. Tech. rep., Move Networks, Inc., November 2008.

[121] Networks, M. It's not web video, it's television: The power of internet tv. Tech. rep., Move Networks, Inc., September 2008.

[122] Ni, P., Eichhorn, A., Griwodz, C., and Halvorsen, P. Fine-grained Scalable Streaming from Coarse-grained Videos. In *Proc. ACM NOSSDAV* (2009), pp. 103–108.

[123] Nielsen, H. F., Gettys, J., Baird-Smith, A., Prud'hommeaux, E., Lie, H. W., and Lilley, C. Network Performance Effects of HTTP/1.1, CSS1, and PNG. *SIGCOMM Computer Communications Review 27* (October 1997), 155–166.

[124] NLANR/DAST. Iperf - the TCP/UDP bandwidth measurement tool. http://dast.nlanr.net/Projects/Iperf/, Accessed 2008.

[125] Nordmark, E., and Bagnulo, M. Shim6: Level 3 Multihoming Shim Protocol for IPv6. IETF RFC 5533, June 2009.

[126] NS-2. The network simulator ns-2. www.isi.edu/nsnam/ns.

[127] Paxson, V., and Allman, M. Computing TCP's Retransmission Timer. IETF RFC 2988, November 2000.

[128] Paxson, V. E. Measurements and Analysis of End-to-End Internet Dynamics. *ACM SIGCOMM Computer Communication Review* (1997).

[129] Perotto, F., Casetti, C., and Galante, G. SCTP-based Transport Protocols for Concurrent Multipath Transfer. *WCNC* (2007).

[130] Phatak, D. S., and Goff, T. A novel mechanism for data streaming across multiple ip links for improving throughput and reliability in mobile environments. In *INFOCOM* (2002).

[131] Phatak, D. S., Goff, T., and Plusquellic, J. IP-in-IP tunneling to enable the simultaneous use of multiple IP interfaces for network level connection striping. *Computer Networks* (2003).

[132] Piratla, N. M., and Jayasumana, A. P. Metrics for packet reordering - a comparative analysis. In *International Journal of Communication Systems* (2007).

[133] Polishchuk, T., and Gurtov, A. Secure multipath transport for legacy internet applications. In *Broadnets* (2009).

[134] Postel, J. User datagram protocol. IETF RFC 768, August 1980.

[135] Postel, J. NCP/TCP Transition Plan. IETF RFC 801, November 1981.

[136] Postel, J. Transmission control protocol. IETF RFC 793, September 1981.

[137] Postel, J., and Reynolds, J. File Transfer Protocol (FTP). IETF RFC 959, October 1985.

[138] Prasad, R. S., Murray, M., Dovrolis, C., and Claffy, K. Bandwidth Estimation: Metrics, Measurement Techniques, and Tools. *IEEE Network 17*, 6 (December 2003), 27 – 35.

[139] Qureshi, A., Carlisle, J., and Guttag, J. Tavarua: video streaming with wwan striping. In *MULTIMEDIA '06: Proceedings of the 14th annual ACM international conference on Multimedia* (New York, NY, USA, 2006), ACM, pp. 327–336.

[140] Raiciu, C., Barré, S., Pluntke, C., Greenhalgh, A., Wischik, D., and Handley, M. Improving Datacenter Performance and Robustness with Multipath TCP. In *SIGCOMM 2011, Toronto, Canada* (August 2011). See http://inl.info.ucl.ac.be/mptcp for related work on MPTCP and the Linux kernel implementation used in this paper.

[141] Raz, G. 'Lo' And Behold: A Communication Revolution. National Public Radio – All Things Considered, October 29 2009.

[142] Rodriguez, P., and Biersack, E. W. Dynamic Parallel Access to Replicated Content in the Internet. *IEEE/ACM Transactions on Networking* (2002).

[143] Sarolahti, P., Kojo, M., and Raatikainen, K. F-rto: an enhanced recovery algorithm for tcp retransmission timeouts. *SIGCOMM Computer Communications Review 33* (April 2003), 51–63.

[144] Sarolahti, P., Kojo, M., Yamamoto, K., and Hata, M. Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP. IETF RFC 5682, September 2009.

[145] Sarolahti, P., and Kuznetsov, A. Congestion Control in Linux TCP. In *In Proceedings of Usenix 2002, Monterey CA, USA* (June 2002).

[146] Schulzrinne, H., Casner, S. L., Frederick, R., and Jacobson, V. RTP: A Transport Protocol for Real-time Applications. IETF RFC 3550, July 2003.

[147] Shakkottai, S., Altman, E., and Kumar, A. The case for non-cooperative multihoming of users to access points in ieee 802.11 wlans. In *INFOCOM* (2006).

[148] Shiels, M. BBC interview with Martin Cooper. BBC News, April 2003.

[149] Simpson, W. A. The Point-to-Point Protocol (PPP). IETF RFC 1661, July 1994.

[150] Sinha, S., Kandula, S., and Katabi, D. Harnessing TCPs Burstiness using Flowlet Switching. In *3rd ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)* (San Diego, CA, November 2004).

[151] Sivakumar, H., Bailey, S., and Grossman, R. PSockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks. *Supercomputing, ACM/IEEE 2000 Conference* (2000).

[152] Sklower, K., Lloyd, B., McGregor, G., Carr, D., and Coradetti, T. The ppp multilink protocol (mp). IETF RFC 1990, August 1996.

[153] Snoeren, A. C. Adaptive Inverse Multiplexing for Wide-Area Wireless Networks. In *GLOBECOM* (1999).

[154] Stewart, R. Stream control transmission protocol. IETF RFC 4960, 2007.

[155] Stewart, R., Xie, Q., Tuexen, M., Maruyama, S., and Kozuka, M. Stream control transmission protocol (sctp) dynamic address reconfiguration. IETF RFC 5061, September 2007.

[156] Taguchi, G. *System of Experimental Design*. UNIPUB/Kraus International Publications, 1987.

[157] Thaler, D., and Hopps, C. Multipath issues in unicast and multicast next-hop selection. IETF RFC 2991, November 2000.

[158] Thompson, N., He, G., and Luo, H. Flow scheduling for end-host multihoming. In *INFOCOM* (April 2006).

[159] Tinta, S. P., Mohr, A. E., and Wong, J. L. Characterizing end-to-end packet reordering with UDP traffic. In *IEEE Symposium on Computers and Communications (ISCC)* (2009).

[160] Townsley, W. M., Valencia, A. J., Rubens, A., Pall, G. S., Zorn, G., and Palter, B. Layer Two Tunneling Protocol (L2TP). IETF RFC 2661, August 1999.

[161] Tsao, C.-L., and Sivakumar, R. On Effectively Exploiting Multiple Wireless Interfaces in Mobile Hosts. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies* (New York, NY, USA, 2009), CoNEXT '09, ACM, pp. 337–348.

[162] Walden, D. C., and McKenzie, A. A. The Evolution of Host-to-Host Protocol Technology. *Computer 12*, 9 (September 1979), 29 – 38.

[163] Wang, B., Wei, W., and Guo, Z. Multipath Live Streaming via TCP: Scheme, Performance and Benefits. In *ACM Transactions on Multimedia Computing, Communications and Applications* (2009).

[164] Wang, B., Wei, W., Guo, Z., and Towsley, D. Multipath live streaming via TCP: scheme, performance and benefits. In *ACM CoNEXT* (New York, NY, USA, 2007), ACM, pp. 1–12.

[165] Wang, Y., Lu, G., and Li, X. A Study of Internet Packet Reordering. *ICOIN 2004, LNCS 3090* (2004), 350–359.

[166] Wikipedia. Comparison of download managers, June 2009.

[167] Xu, L., Harfoush, K., and Rhee, I. Binary Increase Congestion Control for Fast, Long Distance Networks. In *INFOCOM* (2004).

[168] Yang, P., Luo, W., Xu, L., Deogun, J., and Lu, Y. TCP Congestion Avoidance Algorithm Identification. In *in Proceedings of IEEE ICDCS* (June 2011).

[169] Ye, B., Jayasumana, A. P., and Piratla, N. M. On monitoring of end-to-end packet reordering over the Internet. In *ICNS* (Washington, DC, USA, 2006), IEEE Computer Society.

[170] Zakon, R. H. Hobbes' Internet Timeline. IETF RFC 2235, November 1997.

[171] Zambelli, A. IIS Smooth Streaming Technical Overview. Tech. rep., Microsoft Corporation, March 2009.

[172] Zhang, L., Shenker, S., and Clark, D. D. Observations on the Dynamics of a Congestion Control Algorithm: the Effects of Two-Way Traffic. *SIGCOMM Comput. Commun. Rev. 21* (August 1991), 133–147.

[173] Zhang, M., Lai, J., Krishnamurthy, A., Peterson, L., and Wang, R. A transport layer approach for improving end-to-end performance and robustness using redundant paths. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference* (2004).

[174] Zhang, Q., Guo, C., Guo, Z., and Zhu, W. Efficient mobility management for vertical handoff between wwan and wlan. *IEEE Communications Magazine 41*, 11 (November 2003), 102 – 108.

[175] Zhou, X., and Van Mieghem, P. Reordering of IP packets in Internet. In *Passive and Active Measurements (PAM)* (2004).