Aggregating the Bandwidth of Multiple Network Interfaces to Increase the Performance of Networked Applications

by

Kristian R. Evensen

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

Devices capable of connecting to two or more different networks simultaneously, known as host multihoming, are becoming increasingly common. For example, most laptops are equipped with a least a Local Area Network (LAN) and a Wireless LAN (WLAN) interface, and smartphones can connect to both WLANs and 3G-networks (High-Speed Downlink Packet Access, HSDPA). Being connected to multiple networks simultaneously allows for desirable features like bandwidth aggregation and redundancy.

Enabling and making efficient use of multiple network interfaces or links (network interface and link will be used interchangeably throughout this thesis) requires solving several challenges related to deployment, link heterogeneity and dynamic behavior. Even though multihoming has existed for a long time, for example routers must support connecting to different networks, most existing operating systems, network protocols and applications do not take host multihoming into consideration. The default behavior is still to use a single interface for all traffic. Using a single interface is, for example, often insufficient to meet the requirements of popular, bandwidth intensive services like video streaming.

In this thesis, we have focused on bandwidth aggregation on host multihomed devices. Even though bandwidth aggregation has been a research field for several years, the related works have failed to consider the challenges present in real world networks properly, or does not apply to scenarios where a device is connected to different heterogeneous networks.

In order to solve the deployment challenges and enable the use of multiple links in a way that works in a real-world network environment, we have created a platform-independent framework, called MULTI. MULTI was used as the foundation for designing transparent (to the applications) and application-specific bandwidth aggregation techniques. MULTI works in the presence of Network Address Translation (NAT), automatically detects and configures the device based on changes in link state, and notifies the application(s) of any changes.

The application-specific bandwidth aggregation technique presented in this thesis was optimised for and evaluated with quailty-adaptive video streaming. The technique was evaluated with different types of streaming in both a controlled network environment and

real-world networks. Adding a second link gave a significant increase in both video and playback quality. However, the technique is not limited to video streaming and can be used to improve the performance of several, common application types.

In many cases, it is not possible to extend applications directly with multilink support. Working on the network-layer allows for the creation of bandwidth aggregation techniques that are transparent to applications. Transparent, network-layer bandwidth aggregation techniques must support the behavior of the different transport protocol in order to achieve efficient bandwidth aggregation. The transparent bandwidth aggregation techniques introduced in this thesis are targeted at Universal Datagram Protocol (UDP) and Transmission Control Protocol (TCP), the two most common transport protocols in the Internet today.

Acknowledgements

Working on the PhD has been a sometimes frustrating, but most of all a fun, rewarding and fulfilling experience. I would like to thank my supervisors, Dr. Audun F. Hansen, Prof. Paal E. Engelstad, Prof. Carsten Griwdoz and Prof. Pål Halvorsen for interesting discussions, encouragement, ideas, for learning me to work more independently and for putting up with me during my most stubborn times. I would also like to thank my colleague Dominik Kaspar for a fruitful collaboration on the topic of multilink.

Simula Research Laboratory has provided an excellent working environment with great colleagues, several whom have become good friends. I would especially like to thank Paul Beskow, Håvard Espeland, Håkon Stensland, Andreas Petlund, Ragnhild Eg, Pengpeng Ni, Tomas Kupka, Saif Shams, Molly Maleckar, Vo Quoc Hung and Ahmed Elmokashfi for a great time and lots of fun.

Thanks to my girlfriend, Aiko, for keeping me sane during the last months of thesis writing, giving me other things to think of and focus on, and making sure that I got fresh air. Finally, I would like to thank my friends and family for supporting me in whatever I choose to do, and my parents for providing cheap accommodation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Video streaming, cloud storage and other bandwidth intensive applications are among the most popular services on the Internet today. As the total available network capacity increases, so do the consumption rate of such services and the user's expectations. At the same time, most networked devices come equipped with multiple network interfaces. For example, smart-phones can typically connect to both WLAN and 3G-networks (HSDPA), while laptops come equipped with at least a LAN and a WLAN-interface. Due to the increased development of different types of wireless networks, devices are often within coverage range of multiple networks simultaneously.

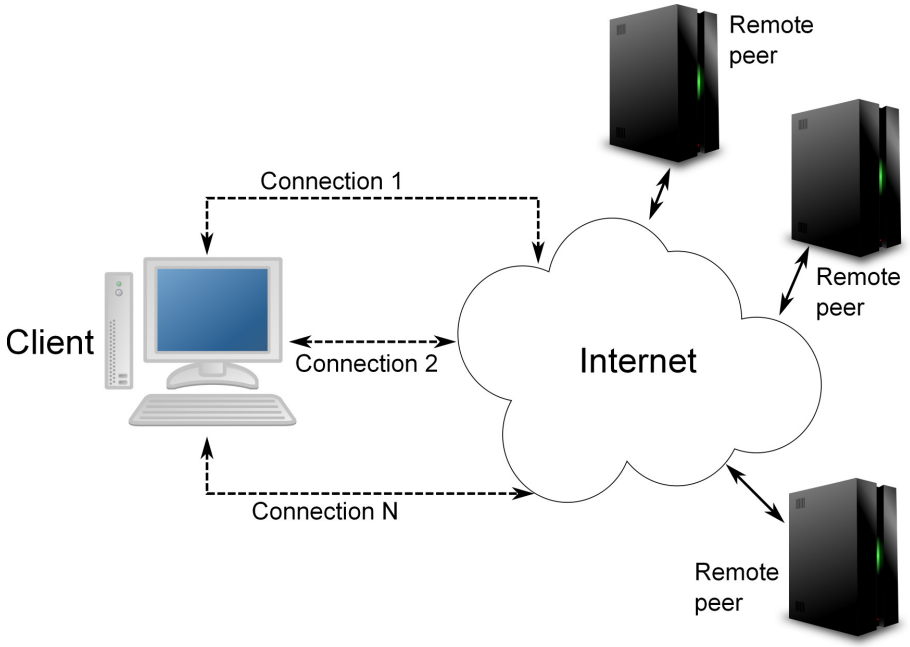


Figure 1.1: The default host multihoming scenario. A client device is equipped with multiple network interfaces and has several active network connections.

Devices that have multiple active network connections and have acquired a unique network identifier (for example, an IP-address) for more than one interface, are known as multihomed. There exists two types of multihoming, site and host multihoming. Site multihoming is used to describe multihoming in access and core networks. An example of a site multihomed device is a router. Host multihoming, on the other hand, is used to refer to client devices that can connect to multiple networks simultaneously. The default host multihoming scenario is shown in figure 1.1. A client is equipped with different network interfaces, has several active network connections to the Internet and wants to request a resource from a remote peer. Host multihoming enables applications to provide two desirable features, i.e., sequential access over different links and simultaneous use of multiple links. Sequential access can be used to for example add support for connection handover, by rerouting traffic to another interface if the current loses its connection to the network. Simultaneous use of multiple links is required to support bandwidth aggregation, which is the focus of this thesis.

For applications to communicate through a computer network, well-defined protocols are used. However, most standardized protocols only make use of a single link at a time. Even though the overall network capacity increases, clients will frequently be connected to networks that are unable to meet the requirements imposed by the services or the users expectations. For example, smooth streaming of high quality video frequently requires more bandwidth than what is often available in public WLANs, and having to wait a long time to receive a remotely stored file will lead to annoyed users. In our work, we have designed, implemented and evaluated different techniques for efficiently aggregating the bandwidth of multiple links. When our techniques are used, the logical bandwidth aggregation-enabled link provides higher bandwidth and a higher in-order throughput (throughout this thesis, throughput implies in-order throughput also known as goodput) than any of the single links. This allows, for example, increased quality in video streaming systems.

1.1 Background and motivation

The dominating design principle in the Internet today, is the end-to-end principle [67]. It states that the core Internet shall be as simple as possible, while all advanced logic is placed in the endpoints. This principle is followed by the Internet Protocol Suite [10, 11], commonly referred to as TCP/IP, a less rigidly designed model than the standardized OSI-model [80]. TCP/IP consists of a set of protocols deciding how computers communicate through networks. The suite is divided into four abstraction layers, i.e., the link layer, the network layer, the transport layer and the application layer, as summarized in

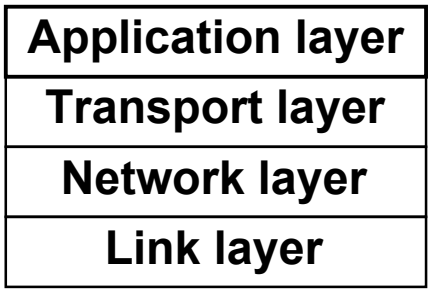


Figure 1.2: The four layers of the Internet Protocol stack, or the TCP/IP model.

figure 1.2. A layer can only communicate with the ones directly above and beneath it. At the bottom, the link layer is responsible for communicating with the actual, physical network. Then, the network layer (or Internet layer) transports packets between hosts across network boundaries (routing). Furthermore, the transport layer provides end-to-end communication, while the application layer enables processes to communicate. While the three others typically belong to the operating system (OS) kernel, application layer protocols are defined and implemented by the application developers. This allows for a great deal of flexibility, as the developer has full control over the behavior of the protocol.

Even though multihoming has existed for many years, it has been in the context of site multihoming. Site multihoming is described already in [10] and is used in access and core networks. Routers have to be connected to several networks in order to move packets between end-hosts belonging to different networks. However, as wireless technologies like WiFi and 3G have become sufficiently cheap, the number of clients that are able to connect to multiple networks simultaneously (known as host multihoming [1]) has increased rapidly. For example, almost every smartphone in sale today can connect to WiFi and 3G. However, most existing operating systems, network protocols and applications do not take multihoming properly into consideration. For example, Linux does not configure its routing tables correctly when more than one network interface is active, while a Transmissions Control Protocol-connection (TCP) [5, 61] is by design bound to a single interface. TCP is the most commonly used transport-layer protocol today.

As the overall capacity of the Internet has increased, along with the bandwidth of available consumer Internet connections, so has the consumption of high-bandwidth services. As of 2011, two billion videos are streamed from the video service YouTube every day, and over 24 hours worth of content is uploaded every minute [2]. Moreover, larger and larger files are stored in the cloud and downloaded from the web or through peer-to-peer networks. At the same time, client devices equipped with multiple network interfaces have become the norm. Today, smartphones can offload data traffic from 3G to WLAN (to pro-

[1] In the rest of the thesis, we refer to host multihoming when we say multihoming.
[2] http://www.youtube.com/t/fact_sheet

vide higher bandwidth and reduce the load on the phone network), while most computers are equipped with at least LAN- and WLAN-cards. Due to the increased popularity and expansion of different wireless networks, clients are often within coverage range of multiple networks simultaneously. For example, bigger cities, at least in most well-developed countries, have close to 100% 3G coverage. In addition, telecommunication companies, private companies and individuals offer access to WiFi-hotspots. One example of such a company is Telia, which gives their cell phone subscribers access to hotspots in several cities around the world [3].

However, even though multiple networks are available, the default behavior is still to use a single link for all network communication. The OS regards one of the links as the default link, and the default link is only updated when the current becomes unavailable. In many cases, using a single link is insufficient to meet the requirements imposed by a service, or a user's expectations. For example, a public WiFi-network might not be able to stream a video without causing playback interruptions due to buffer underruns, and the download time when receiving a large file over a 3G-connection might not be acceptable. The problem can be alleviated by aggregating the bandwidth of the different links, giving applications access to more bandwidth than a single link can provide. Multiple links can also be used to provide different services or features, for example, increased reliability of a networked application by using the additional link(s) for redundancy.

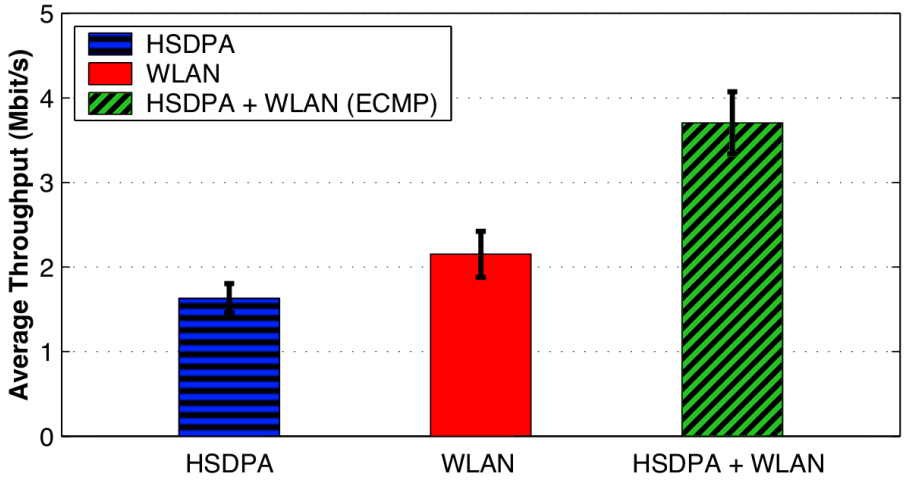


Figure 1.3: Throughput aggregation with BitTorrent over simultaneous HSDPA and WLAN links, using Equal-Cost Multipath Routing. The results were obtained by downloading a total of 75 copies of a 700 MB large file.

An example of the potential of bandwidth aggregation is shown in figure 1.3. Here, the client was connected to one HSDPA-network and one WLAN, and the popular peer-to-

[3] http://www.homerun.telia.com/eng/start/default.asp

peer protocol BitTorrent [4] was used to download a 700 MB large file. In BitTorrent, files are divided into smaller pieces. A client (peer) is connected directly to several other peers, and pieces are requested from different peers. To use both interfaces simultaneously, we enabled Equal-Cost Multipath Routing [34] (ECMP). ECMP enables system administrators to allocate weights to different routes and thereby distribute the traffic. We gave each link the same weight, and the connections to the other BitTorrent-peers were distributed across the two links using round-robin. As can be seen in the graph, when the WLAN- and HSDPA-connections were used together, the average achieved throughput was close to the sum of the throughputs when the two links were used alone.

What makes BitTorrent ideal for showing the potential of bandwidth aggregation, is that it relies on opening several connections. These connections can be distributed among the available interfaces, ideally ensuring full utilization of every link. This behavior is not common, most networked applications use a single connection for receiving all data related to one item, for example a file. Adding support for bandwidth aggregation either requires changing the application, or developing a transparent bandwidth aggregation solution. Transparent bandwidth aggregation solutions operate on the network layer, and can be designed in such a way that no changes to either application, operating system or network protocols is needed.

Bandwidth aggregation has been a research field for many years, as will be discussed in chapter 2. However, the related work we have found (some developed in parallel with our techniques) is mostly either based on 1) unrealistic or incorrect assumptions or requirements [12], 2) simulations [3, 12, 38], 3) fail to consider the different challenges present in real-world networks [2, 59, 70, 71] or 4) cannot be applied to a scenario where the devices are connected to different networks [2, 6, 71]. Performing efficient bandwidth aggregation requires addressing challenges related to connectivity, link heterogeneity and link reliability. A presentation of the different challenges and their effects is given in section 1.2.

For this thesis, we have designed, implemented and evaluated techniques to perform bandwidth aggregation at both the application and the network layer, and these techniques were experimentally evaluated in both fully controlled and real-world network environments. The proposed application layer technique is optimized for improving the performance of quality-adaptive video streaming. However, the technique is not limited to video streaming. As long as a common requirement is met (the client must be able to simultaneously request different parts of a file over different links), it can also be applied to for example bulk data transfer.

Operating on the network layer allows for the development of transparent multilink

[4] http://www.bittorrent.org/

techniques, i.e., no changes have to be made to the applications running on top, the network protocols or the operating system. In many cases, changes to the application code would not be desirable or even possible. For example, several applications are proprietary and only the original developers have access to the source code, while protocol changes have to be implemented at all machines that will communicate. As many protocols behave differently and require different techniques in order to achieve efficient transparent bandwidth aggregation, we have limited this thesis to TCP [61] and UDP [60]. These are the two most common transport layer protocols, and are used by almost every mainstream application communicating over the Internet today.

1.2 Bandwidth aggregation related challenges

Multihoming is supported by all major operating systems - Linux, Windows and BSD/OS X all support multiple active network interfaces simultaneously. However, when the different network protocols was designed, client devices were only equipped with a single interface, and multihoming has not been considered properly. For example, TCP-connections are bound to one interface, and operating systems, by default, regard one link as the default link and uses it for all traffic.

In order to enable efficient bandwidth aggregation, different challenges have to be overcome. Some are introduced by the operating system or network design, while others are a consequence of combining different networks or network technologies. For example, most Internet Service Providers (ISP) use NAT [17] to manage their networks, which makes clients unreachable from the outside. Link heterogeneity, on the other hand, have to be taken into consideration in order to utilize the full capacity of the links.

We have identified the key challenges relevant for our targeted scenarios and divided them into three main groups - deployment, link heterogeneity and unstable link performance. The first group contains challenges involving deployment, i.e., how to enable multiple links and build multilink applications/solutions that will also work in real-world networks. The second group consists of challenges related to the performance characteristics of different network technologies. Unstable link performance is especially a challenge when wireless links are used, different phenomenas and events (like rush hour or physical objects blocking the signal) affect the available bandwidth.

There are several other types of challenges related to multihoming and multilink usage. However, we consider them to be outside the scope of this thesis. For example, using multiple links will increase battery consumption, which is critical on mobile devices, and we have not looked into the financial side of multilink. In order for, amongst others, companies to develop and encourage the use of a multilink service, they need a sound

business model.

1.2.1 Deployment challenges

Even though all major operating systems support multihoming, the behavior when more than one network interface is connected differs. This is caused by differences in the routing subsystem of each operating system. Before a packet is sent from a machine, lookups are performed to find its correct route. On OS X and Windows (from Vista and onwards), the routing subsystem is configured correctly by default, and the operating system is able to send the packets in the presence of multiple active links. On Linux, on the other hand, the routing subsystem will in many cases be unable to make a routing decision and drop the packet. The first deployment challenge is to ensure that the operating system is properly configured, and that the correct routing decisions will be made.

Properly configured routing tables are sufficient for enabling the use of multiple links when either a connection-oriented transport protocol (for example TCP) is used, or when a connectionless protocol (for example UDP) is combined with machines placed inside the same network. Network sockets can be bound by the applications to the different interfaces and, thus, traffic will pass through the chosen networks. If all interfaces are within the same network, the machines can communicate directly and the connectionless datagrams can flow in both directions.

In the real-world, however, the different interfaces rarely belong to the same subnet/network. For example, a web server is likely on a different network than the WLAN and 3G networks a client is connected to, and the networks are in most cases separated by NAT. NAT is summarized in figure 1.4 and is used to reduce the number of global IP addresses. NAT-boxes are given a public IP address and is placed on the border between a local network and the Internet. Private IP-addresses are assigned to local clients, and when a client connects to a machine on another network, the NAT creates a mapping between the private IP and the destination IP (often by allocating a unique network port number). Then, the NAT rewrites the packet headers, for example the source IP address is set to that of the NAT. When packets arrive from the destination IP, the NAT looks up the mapping and rewrites the packet headers again.

In the scenario presented in figure 1.4, the NAT has been assigned the global IP 128.39.36.93 and the local network consists of two machines. Both machines connect to port 80 of the server with IP 74.125.77.99, and the NAT has created a mapping for each connection. A port number is used to identify each connection, 6666 and 6667. The packet headers are rewritten before packets are sent to the server, and any reply is sent to the NAT's IP using the port number that identifies the correct mapping. The NAT then rewrites the headers again so that they contain the address for the local machine.

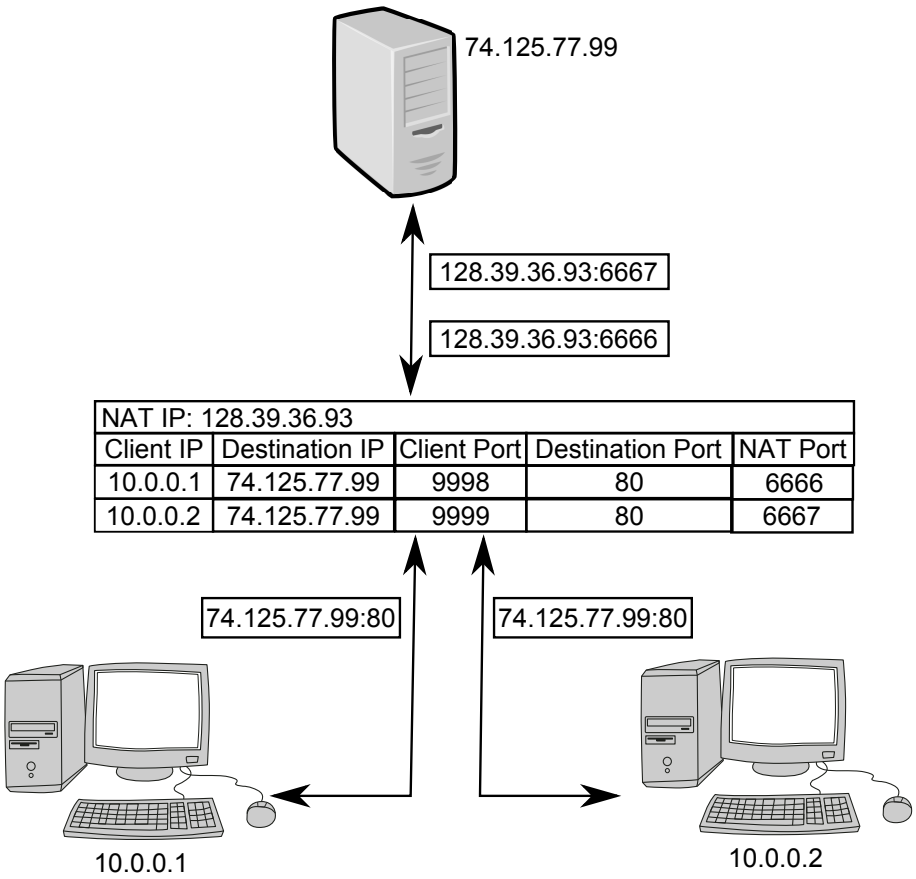


Figure 1.4: An example of NAT. Two clients that are place behind the same NAT-box communicates with the same server. When the server sends a packet, it uses the NAT's IP and the port assigned to the mapping.

Without knowledge about the NAT's IP and the mapping, a client is unreachable from the outside. The client's private IP address, which is the only one it is aware of by default, is invalid in other networks than its own. Techniques for working around the limited connectivity caused by NAT exists and is presented in chapter 3, along with our technique for supporting dynamic configuration of the network subsystem.

1.2.2 Link heterogeneity

Different network and network technologies often have significantly different performance characteristics. For example, the total bandwidth of a WLAN is usually several tens of megabits (the common 802.11g can support a theoretical maximum of 54 Mbit/s and 802.11n 600 Mbit/s [55]), while most HSDPA-networks support a theoretical maximum of 14 Mbit/s. Similarly, the latency of HSDPA is most of the time at least one order of magnitude higher than that of WLAN.

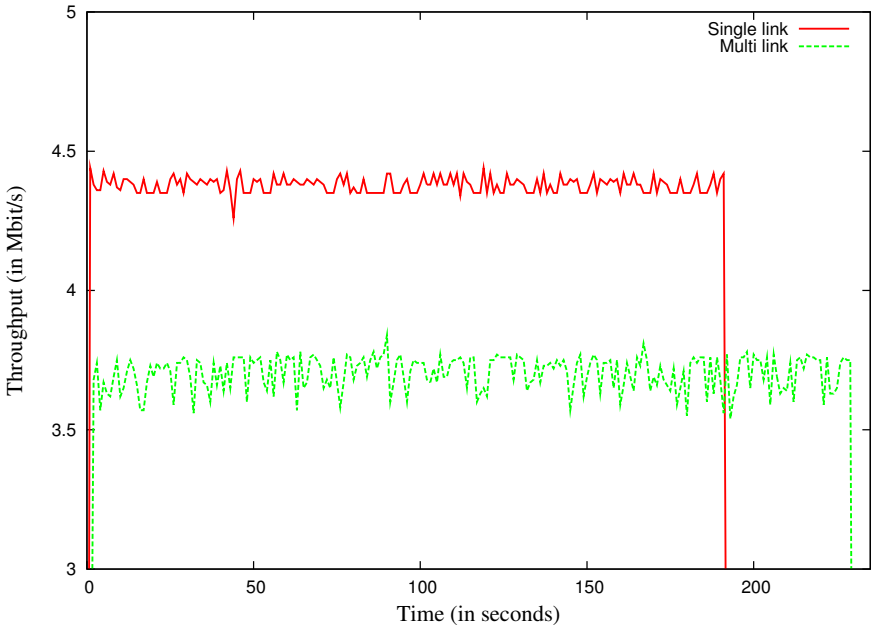


Figure 1.5: Achieved TCP-throughput when doing pure round-robin striping over links with heterogeneous RTT (10 ms and 100 ms, throughput averaged for each second).

Latency heterogeneity

Latency heterogeneity causes packet reordering and imposes a significant challenge when doing bandwidth aggregation, especially with reliable transport protocols. They deliver data to the applications in-order, and any out-of-order data will cause delays in delivery and a drop in performance. Also, TCP, among others, interprets packet reordering as loss. A TCP sender relies on feedback (acknowledgements, ACKs) from the receiver in order to send new data. If reordering occurs, TCP sends a duplicate acknowledgement (dupACK) of the previous in-order packet it has received. Unlike with normal ACKs, the receiver interprets a dupACK as if a packet has been lost, and TCP assumes that packet loss is caused by link congestion. By default, TCP invokes congestion control after receiving three of the same dupACK (known as Fast Retransmit [5]), reducing the sender's allowed sending rate.

The achieved aggregated throughput is dependent on the latency heterogeneity. As the heterogeneity increases, the throughput decreases due to the reordering. In order to illustrate the effect that latency heterogeneity can have on a TCP connection, we created a testbed consisting of two machines running Linux. The machines were connected directly to each other using two 100 Mbit/s Ethernet cards, and the network emulator *netem* [5] was used to add 10 ms round-trip time (RTT, the time it takes for a packet to travel to and from a receiver [51]) to one link, and 100 ms RTT to the second link. The bandwidth

[5]http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

of each link was limited to 5 Mbit/s in order to avoid bandwidth heterogeneity having an effect. A 100 Mb large file was downloaded over HTTP, and the achieved throughput is shown in figure 1.5, using pure round-robin to stripe the packets over the two links. As can be seen, the aggregated throughput was worse than a 5 Mbit/s links alone. Before the in-order packet(s) sent over the high RTT-link arrived and the proper ACK was sent from the receiver, the sender had often received enough dupACKs for a Fast Retransmit.

How UDP reacts to latency heterogeneity depends on the application. UDP is a non-reliable, best-effort transport protocol that will try to send all the traffic generated by the application. Unless the receiver is programmed to send feedback to the sender, the sender will never reduce its send rate, and the performance depends on the in-order requirement of the receiver application.

Application layer bandwidth techniques usually rely on opening multiple connections, and then requesting/receiving data over these connections. Each independent connection will not experience any reordering caused by the latency heterogeneity. However, the heterogeneity can affect the performance of the application. A significant latency heterogeneity causes gaps in the received data, which is critical as most applications process data sequentially. For example, a video streaming application will not be able to resume playback before the gap is filled, and gaps can lead to higher memory (buffer) requirements as applications need a temporary storage for out-of-order data.

Bandwidth heterogeneity

Bandwidth heterogeneity has to be taken into consideration in order to utilize the links efficiently. Otherwise, the slow(er) link might be allocated too much data and reduce the network performance of the application/bandwidth aggregation technique. For example, with TCP and pure round-robin striping, the aggregated bandwidth is limited by the bandwidth of the slowest link. As mentioned earlier, TCP invokes congestion control when it detects packet loss, and this will happen as soon as the congestion window has grown to N times what the slowest link can support (where N is the number of links). When the congestion window reaches this size, the slow link will be saturated and starts loosing packets. Thus, the full capacity of the other links will never be used. We have illustrated this in figure 1.6. The same testbed was used as in the latency-heterogeneity example, except that we limited the bandwidth instead of adding latency. Using the *hierarchical token bucket* [6], the bandwidth of one link was limited to 5 Mbit/s (measured to $\sim$ 4 Mbit/s), while the other was limited to 10 Mbit/s (measured to $\sim$ 9 Mbit/s). With pure round-robin striping, the achieved aggregated throughput was close to 9 Mbit/s, or almost twice that of the slowest link. However, when striping the packets according to the

[6] http://luxik.cdi.cz/~devik/qos/htb/

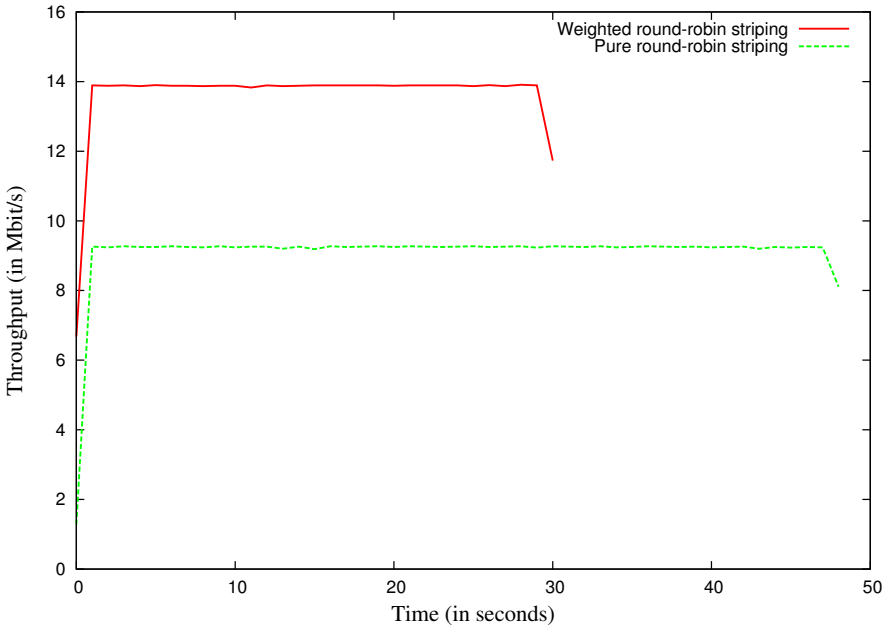


Figure 1.6: Achieved TCP-throughput with pure round-robin and weighted round-robin striping over links with heterogeneous bandwidth (5 Mbit/s and 10 Mbit/s, throughput averaged for each second).

bandwidth ratio (1:2), using weighted round robin, an aggregated throughput of close to 14 Mbit/s was achieved. The reason that the full 15 Mbit/s was not reached, was TCP's congestion control.

As with the latency heterogeneity, the behavior of UDP when faced with bandwidth heterogeneity depends on the applications. Because UDP has no congestion control, a UDP sender will never back off and might generate a high-bandwidth stream that will saturate every link. However, packets will be lost over the links that are unable to support the bandwidth requirement for their share of the stream. The effect of bandwidth heterogeneity on an application layer bandwidth aggregation technique resembles that of latency heterogeneity. If too much data is allocated to a slower link, more and larger gaps will occur in the received data.

In summary, not properly considering bandwidth heterogeneity limits the effectiveness of bandwidth aggregation. The links will not be fully utilised and, thus, the performance will suffer.

1.2.3 Unstable link performance

In addition to bandwidth and latency heterogeneity, wireless network technologies tend to deliver unstable throughput and latency. How much capacity a client is allocated or able to use is decided by several factors including numbers of users sharing a wireless channel,

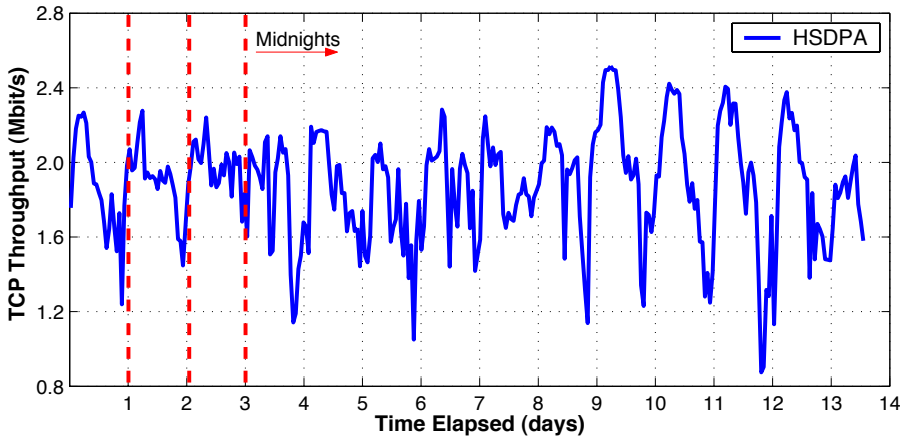


Figure 1.7: HSDPA throughput varying over a period of 14 days. Results obtained by repeatedly downloading a 5 MB large file over HTTP.

fading, interference and radio conditions. This must also be taken into consideration, for example through dynamic adaptation, when developing multilink applications for use in or with wireless networks. An example of the fluctuating performance can be seen in figure 1.7. A 5 MB large file was downloaded repeatedly over a period of 14 days using a public HSDPA network, and a significant variance in the throughput can be observed.

Not considering unstable link performance when designing a bandwidth aggregation solution, will lead to a combination of the drawbacks discussed for bandwidth and latency heterogeneity. The solution will not properly consider the capacity and characteristics of the links and, thus, the performance will suffer.

1.3 Problem statement

Bandwidth aggregation is often a desirable property for a client device being connected to multiple networks simultaneously, at least from a user's perspective. However, performing efficient bandwidth aggregation requires addressing several challenges, as described in the previous section. One has to consider connectivity issues, link heterogeneity and link stability. The main goal of this thesis has been to design, develop and evaluate bandwidth aggregation techniques that address these challenges, and improve the performance of different bandwidth intensive applications when run on multihomed devices. We divided the main goal into the following subgoals:

- Design, develop, optimise and evaluate a platform-independent technique for solving the deployment challenges, in order to ease the design, development and deployment of multilink solutions.

- Design, develop, optimise, and evaluate a technique for aggregating bandwidth at

the application layer, optimised for on common type of bandwidth-intensive application - quality-adaptive video streaming.

- Design, develop, optimise and evaluate techniques for transparently aggregating UDP and TCP-streams, the two most common transport protocols. Neither the protocol nor protocol behavior can be changed.
- A technique should not require changes to the existing protocols, protocol behavior or the operating systems.
- Every solutions must work in real-world networks.

1.4 Limitations

Due to time constraints, we have had to limit the scope of this thesis. The following areas have not been investigated:

- **IPv6:** IPv6 would remove some of the deployment challenges. For example, due to the large increase in number of available IP-addresses, NAT will probably no longer be needed. However, even though our techniques have been designed for and evaluated with IPv4, they do not rely on it and should work with any network layer addressing protocol.
- **Other transport protocols than UDP and TCP:** A transparent bandwidth aggregation technique has to support the behavior of the targeted transport protocol. Because it is not feasible to design a technique for every existing transport protocol, we have focused on the two most common, TCP and UDP.

1.5 Scientific context and Methodology

Science is derived from the Latin word *scientia* and means knowledge. A more precise definition is given in [54]: *Science is knowledge or a system of knowledge covering general truths or the operation of general laws especially as obtained and tested through scientific method.* Computer science is a subset of science and was introduced in the 1940's, with the first computer science department formed at Purdue University in 1962.

Computer science encompasses several different fields, however, they are all related to the evolution of computers and how computers have become a part of every day life. Fields include system design, studying the properties of complex computational problems and computer architecture and engineering. According to [16], computer science can be divided into three paradigms:

- The **rationalist paradigm** defines computer science as a branch of mathematics. Programs are treated as mathematical objects, and deductive reasoning is used to evaluate their correctness based on a priori knowledge.
- The **technocratic paradigm** defines computer science as an engineering discipline. Programs are treated as data and the knowledge is collected a posteriori. I.e., programs are evaluated using testing suites, and the results/experience are considered as the knowledge.
- The **scientific paradigm** defines computer science as natural (empirical) science. Programs are entities on par with mental processes, and a priori and a posterior knowledge is gathered using a combination of formal deduction and scientific experimentation.

Similar paradigms are introduced by the Association for Computing Machinery (ACM) in [14]. They describe three main paradigms: abstraction, design and theory. The abstraction paradigm seeks knowledge through validating models of given systems, while the design paradigm seeks knowledge through building systems and then validating them through testing. Finally, the theory paradigm is rooted in mathematics and knowledge is gathered by giving formal proof of the properties of a system.

All the different paradigms can be applied to the field of computer networks and communications. For example, the rationalist paradigm is needed when the goal is to prove certain properties, for example the effect of latency heterogeneity on TCP, while the technocratic paradigm can be used when the research is targeted at improving the performance of specific application types in real-world networks.

In terms of the paradigms, we make use of the technocratic paradigm and ACM's design paradigm. The work presented in this thesis was motivated by the potential of bandwidth aggregation and the increasing number of multihomed clients. In order to get realistic results, as well as the fact that most related work has only been implemented and evaluated in simulators, the techniques were evaluated by building systems that make use of them. To get the most realistic behavior and test conditions, the systems were inserted into real computer networks.

According to [41], there are three main techniques for evaluating the performance of a system. *Analytical modeling* uses mathematics and formulas to describe the behavior of a system, *Simulation* involves implementing a model of a system and then evaluates it using different workloads in a deterministic state-machine. *Measurements* can be used when the system can be implemented and evaluated in the real world.

Doing measurements provides the most valid results for the techniques presented in this thesis. The simplifications caused by modelling or simulating the behavior of different

types of networks and protocols, can have a significant impact on the results. In addition, link characteristics like fluctuating bandwidth (present in for example wireless networks) are difficult to model. Finally, our goal was to develop techniques that would work in real-world networks and without changing existing infrastructure. This claim needs to be verified.

In order to do measurements and get reproducible results, as well as evaluate the effect of different levels of different parameters (bandwidth and latency), each bandwidth aggregation technique was evaluated in two testbeds. The first testbed was a controlled network environment where a network emulator was used to limit the bandwidth and control link latency. This allowed us to emulate different types of networks. The second testbed consisted of a client connected to multiple, real-world wireless network. This testbed was used to provide results that gave an impression of the performance if a technique is deployed.

Using emulators and real-world machines affects the validity of results, as the results are affected by both hardware and software. For example, there exist different implementations of TCP, drivers for wireless cards might behave differently and applications/OSes might contain bugs. Ideally, one should test every possible combination of software and hardware, and fix every possible bug, but this is not feasible. In order to reduce the probability of our results being affected by implementation differences, the same machines were used for every experiment, with the same hardware and OS configuration. Also, all the techniques presented in this thesis are based on standards and the core concepts of the different protocols. The same applies to the implementations used for the evaluations. No operating system specific optimizations have been made or features used, and no assumptions have been made about the behavior of the underlaying operating system. In other words, the techniques presented in this thesis are generic. The only exception is our multilink framework, MULTI, which relies on operating system specific behavior. However, the features required by MULTI are supported by all major operating systems.

There is related work in the field of bandwidth aggregation. However, comparing the performance of different techniques and solutions directly is difficult. Different metrics and scenarios have been used, and we did not find any working open-source implementations of related work. However, there is some common ground. For transparent bandwidth aggregation solutions, the aggregated bandwidth and the throughput are the preferred metrics, as they are what one seeks to improve. The bandwidth indicates the effectiveness of the solution, while the throughput is of uttermost importance to the application. Because most applications, as well as TCP, require data to arrive in-order, out-of-order data will cause processing delays.

The application-specific bandwidth aggregation technique was evaluated together with

quality-adaptive video streaming. The goal of adding a second link is to increase the achieved video quality, and both video quality and deadline misses were used as metrics for evaluating the performance gain. Deadline misses give an indication of how correct the solution is, i.e., is the technique able to request a higher quality video without causing playback interruptions. If a video segment is not ready for playout, it will cause interruptions in playback and annoy the user.

Our workloads were based on a combination of applications generating synthetic streams, real applications and real video clips. The transparent techniques were evaluated together with applications that generate a data stream of a given bandwidth or used as much of the capacity as possible, in order to get an impression of the possible performance gain offered by multilink. As transparent techniques can be used together with any kind of application, creating a workload for every scenario is not feasible. The application-specific bandwidth aggregation technique was evaluated using a real, variable bitrate encoded video showing a football match.

1.6 Contributions

In this work, we present multiple techniques for achieving efficient bandwidth aggregation. Unlike several of the techniques presented in the related work, all the techniques introduced in this thesis address the different challenges presented earlier, and can be used in real-world networks. Each technique was evaluated both in a controlled network environment and real-world networks. The main contributions are summarized here:

- **A framework for enabling multiple links dynamically and automatically:** Different operating systems vary in how they behave when the client device is connected to more than one network. In order to provide a platform-independent, generic way to enable the use of multiple links, as well as allow for easier design, development and deployment of multilink applications, we developed our own framework called MULTI. MULTI automatically detects new network connections, configures the routing subsystem and notifies the application.

- **Application-specific bandwidth aggregation:** Two of the most popular, bandwidth-intensive services on the Internet today is bulk data transfer and video streaming. We have created a technique for increasing the performance of bulk data transfers when multiple links are present. Each file is divided into smaller pieces, and the pieces are requested over the different available links. The technique was refined and optimised to meet the demands of video streaming, and was evaluated together with a segmented HTTP-based quality-adaptive video streaming solution. Our approach

utilized close to 100 % of the available bandwidth, and, compared to when a single link was used, the video and playback quality increased significantly.

- **Transparent bandwidth aggregation for UDP and TCP:** Transparent bandwidth aggregation must be used when it is not desirable or possible to change the applications that would benefit from bandwidth aggregation. We have focused on improving the performance of applications using the two most common transport layer protocols, TCP and UDP. Due to their different characteristics and behavior, separate techniques are needed for each protocol. Our techniques operate on the network layer, and the technique for transparent bandwidth aggregation of UDP was able to cope well with both bandwidth and latency heterogeneity. The performance of the TCP technique, however, depended on the latency heterogeneity. Based on our experiences, observations and knowledge of TCP's design and default behavior, we have not been able to design a bandwidth aggregation technique for TCP that is independent of latency heterogeneity. Instead, we present the design of a semi-transparent bandwidth aggregation technique that is more robust to latency heterogeneity.

The work presented in this thesis has resulted in 10 peer-reviewed conference publications, one patent-application and one journal article. Descriptions of the publications are given in appendix A.

1.7 Outline of thesis

Chapter 2 presents the related work in the fields of multilink and bandwidth aggregation.

Chapter 3 introduces our multilink framework MULTI. We describe how it is designed, how it can be used and how it was implement for Linux, BSD and Windows 7.

Chapter 4 presents our application-layer bandwidth aggregation technique, based on HTTP. After giving an introduction to how HTTP can be used to support simultaneous use of multiple links, we present how bandwidth aggregation was used to enhance the performance of quality-adaptive streaming.

Chapter 5 presents the transparent bandwidth aggregation techniques for UDP and TCP. The techniques are built around the same core concepts. However, based on our experience and evaluations, we were not able to design a transparent bandwidth aggregation technique for TCP. Therefore, a semi-transparent technique based on the concept of connection splitting is also described.

Chapter 6 concludes the thesis and presents ideas for future work.

Chapter 2

Background and related work

Bandwidth aggregation has been a research topic for several years and different solutions have been proposed at every layer of the TCP/IP stack. However, the existing work has mostly involved 1) stable links, 2) fully controlled network environments, 3) will not work with clients connected to independent networks, or 4) require changes to the existing infrastructure. In other words, the deployment and link heterogeneity challenges, as well as the dynamic behavior of wireless links, have largely been ignored or not considered properly by existing research. Ignoring any of these challenges will lead to a less than ideal performance in the real world, if the solution/technique works at all. For example, not properly considering the effect of reordering will lead to bad throughput. Also, new protocols or protocol modifications take years until they reach standardization and widespread deployment, if it ever happens.

In this thesis, we have focused on transparent and application-specific bandwidth aggregation. The application-specific bandwidth aggregation technique presented in this thesis was optimised for quality-adaptive video streaming, which has, to the best of our knowledge, not been done before. The amount of related work we found were therefore limited. However, certain techniques and ideas could be used as inspiration or borrowed from other types of application-specific bandwidth aggregation, as well as parallel download.

Transparent bandwidth aggregation requires knowledge about the transport protocol being used. The transport layer is the second highest layer in the IP-stack, and is responsible for providing end-to-end communication. When an application wants to send data through a network, it first has to open a network socket. This socket is then bound to a specific transport protocol. There exists a large number of transport protocols, each offering a different, sometimes partially overlapping, set of features, and the behavior is defined by a set of rules and mechanisms. In order for a transparent bandwidth aggregation technique to aggregate bandwidth efficiently, it has to be designed according to and

support the behavior of the targeted protocol(s).

Even though there are several different transport protocols, only two have so far reached widespread deployment and is supported by all major operating systems, TCP and UDP. The transparent bandwidth aggregation techniques developed during the work with this thesis are targeted at improving the performance of these two protocols. In the first part of this chapter, we describe TCP and UDP. Knowledge about TCP and its features is also needed in order to understand parts of the related work, which is presented in the second part of this chapter.

2.1 Transport protocols

In order to understand the transparent bandwidth aggregation techniques presented in chapter 5, as well as parts of the related work, knowledge about TCP and UDP is needed. In this section, we describe the two transport layer protocols.

2.1.1 UDP

The User Datagram Protocol, UDP, was standardized in 1980 and is described in RFC768 [60]. It provides a simple, best-effort protocol for applications to communicate.

UDP allows applications to send messages (called datagrams) to each other without setting up a connection. In addition, UDP does not provide any guarantees for reliability, ordering or data integrity. In other words, UDP will not react if packets are lost during transmission, have been tampered with or arrive in incorrect order. Supporting any of these features is offloaded to the application, in order to reduce the processing overhead. Also, unlike TCP, UDP is compatible with both packet broadcast and multicast.

Applications using UDP are mostly those concerned with latency, for example voice over IP or games. The loss of sound or movement while waiting for a packet retransmission will have a more significant effect on the user experience than dropping the packet. Another common use for UDP is IP tunneling. An IP tunnel works by encapsulating the original data packet (containing both network and transport layer header) inside another packet, and is used to for example create secure communication channels between corporate sites. Except for a reduction in the amount of payload one packet can contain, the behavior and performance of the original transport protocol is not affected, as UDP only provides a best-effort service and introduces no new mechanics (like congestion control).

UDP is, for the reasons described in the previous paragraph, used as the tunneling protocol by our multilink framework MULTI (introduced in the next chapter). Also, in chapter 5.1 we describe a technique for efficient, transparent bandwidth aggregation of UDP-streams.

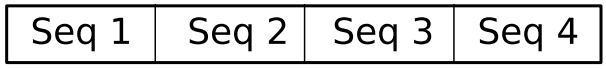
2.1.2 TCP

TCP [61], or the Transmission Control Protocol, is used by popular services like SMTP (e-mail), HTTP (web) and FTP (file transfers). It is significantly more advanced than UDP and has several desirable and advanced features:

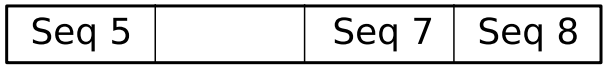
- **Connection-oriented** - A connection has to be established before data can be transferred.
- **Stream-oriented** - The application can send a continuous stream of data for transmission, TCP is responsible for dividing it into suitable units for the network layer to process.
- **Reliable** - All sent data will arrive and be delivered in order to the application. In addition, TCP uses checksums to detect (and reject) corrupted packets.
- **Flow control** - Throughout the connection, the receiver keeps the sender updated on how many packets it is able to receive. The sender has to adjust the packet send rate to avoid exceeding this limit, otherwise the receiver will not be able to process packets fast enough and overflows will occur.
- **Congestion control** - To stop the sender from consuming so much bandwidth that it would affect the performance of other streams, TCP limits the packet send rate. In addition, TCP assumes that all packet loss is caused by congestion, and reduces the send rate when loss occurs.

One of the design goals of TCP is to be considerate to other streams sharing the same path. TCP assumes that all packet loss is due to congestion, and as long as no packets are lost, the sending rate is increased. When congestion occurs, the protocol follows an "Additive Increase, Multiplicative Decrease" scheme (AIMD) to adjust the sending rate. The name of the scheme implies that the sending rate increases linearly and decreases exponentially.

To ensure reliability, each TCP packet is marked with a sequence number. This number is the byte offset for the packet's payload (the data contained in the packet) in the file/stream that is transferred. To let the sender know that the data has been received, the receiver sends an ACK-packet containing the next expected sequence number. In other words, the receiver lets the sender know that it has received all bytes up to this sequence number. Should a packet arrive out of order (i.e., the sequence number is higher than the expected one), the receiver sends a dupACK. Exactly what these are used for, and how the sender reacts to them will be discussed later.



(a) All packets arrive in order and is delivered to the application.



(b) The packet with sequence number 6 is lost. The two last packets cannot be delivered to application before the lost packet is retransmitted (and received).

Figure 2.1: Example of a TCP receive buffer.

If a packet arrives out of order, it will be buffered (stored) at the receiver until the expected packet(s) arrive (figure 2.1). The size of the receiver's advertised window (`rwnd`) states how much outstanding data the receiver is able to store, and every ACK contains its size. The sender has to adjust the send rate accordingly, and this is the flow control [5]. At the sender, the congestion window (`cwnd`) determines the amount of data that can be sent before receiving an ACK. The `cwnd` and `rwnd` change throughout the connection, and the lowest of the two decide the transfer rate. RFC2581 [5] states that TCP is never allowed to send data with a sequence number higher than the sum of the highest acknowledged sequence number, and the minimum of `cwnd` and `rwnd`.

TCP congestion control

Congestion control is concerned with how much network resources senders are allowed to consume, and its goal is to avoid what is known as a congestion collapse. This is a condition where the network is constantly overloaded, thus, the delays will be long, the loss rate high and the throughput low. A large number of TCP protocol variations have been developed in order to optimise TCP's performance in different scenarios (very high speed links, wireless, and so on), and what they alter is mostly related to the congestion control. However, in order to understand the work presented in this thesis, only knowledge about the concepts and mechanisms introduced by TCP Reno [5] and TCP New Reno [27] are needed.

When starting a transmission, **TCP Reno** uses a technique called slow start to avoid sending more data than the network can support, i.e., to avoid causing congestion. During the initial phases of a connection, TCP determines the maximum segment size (MSS, the largest amount of data a TCP segment can contain) and initializes the `cwnd` to be less than or equal to `2*MSS` (depending on the implementation). The size of the `cwnd` is increased by one MSS for every ACK that the sender receives, which means that the size of the congestion window doubles for every RTT. Provided that there is enough data to transfer and no packets are lost, the connection will first be allowed to send one packet, then two,

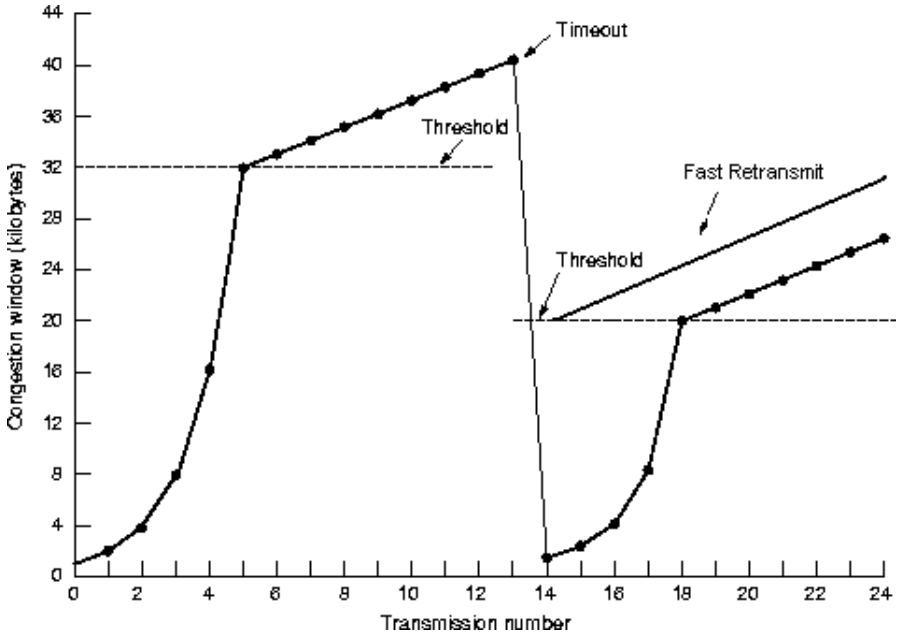


Figure 2.2: An illustration of slow start, retransmission timeout and fast retransmit

then four, and so on without receiving ACKs. Figure 2.2 shows how the `cwnd` grows, and also what happens when a packet is lost. The latter will be discussed later in this section.

This doubling of the `cwnd` continues until a pre-determined threshold, called the slow-start threshold (`ssthresh`), is reached, or packet loss occurs. When `ssthres` is exceeded, the connection enters the congestion avoidance phase. The `cwnd` is increased by one MSS for each RTT, thus, we have exponential growth before `ssthresh` is passed and linear growth after. To avoid relying on clocks (which are often too coarse), it is recommended [5] that the `cwnd` is updated for every received ACK using the following formula:

$$\texttt{cwnd} = \texttt{cwnd} + (MSS * MSS/\texttt{cwnd}). \tag{2.1}$$

TCP Reno uses two different techniques to discover and retransmit lost packets. If no acknowledgment is received before the retransmission timer expires, a retransmission timeout (RTO) is triggered. Exactly how and when the timer is updated is OS-specific. When a timeout occurs, the `ssthresh` is set to `cwnd`/2 and the connection re-enters slow start. The reason for reducing `ssthresh` is that TCP Reno assumes that all loss is due to congestion. The estimated share of the bandwidth was apparently too high, and must therefore be reduced. Also, a lower `ssthresh`-value ensures a slower growth rate. This stops the connection from suddenly flooding the network with more data than it can handle.

Another important event that occurs when an RTO occurs, is that the retransmission timer is doubled. This is called the exponential backoff and will inflict severe delays if

the same packet is lost several times. In for example Linux, the minimum RTO (`minRTO`) value is 200 ms, meaning that if the packet is lost three times, then the wait for the next retransmission is over a second long ($200 * (2^0 + 2^1 + 2^2)$).

As mentioned in the previous section, when a receiver receives a packet with a higher sequence number than the one expected (e.g. if the previous packet was lost or reordered), it sends a dupACK. This is done to let the sender know that it has not received all of the previous packets, meaning that it can not deliver any more data to the application. Because of the in-order requirement, data delivered to the application must form a continuous byte range.

Until the packet with the expected sequence number arrives, the receiver will continue to send dupACKs for every received packet. After N dupACKs (three is a frequently used value) are received, the sender will retransmit the first lost packet (this is called a fast retransmit) and enter fast recovery. In this state, `ssthresh` is set to `cwnd`/2 (as when an RTO occurs), but the connection does not have to go through slow start again. DupACKs indicate that the packets are buffered at the receiver and no longer consume network resources.

Instead of `2*MSS` (or less), `cwnd` is set to `ssthresh + 3*MSS`. The latter part of the last equation is there to ensure that new data can be sent, as long as it is permitted by the `cwnd` and `rwnd`. The three packets that generated the dupACKs have not been "properly" acknowledged, and are therefore occupying space in the congestion window. Thus, the `cwnd` has to be artificially inflated to allow new data to be transferred. The `cwnd` is increased by one MSS for every received dupACK for the same reason. When the next ACK arrives, the `cwnd` is reduced to `ssthresh` and the connection leaves fast recovery.

Because an ACK indicates that the network is no longer congested, it is "safe" to start with a congestion window size of `ssthresh` (since this is the estimated share of bandwidth). If the sender continues to receive dupACKs, the fast retransmit/recovery will be repeated until the `cwnd` is so small that no new packets can be sent. If the retransmitted packet is then lost, a retransmission timeout will occur.

The way TCP Reno deals with fast recovery is not ideal when there are multiple packet losses. If the ACK that makes the sender leave fast recovery is followed by N dupACKs, the sender will enter fast recovery again, halving the `cwnd` once more. **TCP New Reno** [26] modifies the way fast recovery/retransmission works, it stays in fast recovery until all unacknowledged data has been confirmed received.

To be able to do this, New Reno uses a partial acknowledgment concept. When entering fast retransmit, the highest sequence number sent so far is stored. Every received ACK is then compared against this value, and if the acknowledged sequence number covers

the stored sequence number, then it is safe to leave fast recovery. Otherwise, more packets will be lost and the first unacknowledged packet is retransmitted (when an ACK arrives).

The partial acknowledgment concept also allows TCP New Reno to check for false dupACKs. The highest transmitted sequence number is also stored whenever an RTO occurs. When three dupACKs are received, the sender checks if they cover this sequence number. If they do, then the connection enters fast recovery. Otherwise, the dupACKs are for packets sent before the timeout, thus the lost packet is already retransmitted.

One scenario where false dupACKs may be a problem, is when there are long RTTs. If a packet is lost and the N consecutive packets arrive, the dupACKs might not get back before an RTO is triggered. When they are received they will acknowledge a packet with a lower sequence number than the highest transmitted. The sender will then detect these dupACKs as false and not enter fast recovery.

TCP New Reno does not fix all the flaws in TCP Reno. It could still take many RTT's to recover from a loss (since a sender have to wait for the ACK/more dupACKs), and a sender will have to send enough data to get the receiver to respond with N dupACKs.

TCP SACK

Selective Acknowledgment (SACK) [40] [53] is a strategy that handles multiple packet losses better than plain TCP Reno and TCP New Reno. When a packet arrives out of order, TCP will send a dupACK acknowledging the last packet that arrived in order. Thus, the sender will be informed that one packet has arrived, but not told which one. This forces the sender to wait at least an entire RTT to discover further packet loss (since it must wait for a retransmission to be acknowledged), or it might retransmit packets that have already been received (if allowed to by the flow- and congestion control). In other words, multiple packet losses can cause a significant reduction in throughput.

With SACK, however, the dupACK will also contain the sequence number of those packets that have arrived out of order. SACK does this by using SACK Blocks (which are stored in the option part of the TCP header), and each block contains the start and end sequence number of the most recent continuous byte ranges that have been received (up to three). This leads to an increased throughput, the sender no longer has to wait at least one RTT to discover further packet loss, and only the packets that have not arrived will be retransmitted.

TCP fairness

One of the most important aspects of TCP is the fairness principle. If N TCP streams share the same link, they will each get an equal share (1/N) of bandwidth. Congestion control, which was discussed earlier, is used to enforce fairness by limiting the packet send

rate (the `cwnd`), and reducing it if needed. As previously mentioned, TCP assumes that all loss is due to congestion, i.e., the stream has exceeded its fair share of the bandwidth. For example, a TCP session is never allowed to send more than the congestion window allows to avoid flooding the network, and exponential backoff tells the streams to hold back due to congestion.

However, the fairness principle does not apply to all streams. If several streams with different RTT use the same link, then those with a short RTT will have an advantage. Their send rate will grow faster, e.g., in slow start the `cwnd` will double for each RTT, and they will spend shorter time recovering from a loss. Thus, they will consume a larger share of the bandwidth.

The work presented in this thesis will, according to some, violate the TCP fairness principle, or at least allow for an uneven use of resources. Even though each TCP stream is fair, opening multiple TCP-streams (or TCP-friendly streams) will allow some clients to consume more bandwidth than others (for example at the server or in shared bottlenecks). However, we believe that ensuring a fair usage is a network engineering and protocol design task. We have only focused on the potential benefit of using multiple links simultaneously.

TCP and reordering

By default, TCP will struggle with the packet reordering caused by dividing a TCP stream over heterogeneous links. Each out-of-order packet will cause the receiver to generate a duplicate acknowledgement, and the receiver triggers a fast retransmit when it has received N dupACKs. When doing fast retransmit, TCP reduces the `cwnd` and, thereby, the throughput. In other words, if the reordering is higher than N, often unnecessary retransmissions will frequently be triggered.

In order to be more robust against reordering, different OS-specific optimizations have been implemented. Linux, for example, adjusts the dupACK-threshold dynamically if reordering is constant. However, this technique is a trade-off. In addition to struggling with dynamic packet reordering, it will delay the triggering of the fast retransmits that are caused by packet loss. Because we have not developed or optimised for a specific OS or TCP-variant, we have ignored the behavior of the different reordering specific optimizations.

2.2 Related work

Bandwidth aggregation, also referred to as inverse multiplexing or multilink transfer in related work, is the process of aggregating the bandwidth of physical links into one logical link. The main purpose is to give applications access to more bandwidth than a single

link can provide. Bandwidth aggregation techniques have been proposed on all layers of the network stack, and the structure of this section follows the IP-stack.

In addition to the work done within the field of bandwidth aggregation, we also present different techniques for parallel download, i.e., when a file is downloaded from multiple servers. Parallel download techniques apply when the content is mirrored and the client has access to more bandwidth than any one server. Even though the scenarios are the opposite of each other, bandwidth aggregation is motivated by the assumption that the server has more available bandwidth than a single link at the client, several of the same challenges have to be solved in order to increase performance.

2.2.1 Link layer

The link layer is the lowest layer of the TCP/IP stack and is responsible for communicating with the actual, physical network. Bandwidth aggregation at this layer is commonly referred to as channel or Ethernet bonding, and requires support in an external device (for example a switch) as well as changes to the operating system kernel.

Channel bonding was motivated by the fact that network bandwidth usually increases by an order of a magnitude for each generation (10 Mbit/s, 100 Mbit/s, 1000 Mbit/s and so on). Upgrading equipment to support the next generation is expensive and will in many cases be unnecessary. For example, if a server needs 200 Mbit/s of available bandwidth, the only choice without bonding would be to invest in 1000 Mbit/s equipment.

The general idea of channel bonding is that a client stripes its traffic over multiple network interfaces, according to some scheme. This traffic is merged at an endpoint, for example a switch, and the client and the endpoint must be connected directly by a physical medium. I.e., if wired networking is used, then one end of the cable is connected to the client, and the other to the endpoint. Also, all interfaces have to be connected to the same endpoint. The endpoint will also stripe traffic going back to the client and it has to support the aggregated bandwidth. Using bonding, 200 Mbit/s can be achieved by for example using two 100 Mbit/s cards.

In addition to commercial solutions like Cisco's EtherChannel [13] and Atheros' WLAN-specific "Super G" [7], the IEEE has created a standard for channel bonding, IEEE 802.1AX Link Aggregation [6]. To deal with the packet reordering that can occur due to link heterogeneity, IEEE 802.1AX supports different techniques. The most common approach to avoid reordering is to send packets that belong together (for example a TCP connection) over the same link. Unfortunately, this does not guarantee an even distribution of traffic over the multiple links. Also, this will not increase the performance of a single stream.

A potential solution to the load balancing and reordering problem is presented in [2].

Even though the authors have not targeted IEEE 802.1AX specifically, they suggest a more advanced striping algorithm, combined with a resequencer at the receiver. The purpose of the resequencer is to buffer out-of-order packets until reordering has been resolved. Their striping algorithm, Surplus Round Robin (SRR), is targeted at downstream traffic (even though it can be applied to uplink as well) and the sender (or switch) adjusts the weight of the links dynamically based on feedback from the receiver. However, this solution does not properly consider the effects the resequencer can have on the transport protocol. Also, SRR requires the sender and receiver to frequently sync up in order for the striping to be correct.

Another striping algorithm is presented in [71]. The authors have focused on wide-area wireless networks (WWAN like 3G), and suggest a link-layer striping approach where packets are fragmented and distributed based on the MTU of an interface. The MTU is adjusted dynamically based on link layer feedback, thus, dynamic load balancing is achieved. Fragmented packets are reassembled by an endpoint before they are forwarded to another network. This approach works well with homogeneous links, however, the technique will not work with significant link heterogeneity. The delay while waiting for the missing packet fragments will increase along with the heterogeneity, affecting throughput. The loss rate will also have a significant effect: While WWANs typically have a low loss rate (due to aggressive link layer retransmissions), a high packet loss can be a problem in for instance public WLANs.

The Point-to-Point Protocol (PPP) [68], which is frequently used for dial-up connections, is an example of a link layer protocol with support for channel bonding. Through Multilink PPP [70] (MLPPP), multiple channels can be aggregated. However, MLPPP only supports round robin striping or an MTU-based approach, similar to the one described in the previous paragraph, and does not perform any resequencing (only merging of fragments). Thus, it is not well suited for use with heterogeneous links.

Different approaches for bandwidth aggregation at the link layer exist. However, they all have different drawbacks and can not be applied to our scenario. In addition to the need for operating system support (or for changing the operating system), link layer bonding requires the interfaces to be connected directly to the endpoint, and all interfaces have to be connected to the same endpoint. We aggregate the performance of different network technologies, with interfaces belonging to different networks.

2.2.2 Network layer

The network layer is responsible for routing, i.e., forwarding packets to another network or delivering them to the current machine. The primary addressing and routing protocol in computer networks today is IP (version 4). Every connected network interface is given an

IP-address, and doing bandwidth aggregation on the network layer offers several advantages. For example, solutions can be completely transparent and bandwidth aggregation does not require modifying the IP-protocol. As IP is connectionless, packets are simply forwarded to the machine/interface with the correct IP address.

The most common way of enabling bandwidth aggregation at the network layers is to establish IP-tunnels between an endpoint and the different interfaces at a multihomed client. Packets are striped across the multiple links according to a scheme, similar to link bonding. However, unlike link bonding, network layer solutions for bandwidth aggregation does not require the client and endpoint to belong to the same network. Unless they are blocked by a firewall or another middlebox, the IP packets will be routed automatically to their destination.

An example of a network layer bandwidth aggregation solution is presented in [59]. Here, the authors propose using IP-in-IP tunnels between a multihomed client and a server, and they present a striping scheduler which is able to utilize the full bandwidth of the links. However, the work is based on the assumption that bandwidth heterogeneity presents the largest challenge, and that latency heterogeneity can be compensated for by adjusting the packet size. Our observations and experiences contradict this assumption - compensating for bandwidth heterogeneity is often the easiest of the two heterogeneities, and the latency does not only depend on the packet size. For example, the queue delay imposed by routers or link layer retransmissions can be significant. In addition, even though the solution is presented as transparent, it requires changes to the IP and transport layer header. Also, TCP must be tuned in order to achieve maximum performance. Finally, IP-in-IP tunneling does not work between most real-world networks, as both end points must be able to communicate directly for it to work. This is often not possible due to for example NAT, due to the required port translation taking place in the NAT. As IP has no knowledge of port, NATs are by default not able to create a mapping.

Another network layer bandwidth aggregation approach is described in [12]. The authors focus on WLANs and downstream traffic. A proxy is used and the desired traffic to and from the multihomed client goes through the proxy. An algorithm called *Earliest delivery path first* is used to stripe packets efficiently. The algorithm relies on knowledge about the queue length at the base stations to which each client interface is connected, and then the proxy sends packets over the link with the earliest delivery time. Earliest delivery path first is able to reduce reordering and utilizes the links effectively (the queue length is another way to express the current available link capacity). However, in real world networks, proxies (or external machines) do not have access to link-layer information for the active base stations. Also, Earliest delivery path first has only been evaluated with simulations, and, for example, the effect of fluctuating link performance has not been

considered.

A network layer approach which does not make use of proxies or tunnels, and is truly transparent, is the earlier mentioned Equal Cost Multipath Routing [34]. It allows users to assign weights to the network interfaces, and then these weights are used to decide which interface shall be used when network connections are opened. ECMP works well in a static scenario and with applications which opens several connections (for example P2P-applications), but it is not well suited for a real-world scenario with dynamic link behavior. Without static links, the weight of each link changes dynamically. In addition, a transparent bandwidth aggregation technique should not rely on a certain application behavior.

The current network layer approaches to bandwidth aggregation does not consider the challenges present in real-world networks properly, or are based on unrealistic assumptions (for example about available information). A transparent network layer bandwidth aggregation solution or technique must work in real-world networks and with real-world link behavior. Also, one of the requirements to the techniques presented in this thesis is that they cannot rely on changes to existing protocols. Therefore, none of the existing solutions seems complete, and can not be applied to the scenario we have focused on.

2.2.3 Transport layer

When applications want to communicate over a network, they have to create a network socket. This network socket is bound to a specific transport protocol, and transport protocols are responsible for the end-to-end communication. In other words, an application passes the data that will be sent to the network socket, and then the transport protocol sends and delivers the data to an application on the other machine.

Bandwidth aggregation at the transport layer has mostly focused on modifying the Stream Control Transmission Protocol [72] (SCTP) or TCP. This requires changing the OS' network stack at both the client and server. The rest of this subsection is focused on and is structured according to these two protocols. Because the transparent bandwidth aggregation techniques presented in this thesis operate on the network layer, the transport layer solutions cannot be applied directly. However, some of the concepts and ideas can be reused by the network layer techniques, as they have to be tuned to the behaviour of the transport protocol(s).

SCTP

Stream Control Transmission Protocol was motivated by the need for better signaling support than any current transport protocol could support. Signaling is important to for

example multimedia applications, and the first version of the protocol was introduced in 2000. SCTP is similar to TCP in that it provides reliable, message-oriented data transport, and the congestion control follows the AIMD-rules and uses the same mechanisms. SCTP also has native multi-homing support. A connection, known as an *association*, can consist of several *paths* established over different interfaces. However, by default, the additional paths are only used to increase reliability. When an association is initiated, one interface at each host is selected to form the primary path, and this path is used for all communication. The associations only changes which path to use when the current is no longer available.

SCTP-bandwidth aggregation is known as *concurrent multipath transfer* (CMT), and different CMT-approaches have been suggested. CMT was introduced in [38]. In addition to transferring data over multiple paths, the most important changes compared to plain SCTP are that flow and congestion control are decoupled, and that each path is assigned its own congestion window. With a shared congestion window, congestion control will be invoked when the slowest path gets congested. This limits the growth of the congestion window and causes underutilization of the other paths. The flow control still belongs to the association and makes sure that the receiver(s) is not overwhelmed with traffic. Packets are sent over the first path with an open congestion window.

With separate congestion windows, the challenge of transport layer reordering is removed. Unless there is internal reordering on one path, all packets will arrive in order (except when there is packet loss). However, the association will experience reordering. The different transport layer bandwidth aggregation solutions we have found have all ignored the application layer effect of the bursty delivery pattern caused by this reordering. Instead, the focus has been on removing the unnecessary retransmissions.

CMT allocates one virtual buffer to each path, and this buffer contains meta-information about the sent packets. By parsing the SACK-field of the SCTP-header, the sender determines which packets are lost and which have been reordered. In addition to avoiding superfluous retransmissions, the information is used to ensure proper growth of the congestion window. CMT was evaluated using simulations and achieved a good bandwidth aggregation. For example, the transfer time of a file was significantly reduced compared to a single path. However, the authors only analyzed the performance for different loss rates, the links were otherwise homogeneous.

More advanced CMT-techniques are introduced in the SCTP-variations in [3] and [25], which are both based on the core CMT-concepts (such as decoupling). LS-SCTP [3] uses both the congestion window and the measured RTT when scheduling packets. A path monitor is used to gather statistics about the paths and keep the set of active paths updated, while retransmits are never sent over the same path as the original packet. LS-

SCTP was also evaluated using simulations and homogeneous links, and it scaled with the number of available paths and achieved a good bandwidth aggregation in the presence of cross-traffic.

W-PR-SCTP [25] is built on the concept of partial reliability, meaning that a packet is only retransmitted a limited number of times before it is assumed delivered. Partial reliability cannot be applied to every scenario, for example a file transfer requires that all data arrives, but partial reliability fits well with the requirements of multimedia streaming. W-PR-SCTP is inspired by TCP Westwood+ [52], and it continuously estimates the available bandwidth. The bandwidth measurements are used to determine which path a packet shall be sent over, and the authors achieved efficient bandwidth aggregation in a controlled environment with stable link heterogeneity.

Because we have focused on transparent bandwidth aggregation at the network layer and application specific bandwidth aggregation, none of the SCTP-solutions for bandwidth aggregation can be applied directly. Also, one goal with the techniques presented in this thesis is that they cannot rely on transport protocol changes. However, some of the ideas introduced by the SCTP-solutions can be reused. For example, our bandwidth aggregation technique for UDP traffic schedules packets based on the available space in a congestion window, similar to CMT [38].

TCP

TCP is designed to allow communication between two end-hosts with a single, unique identifier (for example an IP-address), and will by default never support multi-homing. In order to augment TCP with support for bandwidth aggregation, protocol extensions or new TCP-based protocols are needed. Two such protocols are pTCP [35] and mTCP [79].

pTCP makes use of a striping manager (SM), which wraps around multiple normal TCP connections (called TCP-v connections or subflows). Like SCTP CMT, pTCP makes use of a virtual buffer for each interface, and SACKs are parsed in order to separate packet loss from reordering. In order to determine which path to use when a packet is to be sent, the amount of open space in the congestion window is used. The authors are able to achieve good bandwidth aggregation in a network emulator, but the solution has some shortcomings. Most notably is how they propose to deal with reordering (which is not evaluated). The suggested approach is to combat reordering by increasing the SM's buffer size. This will work, however, artificially inflating buffers is generally not recommended due to memory constraints. Also, applications will still see a bursty traffic pattern.

mTCP is an improved version of pTCP. It does not provide any additional techniques for dealing with reordering, but it is able to detect shared paths in the network and react accordingly. mTCP assumes that if two subflows share a congested path, packet loss will

occur at almost the same time. Therefore, the mTCP sender assumes shared congestion if there is a strong correlation (in time) between when retransmits occurs for the subflows. If two paths share a congested path, the path with the lowest throughput is removed from the set of active paths. In scenarios with shared congestion, mTCP outperform pTCP. Otherwise, they show the same performance.

Currently, the multipath TCP variant with the most momentum is MPTCP [36]. MPTCP is currently under standardisation by IETF and borrows several of the ideas from SCTP CMT. An MPTCP connection consists of several independent TCP-connections (subflows) and packets are striped according to each connection's congestion window. A resequencer is used at the receiver to delay delivery of packets to userspace until reordering is resolved. Even though a lot of the fundamentals are in place, MPTCP still has some way to go before it is ready for deployment. For example, the developers have yet to agree on which congestion control to use, and if different congestion controls shall be used by the independent subflows and the main flow.

Another technique for doing bandwidth aggregation at the transport layer is PRISM [48]. PRISM consists of a network layer proxy and a sender-side TCP modification, and is targeted at downstream traffic. The solution is designed for community networking, where several multihomed mobile hosts in close proximity pool their resources together. Mobile hosts typically have one fast (WLAN) and one slower WWAN interface (e.g., 3G). In PRISM, the latter is used to receive data from the Internet, while the WLAN is used to share data (and requests) between hosts using a separate application.

With PRISM, IP-tunnels are established between the proxy and different interfaces at the clients, and packets are striped according to the current path capacity. In order to avoid congesting the paths, the proxy has implemented its own AIMD-like congestion control. The senders all use normal TCP and the proxy parses the generated ACKs (SACK-field), in order to separate packet loss from reordering. If reordering is detected, an ACK is buffered until reordering is resolved. When packet loss has occurred, before ACKs are released, the proxy notifies the sender of which path has experienced loss and of its share of the total bandwidth. Using this information, TCP-PRISM reduces the congestion window according to this share and does not, for example, halve it. Also, TCP-PRISM retransmits packets more aggressively, as the dupACK threshold is set to one. In their evaluations (simulations and real-world experiments), the authors show that PRISM was able to increase the performance over normal TCP by up to 310%.

As with the SCTP solutions for bandwidth aggregation, none of the TCP solutions can be applied directly as we have focused on the network layer and required that the transport protocols remain unmodified. However, we have made use of some of the concepts introduced, particularly by PRISM and SCTP CMT/MPTCP. For example, the trans-

parent bandwidth aggregation techniques for UDP and TCP makes use of a resequencer to reduce the reordering exposed to the higher layer.

2.2.4 Application layer

Application layer bandwidth aggregation solutions provide a trade-off between flexibility and fine-tuned approaches, and can be designed as application-specific extensions or middleware. As the behavior at the application layer is completely up to the developers, he or she can tune the bandwidth aggregation approach to fit the needs of a specific application or application type. This allows for the creation of the most efficient bandwidth aggregation techniques possible. However, the drawback is that the approach will not apply to other applications or application types. Also, changes have to be made to every application that wants to benefit from bandwidth aggregation, which in many cases is not possible due to, for example, the source code not being available. The application-specific bandwidth aggregation technique introduced in this thesis is targeted at quality-adaptive video streaming. To the best of our knowledge, this has not been done before and, thus, no directly related work exists. In this section, we therefore present examples of other application-layer bandwidth aggregation solutions and approaches, as well as parallel access schemes.

In [63], a middleware called Tavarua is introduced. The goal of the middleware is to enable ambulances to use the Internet to transfer more and richer information (for example video streams) to hospitals. Each ambulance has several small routers which are equipped with up to two WWAN interfaces. These routes are connected to a laptop using normal Ethernet, and an application named Tribe is responsible for detecting available networks and notifying the laptop of changes in link state. The laptop creates a virtual network interface for each available WWAN-connection, and IP-tunnels are used to transfer data between the laptop and the router. The routers forwards the packets it receives to a central server located at the relevant hospital, which then reassembles the stream (if needed).

In addition to Tribe, the laptop also runs another application, known as Horde. Different applications request a certain level of QoS from Horde, and Horde will react accordingly. For example, one application requires a reliable connection, while another has a higher bandwidth requirement than any single link can meet. In the latter case, Horde will stripe packets over the multiple available links (assuming the ambulance is within coverage range).

A more generic approach to application layer bandwidth aggregation is presented in [75]. An overlay network is created between two end-points, consisting of one TCP connection for each available network interface. Initially, each connection is assumed to have a certain capacity, and then probing is used to get a more accurate estimate. The

packet scheduler schedules packets iteratively and the approach is able to achieve a fair usage of the links. For each scheduler-iteration, paths are selected and the send rate adjusted in order to maximise the aggregated send rate of the source and distribute the bandwidth fairly.

The same authors propose another application-layer bandwidth aggregation technique in [74], targeting live video streaming. A client opens multiple TCP-connections to the server, and the server assigns one manager to each connection. The server maintains a FIFO-queue of packets, and only one manager is allowed to access the queue at a time. Operating systems allocate a certain amount of memory to each network socket (used for buffering), and a manager can consume packets until the buffer is filled (known as socket block). When a socket blocks, the next available manager accesses the queue (if any). Using their solution, the authors show that as long as the aggregated throughput is 1.6 times higher than the bandwidth requirement of the video, a good playback quality will be achieved (with a few seconds startup delay).

Parallel access schemes use multiple sockets rather than multiple interfaces, but the goal is still the same, i.e. to increase the throughput. The performance of a TCP connection is bound by the size of the send/receive buffer, and this is a problem with for example high bandwidth and high latency links. Even though the link can deliver more data, the high latency limits the growth of the congestion window and thereby the available throughput.

The buffers are, however, only bound to a single socket. In other words, opening multiple TCP connections will allow an application to overcome the challenge introduced by limited buffers. Two parallel access schemes are presented in [69] and [4]. A middleware named PSockets is introduced in [69]. PSockets provide a new type of network socket and is based on TCP. One TCP connection is opened for each active interface, and the data received by the PSocket (on the sender side) is divided into equal size pieces (based on the number of connections). Then, the data is striped across the different connections. XFTP [4] is an FTP-modification which makes use of a similar idea. A requested file is divided into 8 kB blocks and the blocks are sent over the first socket which is not blocked. The number of connections is adjusted dynamically based on changes in the measured RTT.

Another scenario where using multiple sockets can result in a performance benefit, is when the clients has more available bandwidth than the server. Several content providers use mirroring, meaning that the same file is stored at multiple locations, to achieve load balancing. By requesting different parts of a file from different servers, a potentially higher throughput can be achieved. The HTTP-protocol, which is the foundation for data communication on the world wide web and will be discussed in more detail later,

allows a client to divide a file into multiple, logical and independent blocks. An approach which combines HTTP with multiple servers is presented in [66]. A file is divided into a fixed number of blocks, and then the parts are requested through the different sockets. The number of blocks is not properly defined and only described as "significantly higher than the number of servers". An improvement to this technique is presented in [31]. Here, the client divides the file into blocks dynamically and on the fly, and the size is based on the measured throughput. In addition, pipelining is used to removed the ideal period between to requests.

As mention earlier, our bandwidth aggregation work at the application layer has focused on improving the performance of quality-adaptive video streaming, without changing the existing infrastructure (like [74] does). To the best of our knowledge, this has not been done before and, thus, no related work exists. However, we have used several of the ideas presented by the related work. For example, the streaming system that was used to evaluate the performance of our technique used HTTP and TCP for requesting and receiving video. The approaches for dividing files into blocks, presented in [4] and [31], served as inspiration for how to efficiently aggregate the bandwidth.

2.3 Summary

Bandwidth aggregation techniques have been proposed at every layer of the network stack. However, none of the existing techniques can be applied directly to address the scenarios and challenges we have focused on in this thesis: real-world networks and heterogeneous links. The existing solutions are mostly either based on 1) unrealistic or incorrect assumptions or requirements [12], 2) simulations [3, 12, 38], 3) fail to consider the different challenges present in real-world networks [2,59,70,71] or 4) cannot be applied to a scenario where the devices are connected to different networks [2, 6, 71].

In this thesis, we present two types of bandwidth aggregation techniques: application-specific and transparent. Our application-specific bandwidth aggregation technique was optimised for quality-adaptive streaming, which, to the best of our knowledge, has not been done before. However, some of the techniques developed for parallel download, particularly [4] and [31], served as inspiration for how to efficiently aggregate the bandwidth.

The transparent bandwidth aggregation techniques must support the behavior of the transport protocol. Before an application can communicate over a network, it must create a network socket. This socket is bound to a transport protocol, and each protocol offers a different set of features. The transparent techniques presented in this thesis have been designed for improving the performance of UDP and TCP. They are the two most common transport protocols in use today - TCP is reliable and has both flow and congestion control,

ensuring fair usage of the network, while UDP provides best effort transfer and is typically used by applications which requires low-latency.

Even though none of the existing solutions are a perfect match to the techniques presented in this thesis, we have used several of the concepts that were introduced by the related work. For example, our technique for transparently increasing the performance of UDP-based applications uses a scheduler similar to that of SCTP CMT [38], and both transparent bandwidth aggregation techniques use a resequencer [36, 48] to reduce the degree of reordering exposed to the higher layer.

Next, we present our multilink framework, called MULTI. MULTI is designed to overcome the deployment challenges and ease the development and deployment of multilink solutions.

Chapter 3

Designing an experimental multilink infrastructure

Enabling and using multiple links requires solving two deployment challenges. In addition to some operating systems requiring explicit configuration changes when multiple network interfaces become available, NAT limits the connectivity of a client. NAT is used by most ISPs to manage their networks, and limited connectivity is a problem because it makes a machine unreachable from outside its own network. Unless static port mappings are configured in the NAT, an external machine can by default not communicate with any machines placed behind a NAT.

During the work with this thesis, we have tried several different solutions and techniques for enabling and making use of multiple links. The goal was to find something that was able to overcome the deployment challenges, and ease the development, deployment and evaluation of multilink applications, techniques and solutions. For example, the system configuration has been done statically through scripts, while we have tried building transparent multilink solutions on top of many different, standardized IP tunneling solutions. However, none of the existing solutions were able to meet all of our requirements, or provide a flexible enough foundation for designing and deploying multilink techniques and solutions. Static configuration of routing tables are error prone and will not work in a mobile or roaming scenario, while the tunneling solutions have added a too large

	WLAN			
Throughput (KB/s)	min.	avg.	max.	std.
No Tunneling	623.5	694.9	720.4	21.0
L2TP Tunnels	571.9	646.3	686.5	33.3

Table 3.1: Observed TCP Throughput (KB/s) when measuring processing overhead with L2TP-tunneling.

overhead, been unstable or not offered sufficient multilink support. We identified two tunneling solutions that we got working with multiple links, the Point-to-Point Tunneling Protocol [33] (PPTP) and the Layer-2 Tunneling Protocol [73] (L2TP). However, PPTP only supports round-robin striping of packets, while the L2TP implementations we have found proved unstable and difficult to configure, and added a significant overhead. Table 3.1 shows the achieved throughput of an L2TP tunnel compared to when tunneling was not used (for TCP connections between the same machines over the same WLAN). On average, the throughput of the connection going through the tunnel was 7.5 % lower.

Because no existing work was able to meet our requirements, we developed our own, generic, platform-independent framework, called MULTI. MULTI is optimised for downstream traffic to multihomed clients and consists of different *modules*. Each module has a specific purpose, for example detecting changes in the available network connections. An application implementing MULTI is known as a *manager*, and MULTI exports a continuously updated list of active network interfaces to the managers. Then, it is up to the managers to make use of the interfaces by supporting scenario specific optimizations. For example, one manager could use the additional interfaces for redundancy, while another for striping packets. That network changes are detected automatically marks a significant improvement over previous approaches, which has mostly relied on static configuration through scripts.

MULTI can either be used to extend existing applications with multilink support, or to create transparent multilink solutions. When used to create a transparent multilink solution, a globally reachable proxy is used, and the manager must create a virtual interface. To an application, there is no difference between a virtual and a normal network interface. Thus, in order to benefit from the scenario specific optimizations, it is sufficient for an application to bind to the virtual interface. This can either be done explicitly by binding the network socket to the IP of the interface, or implicitly by routes being configured to go through the virtual interface. MULTI was used as the foundation for the implementations of the techniques that will be presented throughout this thesis, and the framework is designed to meet the following requirements:

- **Support the most common operating systems (Linux, BSD/OS X, Windows):** One of the goals with MULTI is that it shall ease the deployment of multilink solutions.

- **Handle roaming:** MULTI must be able to automatically detect changes in network connectivity, dynamically configure the network subsystem (if needed) and notify the manager.

- **Work without affecting other applications:** MULTI must not change the be-

havior of the network subsystem of a machine, only augment it with support for multiple links. The routing table must still contain a default route so that, for example, networked applications that do not use multilink will continue to work as normal.

- **Work without changes to the end-systems:** Changing different parts of the end-systems is in many cases not desirable or even possible. For example, applications are often proprietary and cannot be changed, while OS' and transport-protocol changes are complex, error-prone and will take a long time until widespread deployment. In addition, most changes have to be reflected at the remote server, which is often controlled by an independent third-party.

- **Be compatible with NAT and other middle-boxes:** NAT and other middle-boxes can limit the connectivity of a client. This is critical when MULTI is used in invisible mode, as the tunnels require each end-point to be directly reachable. The proxy will often belong to a different network than the interfaces at the client, and, thus, will in many cases by default not be able to send packets back to the client.

In the rest of this chapter, we will give a more thorough introduction to MULTI and introduce its different core components. We also give two examples of how it can be implemented and describe our Windows and Linux/BSD-implementations.

3.1 Overview

MULTI is a modular framework which monitors available networks (active network interfaces), and that provides the applications implementing it with a continuously updated list of available interfaces. An application implementing MULTI is known as a MULTI manager, and developers will implement scenario-specific optimizations or features in the managers. For example, in order to avoid connections failing when a network becomes unavailable, one manager can transparently hand the connection over to another network. The purpose of another manager might be to increase the available bandwidth by pooling links together, and stripe downstream traffic to a multihomed client.

3.1.1 Application specific use of MULTI

MULTI is designed to support two types of multilink solutions, application-specific and transparent. In the first type of solution, illustrated in figure 3.1, the application itself is extended with multilink support, i.e., the application is also the MULTI manager. This can be used when a developer has access to and can change both client and server

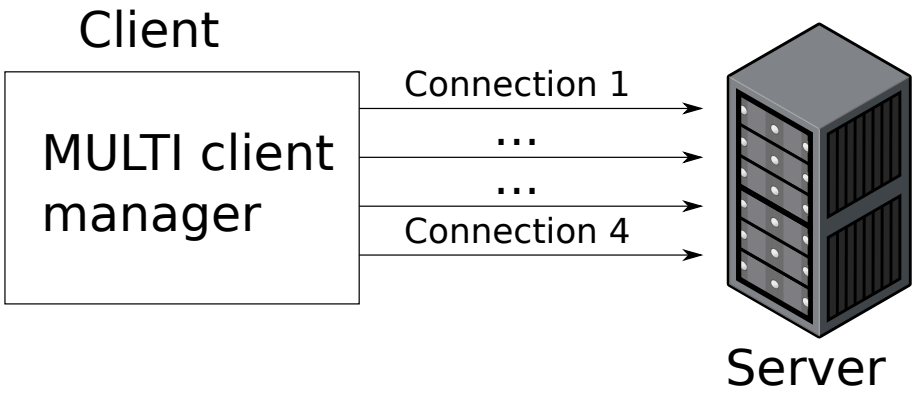


Figure 3.1: An example of MULTI running in visible mode on a client with four active network interfaces.

application, or the server already supports serving multiple connections. When used to create application-specific multilink solutions, MULTI is run in *visible mode* and is only used at the client. The application has access to a list of the available interfaces, and can, for example, create one connection to a server for each available interface. Then, for example, file requests can be distributed over the connections, or a video streaming server can send the base layer over one connection, and then use the others for the additional quality layers.

3.1.2 Enabling transparent multilink

Transparent multilink solutions, on the other hand, are transparent to the applications running on top, as well as to the operating system and transport protocols. Changing the actual application is in many cases not possible or desirable. For example, many applications are proprietary and the changes have to be reflected in the remote applications as well. The remote applications are often managed by independent third-parties, on machines where the client or the client developer has no control. Similarly, modifying the OS or transport protocol is a complex and error-prone process. The changes have to be implemented at every machine that will communicate, often by the OS-developers themselves. New transport protocols or changes to existing protocols take a very long time to gain acceptance and reach widespread deployment.

When used to implement transparent multilink solutions, MULTI runs in *invisible mode* and operates on the network layer. Working on the network layer allows a multilink solution to be transparent to both applications and the transport layer. Also, through the use of network tunnels, network layer solutions can be implemented in userspace, i.e., no change to the OS kernel is needed. In addition, operating on the network layer removes the addressing challenges limiting the deployment of link layer bandwidth aggregation. As discussed in the related work, link layer bandwidth aggregation requires all network

interfaces to be directly connected to the same endpoint. We aggregate links that are connected to different networks, using different technologies.

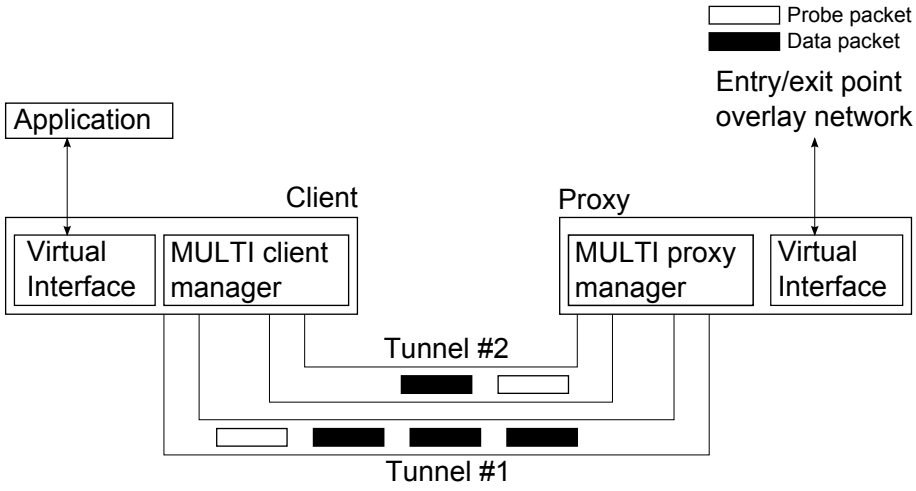


Figure 3.2: An overview of a transparent multilink solution using MULTI (on a client with two interfaces).

MULTI used in invisible mode is illustrated in figure 3.2, for a client with two network interfaces. In invisible mode, MULTI requires a globally reachable proxy to avoid changing the remote server, as unmodified hosts can only parse the original packet stream. In the figure, the MULTI client manager makes use of both the available network interfaces. Virtual interfaces are used to avoid changing the applications that want to benefit from the scenario specific optimization(s) in the managers.

When using MULTI in invisible mode, the developer has to implement both a client and proxy manager. One example of a transparent multilink solution is a bandwidth aggregation proxy. The developer can for example implement a packet scheduler in the proxy manager, in order to utilize the links efficiently, and then a resequencer at the client manager will buffer packets until packet reordering is resolved. Such solutions will be discussed in more detail in chapter 5.

3.1.3 Building a multilink overlay network

MULTI requires that both the client and proxy manager create virtual interfaces. Virtual interfaces behave as normal network interfaces and are for example given a regular IP-address. However, unlike a normal network interface, the virtual interface is owned and controlled by a user-space application. This application receives all the data written to the virtual interface and is responsible for sending it through an actual network. Virtual interfaces are typically used for IP-tunneling and applications can make use of a virtual interface, for example, by explicitly binding network sockets to it, or desired routes can be

configured through the interface. The latter is truly transparent multilink support, as the operating system will route any connection that matches the route through the virtual interface. Thus, the configuration change required by explicit binding is not needed. Exactly how the virtual interface is given an IP address is up to the developer, it can for example either be static or be retrieved from a pool.

For each active interface at the client, an IP-tunnel to the proxy is established. In other words, a multilink overlay network is built, and the manager is notified when a tunnel is added or removed. Data is then sent through or received from these tunnels, and in order to reach machines outside the multilink overlay network, the proxy uses IP forwarding and source NAT (SNAT). IP forwarding is an operating system setting and it must be enabled in order for a machine to be allowed to forward packets to their correct destination.

SNAT is also supported by most operating systems and is used to change the source IP of a packet to that of the proxy. This is needed to make sure that packets destined for the client are routed back through the proxy, and thereby the multilink overlay network. When packets destined for the client arrives at the proxy, SNAT automatically rewrites the destination IP to the client's IP (the virtual interface). In order to separate between packets that will and will not have their IP address rewritten (for example those destined for the proxy), SNAT, makes use of a mapping, like NAT.

3.1.4 Packet and processing overhead

The reason a virtual interface is needed at both the proxy and client, is that the data sent through the tunnels gets encapsulated and, thus, must be decapsulated. In addition to the additional network and transport layer header added when a packet is sent by the physical interface, a separate tunneling header is added. This header contains management information needed by the client and proxy manager. We use UDP as the tunnel transport protocol, due to its low-overhead and best-effort design, and the total per packet overhead is 24 bytes (including IP and UDP header). With a 1500 byte MTU (default for normal Ethernet), the overhead is 3.5 %.

Because MULTI and the MULTI managers are userspace applications, the processing overhead is only affected by the speed of and the load on the host computer. Most modern computers, including the ones used for the experiments performed during the work with this thesis, are at least able to support streams of several hundred megabit per second. For example, when measuring MULTI in the same network as used for table 3.1, the observed overhead (throughput reduction) was 3.5 %. This was caused by the packet overhead. That is, the processing overhead did not have an effect, and was significantly less than the 7.5 % seen with L2TP.

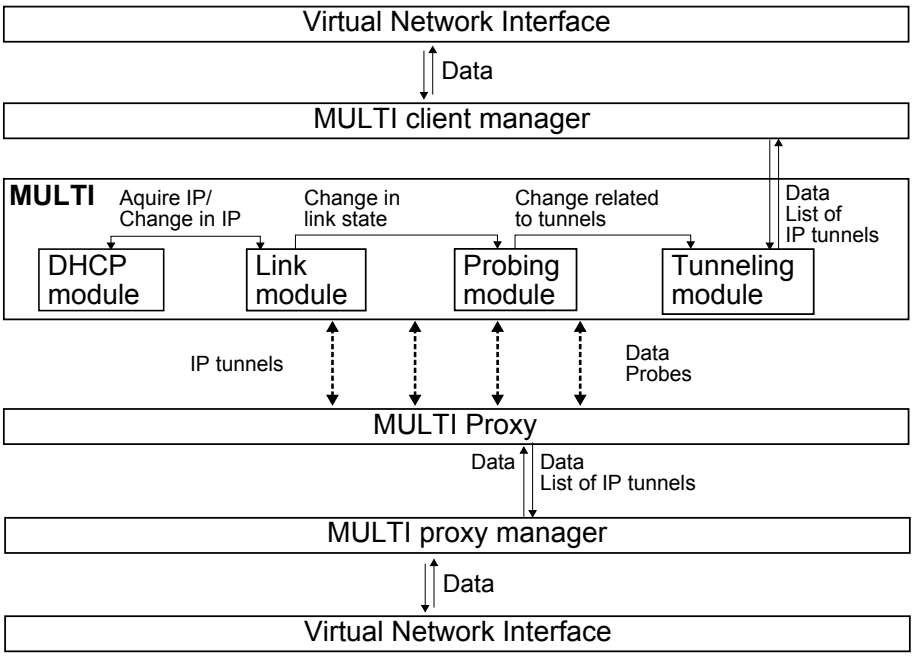


Figure 3.3: Overview of MULTI run in invisible mode on a client with four active interfaces.

3.1.5 Solving the connectivity challenge

In order to solve the limited connectivity challenge caused by clients placed behind NAT, MULTI uses NAT hole punching. At a given interval, probes are sent from the client to the proxy. The proxy stores the source IP and port of the packet (which is set to the NAT mapping if the client is behind NAT) and sends a reply to the client, using the NAT mapping as destination address. This forces the NAT to keep a state for each "connection", allowing packets to flow in both directions. When the proxy sends packets to clients, it sets the destination IP and port to the values it has stored. If the client is behind NAT, the NAT will match the destination address stored in the packet to the correct mapping, and forward the packet. The reason the NAT hole punching is initiated by the client, is that many NAT-implementations only create a state when the first packet was sent from within its network. The probe packets are also used for maintaining the overlay network, which is also why they are sent from clients that are not behind NAT.

3.2 Modules

MULTI consists of four core modules, illustrated in figure 3.3. The different modules are responsible for 1) monitoring the state of the OS' network subsystem (Link module), 2) configuring interfaces and routing tables automatically (Link and DHCP module), 3) managing the IP tunnels and constructing the overlay network (Probing module) and 4)

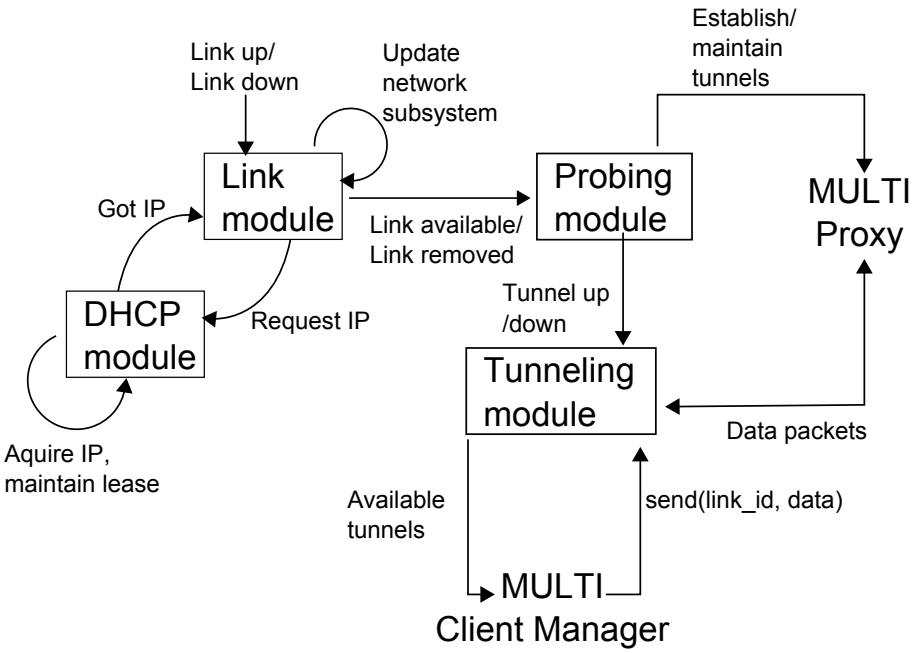


Figure 3.4: How the core modules in the MULTI client interacts (invisible mode).

sending/receiving data from the tunnels (Tunneling module).

When used in invisible mode, the MULTI manager communicates with the tunneling module. When a tunnel is added or removed, the module notifies the manager. The tunneling module also provides a function for sending data through the overlay network, and calls a callback function when data arrives. This function is specified by and implemented in the manager.

Only the link and DHCP module is active when MULTI is used in visible mode, as the manager makes use of the different interfaces directly. I.e., tunneling and thereby the probing module is not needed, or it will be implemented in the manager. The different modules can be regarded as black boxes that provide certain functionality and options to the MULTI Managers. How the different modules and the client-side manager interact is shown in figure 3.4. The following describes the functionality of the modules:

3.2.1 Link module

The link module monitors and detects changes in a client's network subsystem (when the interface is connected to a new network or the current connection is lost), and performs the necessary network and routing configuration. When a new link is ready or a link has been removed, either the manager (visible mode) or the probing module (invisible mode) is notified. A link is not considered ready until it has been assigned an IP address and the client's routing tables have been updated (if needed). IP addresses are either assigned statically (parsed from a configuration file) or requested using the Dynamic Host Control

Protocol [15] (DHCP) module. The link module also calculates a unique ID for each link. This ID can for example be used by the proxy to determine if tunnels have been added or removed.

On OSes where it is required, MULTI must update the routing tables, so that the operating system is able to continue routing packets in the presence of multiple links. If for example two default routes with the same priority exists on a Linux machine, the kernel will become confused when packets are sent, a routing decision will not be made and the packets discarded. The IETF have established a working group, Multiple Interfaces [1] (MIF), for designing a standard on how operating systems shall behave when a multihomed client is connected to multiple networks. However, their work has yet to be completed and they are currently behind schedule. If their work is successful, the configuration part of the link module can be replaced by or removed in favor of MIF (depending on whether the OS kernel supports it or not).

Present in proxy or client: Client only. If run in invisible mode, MULTI currently requires the proxy to have one or more fixed addresses, and that the network subsystem has been configured in advance. The reason for this requirement is that the client manager must know how to reach the proxy.

Options: The link module can be provided as a configuration file stating how the different interfaces shall be given an IP. This file specifies in which networks DHCP will be used, and in which network the interface(s) will be assigned a static IP. If an interface will be assigned a static IP, the file contains the IP address, netmask and gateway.

3.2.2 DHCP module

MULTI performs DHCP for those interfaces that are not assigned a static IP address. DHCP is a protocol used to assign IP addresses dynamically. When connected to a network using DHCP, the client first creates a DHCP DISCOVERY-message and broadcasts it, using UDP, to the pre-defined destination port 68. The network's DHCP server has a database of all valid IP-addresses, and, assuming that there is an IP address left, replies to the client with a DHCP OFFER-message. This message contains information such as the reserved IP address, the netmask of the current network and gateway, as well as the lease time for an IP. A client is only allowed to keep an IP address for a given amount of time, and has to apply for an extension of the lease before the previous expires. If the client accepts the OFFER, a DHCP REQUEST is sent, and the DHCP server replies with a DHCP ACK. If packets are lost during the DHCP packet exchange, it is the client's

[1] http://www.ietf.org/dyn/wg/charter/mif-charter

responsibility to retransmit packets. Also, if the client does not accept the OFFER or aborts the exchange, it sends a DHCP NAK. If a client no longer needs an IP address, it sends a DHCP RELEASE.

The DHCP module supports the full state machine (described in more detail in the DHCP RFC [15]), and a separate DHCP thread is started for each interface that will request a dynamic IP address. The link module is notified when either the DHCP module receives configuration information for an interface, or if an error occurs.

Present in proxy or client: Client only.

Options: If the link has previously been assigned an IP address through DHCP, the Link module includes the old IP address as an argument to the DHCP thread. DHCP then starts in the REBOOT state and will first request the old IP address.

3.2.3 Probing module

In the invisible mode, MULTI creates a multilink overlay network consisting of an IP-tunnel for each active interface. The probing module is responsible for establishing the tunnels between the client and the proxy, as well as sending probe packets in order to maintain the multilink overlay network, and performing NAT hole punching. Each probe packet contains a list of all available interfaces (the unique IDs calculated by link module). This information is used at the proxy to export a list of available tunnels to the MULTI proxy manager.

Present in proxy or client: Both. The client initiates the tunnels and sends the probes. The proxy updates its information based on the received information and replies to the probes, in order to keep the NAT hole open.

Options: The default interval between probe packets is 5 seconds. This can be changed during runtime by the MULTI client manager.

3.2.4 Tunneling module

The tunneling module is responsible for encapsulating/decapsulating and sending/receiving data from the tunnel(s). To send data, the manager calls a function in the tunneling module with the intended payload and desired tunnel as parameters. If the payload fits within the tunnel MTU, the payload is encapsulated and sent through the tunnel. If the size of the payload exceeds the MTU, an error code is returned to the manager. When data is received from a tunnel, it is decapsulated and the tunneling module calls a callback function implemented in the manager. This function is specified during the initialization.

An example of a callback function is a function which stores the received data in a buffer for further processing.

Present in proxy or client: Both.

Options: The callback function called when data is received.

3.3 Implementation details

MULTI mostly uses functionality that is the same in all OS'. For example, the tunnels and DHCP use normal network sockets that can be used together with standard, network function calls. However, how to detect changes in link state and configure the network subsystem differ between OS'. In this section, we will describe how this functionality was implemented in the Linux and Windows version. The BSD-implementation, which also works on OS X, is, except for a few minor details related to the used libraries, the same as Linux.

3.3.1 Linux/BSD

Our Linux MULTI-implementation was written in C. When changes in the network subsystem occur in Linux, the kernel broadcasts messages. These messages are divided into several groups and are available to userspace applications. In order to receive these messages, a client must create a NETLINK-socket and subscribe to one or more groups. Sending network configuration messages to the the kernel requires the use of the RTNETLINK [2]-library. MULTI's link module must be notified when new links are added or if a link has been removed/lost connection to its network. These messages are sent to the RTMGRP_LINK-group.

The information MULTI needs is contained in RTM_NEWLINK messages, and when such a message is received and after the link module has received an IP for the interface, RTNETLINK-messages are used to update the network configuration of the client. First, an RTM_NEWADDR message is sent to assign the IP address to the network interface. Then, RTM_NEWROUTE messages are sent to update the main routing table, as well as the private routing table for the current network interface. When multiple interfaces are active, Linux requires that each has its own routing table. Finally, a RTM_NEWRULE message is sent to create a rule stating which table shall be used for lookup when packets are sent to or received from the current interface's network. If an interface is no longer active, the opposite messages are sent (RTM_DELADDR, RTM_DELROUTE and RTM_DELRULE) and in the opposite order.

[2] http://www.kernel.org/doc/man-pages/online/pages/man7/rtnetlink.7.html

Linux provides several types of virtual interfaces and we used TUN/TAP [3]. TUN/TAP is robust, well-documented and enables applications to either receive the IP packet (TUN) or the entire Ethernet frame (TAP). When MULTI is used in invisible mode on Linux, the manager first has to create a TUN interface and assign it an IP address (an IP address can for example be requested from the proxy). Then, the client manager or users can add routes that go through it, and applications bind to the interface.

The only change between a Linux and a BSD-implementation of MULTI is that instead of a NETLINK-socket, the client must create a routing socket [4]. Also, even though we have used C, RTNETLINK and TUN can be used from every major programming language.

3.3.2 Windows

Windows supports host multihoming and automatically configures the routing tables when interfaces are added. In other words, the OS configures the system, and the Windows-version of the link module only needs to detect when the state of a network interface changes.

The MULTI-prototype for Windows was written in C#, Microsoft's own object-oriented language. In order to detect new network interfaces, the link module maintains a list of all available interfaces and listens for the NetworkAddressChanged-event. This event is triggered by the operating system when a network interface changes address, i.e., when it either becomes unavailable, switches network or has been configured.

After the event has triggered, the link module compares the most recent list of available network interfaces to its own list, and interfaces are either added or removed. The probing module is then notified and reacts accordingly. Finally, as TUN/TAP also exists for Windows, they were used as virtual interfaces.

3.4 Implementation examples

MULTI can be used in two different modes, visible and invisible. In this section, we will give a detailed example of how MULTI can be used in each mode.

Visible mode

Visible mode is intended for applications that will be extended with multilink support. Only the link and DHCP-modules are in use, and the framework notifies the application when a new interface is ready or when an interface has been removed/no longer is connected to a network.

[3] http://vtun.sourceforge.net/

[4] http://www.netbsd.org/docs/internals/en/chap-networking-core.html#netcore-sockets

In order to demonstrate MULTI's visible mode, we created a multilink enabled file downloader. Using the common Hyper Text Transfer Protocol (HTTP), a file was divided into several smaller logical segments. These segments were then requested over the available links to achieve bandwidth aggregation. The application supported changes in link state, and adapted which segments were requested over which links dynamically according to the available resources.

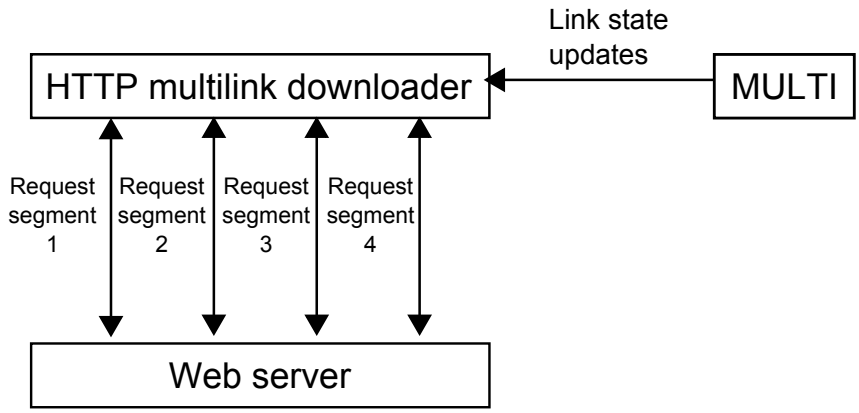


Figure 3.5: The architecture of the HTTP downloader used to demonstrate MULTI's visible mode.

The architecture of the file downloader is shown in figure 3.5, for a client device with four active links. Each link has been made responsible for one segment, and MULTI notifies the application of changes in link state. When notified of a new link, the manager created a socket, bound it to the corresponding interface, connected to the web server and requested data over it. If a link became unavailable, the remaining data would be requested over the first connection that was finished with its current segment.

To measure the performance of our HTTP multilink downloader, the client machine was connected to three 5 Mbit/s links, where only one link was available initially. The achieved aggregated bandwidth is shown in figure 3.6. As the transfer progressed, the two other interfaces were connected (at around 22 and 32 seconds) and the client was able to use close to 100 % of the available capacity, i.e., 15 Mbit/s.

Invisible mode

In many cases, it is not desirable or even possible to extend applications with multilink support, for example with closed source applications. When used to develop transparent multilink solutions, MULTI's invisible mode must be used. In invisible mode, MULTI makes use of a globally reachable proxy and creates a multilink overlay network consisting of IP-tunnels. In order to provide transparency, both MULTI Managers must create

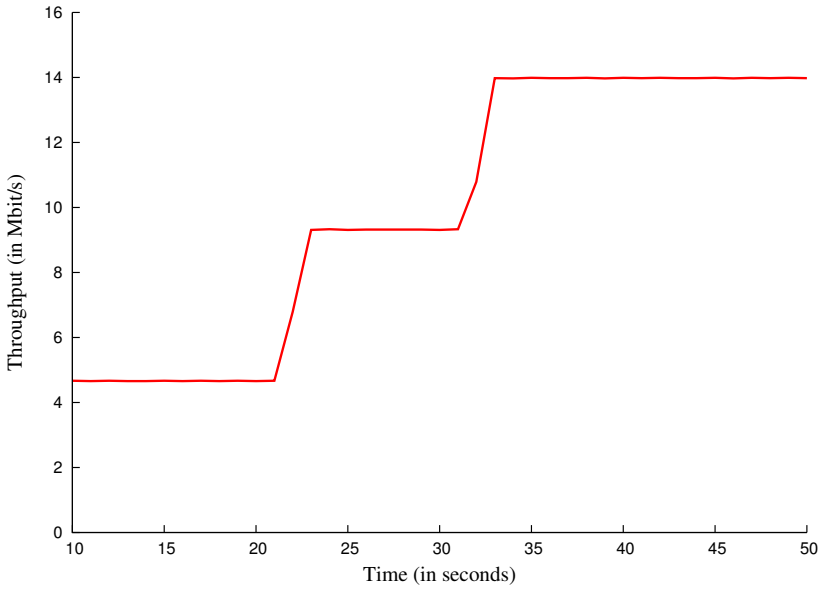


Figure 3.6: Achieved aggregated throughput with our MULTI example application (visible mode).

virtual interfaces. A virtual interface is perceived as a normal network interface by applications and is given its own IP-address. However, the interface is controlled by a userspace application and, in our case, all data flows through the MULTI managers. The managers are then responsible for sending the data over the network.

To demonstrate MULTI's invisible mode, we created a solution that performs transparent connection handover, based on the physical location of a device. Connection handover increases the reliability of network connections from a client and is especially useful in a roaming scenario. Other techniques that could be used to implement transparent connection handover include Mobile IPv6 [44] and Mobile SIP [76]. We would like to point out that this solution also supports bandwidth aggregation. However, as bandwidth aggregation was demonstrated for MULTI's visible mode, it will not be the focus here.

The system that was extended with support for transparent connection handover, was a location-aware quality-adaptive video streaming system [65]. The streaming client uses its current position to predict a path, and then plans for which video quality to request when based on information returned from a lookup service. The client queries the service using its GPS-coordinates, and the reply contains the capacity (bandwidth) of the available networks in the area. Earlier, the system has been limited to using one network. However, by applying transparent connection handover, the client device can switch to the network with the most available bandwidth without changing the application.

The architecture of our location-based, transparent handover application is shown in figure 3.7. To perform transparent handover, the roaming client (the MULTI client

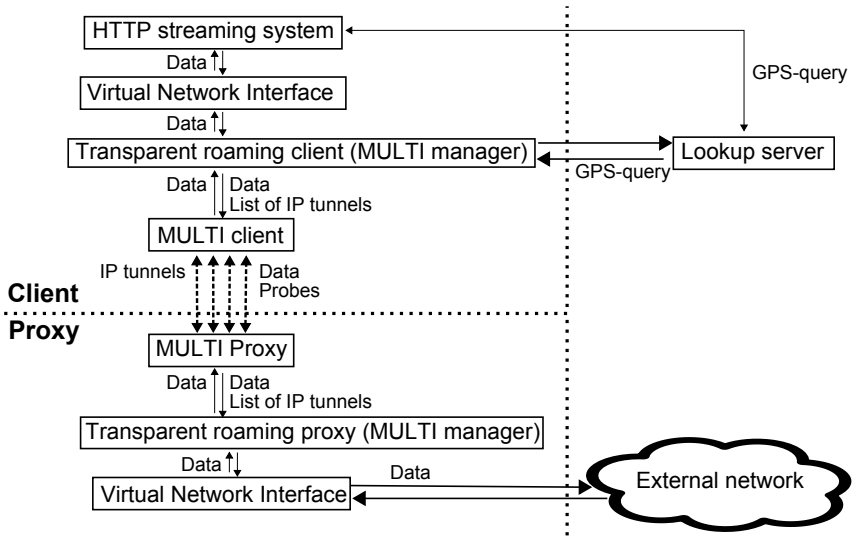


Figure 3.7: The architecture of the transparent handover solution.

manager) at given intervals, or when the connection to the current network is lost, queries the same lookup service as described in the previous paragraph. The MULTI Client manager selects the available link (tunnel) with the most available capacity. The ID of the selected tunnel is relayed to the roaming proxy (the MULTI proxy manager), and the handover is completed when the proxy updates which tunnel to use for downstream traffic. Because the "normal" applications on the multihomed client are bound to the virtual interface, they will not be affected by the handover and will proceed as normal. Thus, they do not need to be modified to support handover. The different components of our solution, as well as a more thorough set of evaluations, are presented in detail in [23].

In order to evaluate the performance of our solution, the video streaming client streamed a video from our webserver, using TCP. The experiments were performed on a tram commute route in Oslo, Norway (see figure 3.9) with 100 % 3G coverage. In addition, WLAN was available at the first station, as well as at a second station along the path. The WLANs always offered a higher bandwidth than the 3G network and was therefore used when possible.

An example of the achieved throughput, in addition to the amount of data downloaded over each interface, is shown in figure 3.8. Initially, a large amount of data was downloaded over the WLAN, at a much higher rate than the 3G network could support. As the tram moved away from the first station, the connection to the WLAN was lost and the connection transparently handed over to the 3G network (at around the 150 second mark). Without transparent handover, the connection would have been closed when the WLAN was no longer available. TCP connections are bound to one interface and fails if an interface becomes unavailable. When the WLAN became available again, after around

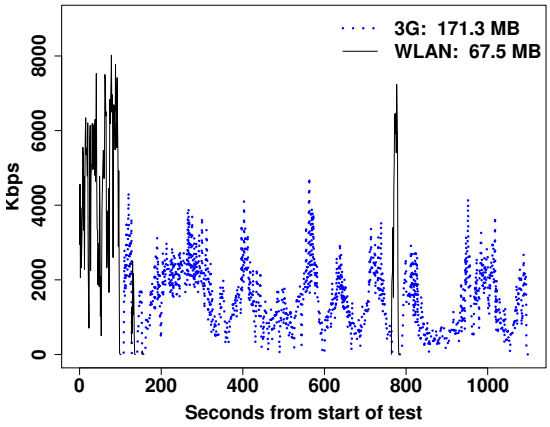


Figure 3.8: The achieved throughput and data downloaded over WLAN and 3G in our transparent handover experiment.

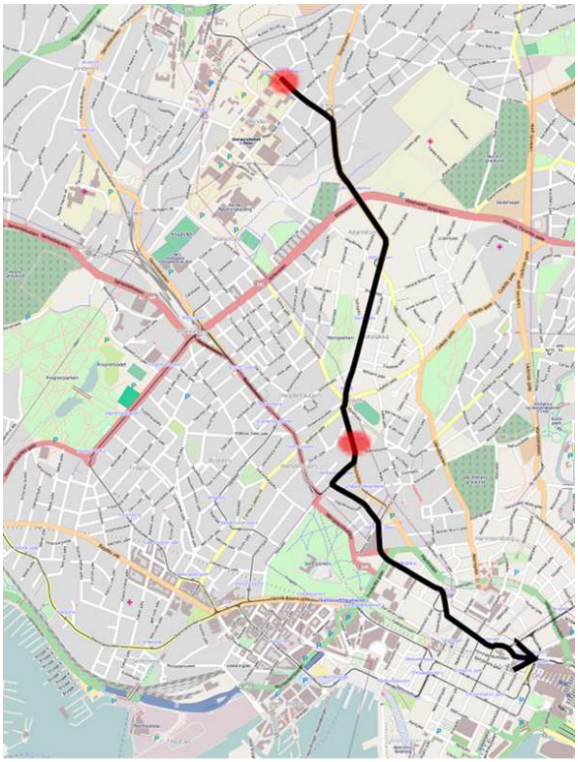


Figure 3.9: The tram route used to test the transparent handover solution. The two marked areas show where WLAN was available.

750 seconds, the connection was again handed over. However, the WLAN was only available for a few seconds, and the connection was handed back to the tunnel established oved the 3G link. 3G was then used for the rest of the journey. Because of the additional capacity offered by WLAN, the video quality increased significantly compared to when only 3G was used [23].

3.5 Summary

To address the shortcoming of earlier multilink solutions, we have designed and implemented MULTI. MULTI is a generic, platform-independent, modular framework for enabling multiple links, overcoming the deployment challenges described in section 1.2.1, and allowing for easier development and deployment of multilink solutions. It automatically detects changes in link state and configures the network subsystem, supports both application-specific and transparent multilink solutions and either notifies applications (called managers) of the changes in link state directly (visible mode), or updates its multilink overlay network first (invisible mode). In this section, we have introduced the design

of MULTI and the four core modules, given examples of how it can be used and described how MULTI has been implemented for Windows and Linux. All implementations that will be presented in the rest of this thesis used MULTI. In the next chapter, we look at application-specific, non-transparent bandwidth aggregation.

Chapter 4

Application-specific bandwidth aggregation

Several types of applications would benefit from having access to multiple links. For example, a video streaming application could request different segments over different links, a computer game could send the most important traffic over one link and use the others to support additional services (like voice), or a file downloader could use the additional links for redundancy. Implementing multilink support directly into the applications enables the creation of solutions that are tuned to a particular application, or application type.

Our application-specific bandwidth aggregation solution improves the performance of one of the most popular bandwidth intensive Internet services today - streaming of high quality video content. Video aggregation sites like YouTube [1] and Vimeo [2] serve millions of HD-videos every day, various events are broadcasted live over the Internet and large investments are made in video-on-demand services. One example is Hulu [3], which is backed by 225 content providers and allows users to legally stream popular TV-shows like Lost, House, Community and Grey's Anatomy. Techniques developed to enhance video streaming can be applied to other types of applications as well. For example, file transfer (bulk data transfer) also aims at transferring large amounts of data as fast as possible.

In order to provide easy deployment and interoperability with the existing server infrastructure and streaming solutions, we have implemented our techniques using the popular and widely supported Hypertext Transfer Protocol [24] (HTTP). However, we would like to point out that our techniques are not exclusive to HTTP, the streaming platform we have worked with or video streaming. The only requirement is that a client must be able to request independent parts of a file over different links, as shown in figure 4.1. Dividing

[1] http://www,youtube.com
[2] http://www.vimeo.com
[3] http://www.hulu.com/about

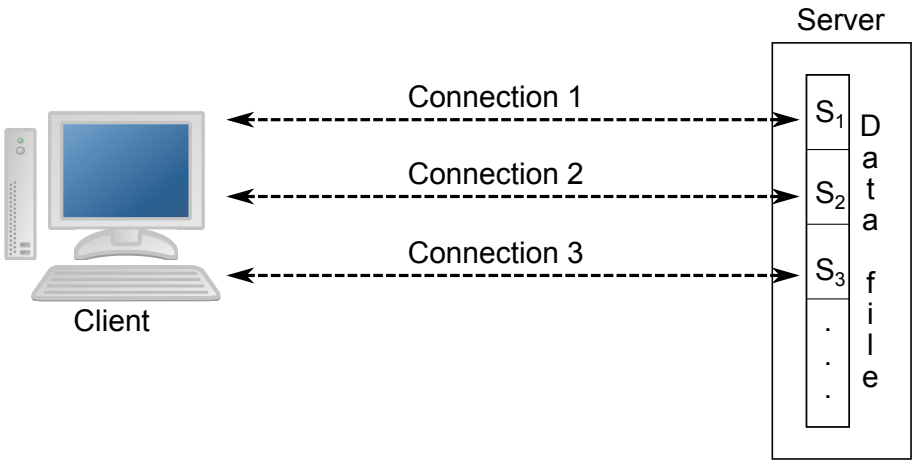


Figure 4.1: An example of application layer bandwidth aggregation. A multihomed client is connected to a server using three interfaces and requests one subsegment over each connection (S_1...,S_3), achieving bandwidth aggregation.

a file into independent, logical parts, from now on referred to as *subsegments*, allows a client to request the file in parallel over multiple links, thereby achieving bandwidth aggregation. Requesting subsegments is also supported by for example the commonly used File Transfer Protocol [62].

In this chapter, we introduce the HTTP protocol and present a generic technique for improving HTTP transfers using multiple links. Then, this technique is optimised for, applied to and evaluated together with quality-adaptive video streaming. Quality-adaptive video streaming allows clients to switch video quality while streaming and is supported by, amongst others, Microsoft (SmoothStreaming [78]), Apple (QuickTime Streaming Server [37]) and Move Networks [56].

4.1 HTTP and multiple links

The Hypertext Transfer Protocol is an application layer protocol that serves as the foundation of the World Wide Web. HTTP was introduced in 1991 [4] and is defined in RFC2616 [24] as a network protocol for distributed, collaborative hypermedia systems. The current version, 1.1, was released as an RFC [24] in 1999, and added several features that will be discussed in the next section. The protocol is designed as a request-response protocol for a client-server scenario. A client, for example a web browser, will first request a resource, for example a file, from the server. The server then replies with a response to the client's request. With HTTP, the response always starts with an HTTP header containing status information, before continuing with the requested content (if the request

[4] http://www.w3.org/Protocols/HTTP/AsImplemented.html

was successful). HTTP implicitly assumes that TCP is used as the transport protocol. However, this is not dictated by the RFC, and there are applications that use HTTP on top of UDP. Since the application-type we have focused on require a reliable connection and in-order delivery of data, all our applications combine HTTP with TCP connections.

When an HTTP client requests a resource, it starts by creating an HTTP GET message containing all relevant information. This message is then sent to the server. If the server is able to serve the request, an HTTP 200 OK header is sent to the client, together with the requested content. Otherwise, the server replies with a HTTP 404 Resource Not Found message. The request methods and status codes are defined by the protocol, and a pre-defined string is added to the end of each HTTP message. Using this string, the client can determine where in the message the content payload starts. The status information is only contained in the first packet of a request.

HTTP servers are developed to handle several connections simultaneously. Furthermore, HTTP supports, by default, three separate features which together make the protocol well suited for use together with multihomed clients. The features are:

- **Range retrieval requests** are required in order to divide a request for a file into multiple requests for smaller, independent parts (the subsegments). The subsegment requests will be distributed among the multiple available links. Achieving good bandwidth aggregation depends on requesting the right amount of data over each link, and we have created a formula for calculating this amount.

- **Pipelining** allows a client to send a new request before a response to the previous request has been received or processed at the server. It is used to remove the idle period caused by a client waiting for the first bytes of the next request to arrive. In order for pipelining to have an effect, the next request has to be sent early enough (in time). We have developed a technique for achieving efficient use of pipelining when requesting data over HTTP.

- **Persistent connections** enables reusing TCP connections, i.e., multiple requests can be sent over the same connection to the server. Having to open a new TCP connection for every request would incur a large time overhead, especially on links with a high RTT.

In the rest of this section, we will introduce these features and explain how we have used them to improve the performance of HTTP transfers when multiple links are available on the client.

4.1.1 Range retrieval requests

Range retrieval requests, or range requests, was introduced in HTTP/1.1 and allows a client to request specific byte-ranges of a given file. For example, if the first 50 kB of a 100 kB file is requested, only bytes 0 - 49 999 is sent. Range requests was introduced with the intention of supporting efficient recovery of partially failed transfers, as well as partial retrieval of large resources, and is commonly used by download managers. Download managers can for example resume interrupted transfers or improve the performance of a transfer by opening multiple connections to a server, and then request different parts of a file over each connection. However, to the best of our knowledge, no download manager supports using multiple links [77]. They all focus on single-link hosts with a high bandwidth connection to the Internet.

Range requests is the only feature that, together with server-side support for multiple simultaneous connections, is required in order to do bandwidth aggregation. Using range requests, a file is divided into logical subsegments. The subsegment-requests are then distributed among the different available links and requested in parallel.

Calculating the subsegment size

The subsegment size has a significant impact on throughput, both for a single link and for the achieved aggregated performance over multiple links. Finding the ideal subsegment size for a link is possible. However, it requires detailed knowledge about different link characteristics, such as the available bandwidth and the RTT. This information may not be readily available, accurate enough or require a significant time overhead caused by a probing period. Often, a trade-off has to be made between accuracy (in terms of how much data is allocated to one link) and performance. If an inaccurate amount of data is allocated to one link, it can have a negative effect on the performance and reduce the effectiveness of bandwidth aggregation.

The technique used for deciding how to divide a file into subsegments is known as the *subsegment approach*. Choosing a too small subsegment size will cause the server to process the request too quickly for the client to have time to pipeline another request, leading to idle periods at the server. Pipelining allows clients to make additional requests before the previous are finished and will be presented and discussed in more detail in the next subsection. A too small subsegment size will be referred to as the *minimum segment problem*, and the idle periods will occur if 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 send out a subsegment. In other words, the server will have finished processing the previous request before the next arrives.

An estimate of the transfer time is given by the subsegment size S and the average

throughput Φ_i over interface I_i:

$$t_i = S/\Phi_i \tag{4.1}$$

For pipelining to work efficiently over a path, a new request has to be sent from the client to the server at time t_{req} after the current time t_{now}:

$$t_{req} = t_{now} + S/\Phi_i - d_i \tag{4.2}$$

The minimum segment size S_{min} required for efficient pipelining, can be calculated by setting $t_{req} = 0$ and $t_{now} = 0$ and then solving equation 4.2 for the segment size S. This is the equivalent to the bandwidth-delay product of the path (the maximum amount of data the can be in transit on a link):

$$S_{min} = d_i * \Phi_i \tag{4.3}$$

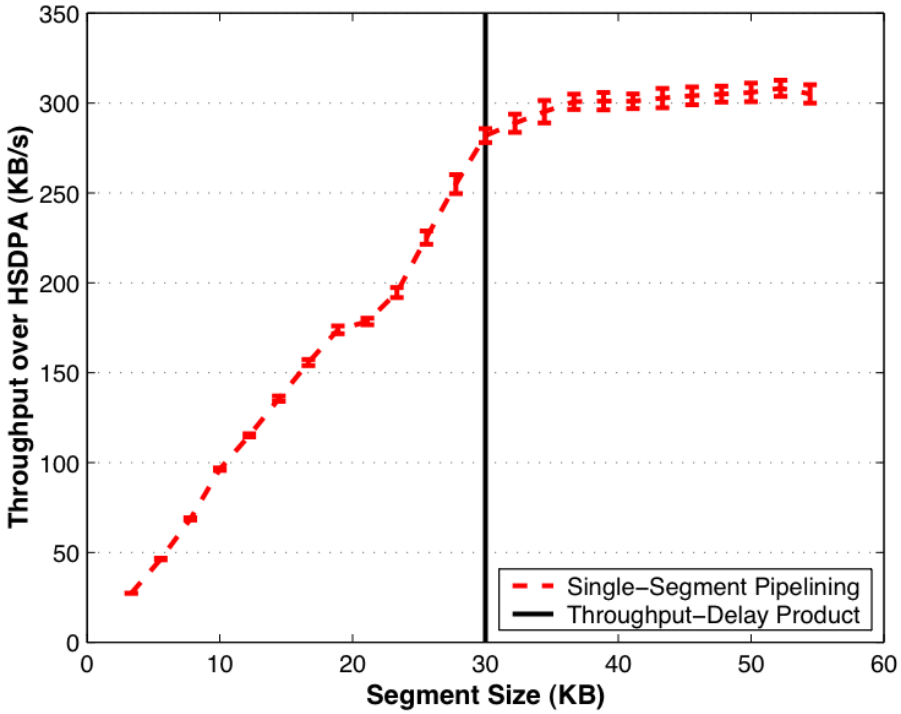


Figure 4.2: An example of the effect the subsegment size has on performance (the average experienced throughput of the HSDPA network was 300 KB/s).

An example of how different subsegment sizes affect performance is shown in figure 4.2. Here, data was requested over an HSDPA network with an average throughput of 300 KB/s and a one-way delay of 110 ms. When for example a subsegment size of 10 KB was used, the server required around 33 ms (equation 4.1) to send the entire subsegment.

However, 33 ms is significantly shorter than the average one-way delay. Equation 4.2 gives a t_{req} of -77 ms, in other words, the next subsegment must be requested $before$ the current subsegment's download has even started. The *minimum segment problem* has occurred. Using equation 4.3, we get an ideal S_{min} of 30 KB (110 ms*300 KB/s). For other combinations of bandwidth and delay, S_{min} would be different.

This is confirmed by figure 4.2. The throughput increased together with the subsegment size, and at almost precisely 30 KB, the throughput reached the expected value. The reason this did not occur at exactly 30 KB, is that the HSDPA link was not able to provide a stable throughput at exactly 300 KB/s. Increasing the subsegment size even further had very little effect, as the throughput was close to what the link could support for a subsegment size of 30 KB.

When subsegments are used together with multiple links, the subsegment size also has an upper bound. Choosing a too large subsegment size will lead to the *last segment problem*. This problem is caused by one or more links being allocated larger subsegment(s) than they can receive within a given time. Ideally, every subsegment that was requested at the same time, should also be finished at the same time. Thus, the sizes of the different subsegments should be set so that each link will spend an equal amount of time receiving data. However, if the subsegment size is set too large, the high bandwidth links will remain idle while waiting for the low bandwidth links to finish.

An example to illustrate this is as follows: Assume a client has two active links with different bandwidths. If we assume that a segment is divided into two equal sized subsegments, the high bandwidth link will finish receiving its subsegment first and remain idle for the rest of the transfer. Thus, the full capacity of both links will not be utilized and the bandwidth aggregation will be less efficient.

Additional byte overhead

Subsegments lead to a higher byte overhead because of the additional HTTP headers (i.e., HTTP requests and responses). This metadata noticeably reduces the throughput aggregation efficiency when the segmentation granularity approaches the size of a single IP packet. Each request/reply contains at least the name and location of the requested resource, the server's IP address and a byte-range specification. For normal length URLs, the byte overhead in each request/response is around 100 bytes. This is negligible compared to the minimum subsegment size for most realistic combinations of bandwidth and delay, which will need a subsegment size in the order of at least several kilobytes.

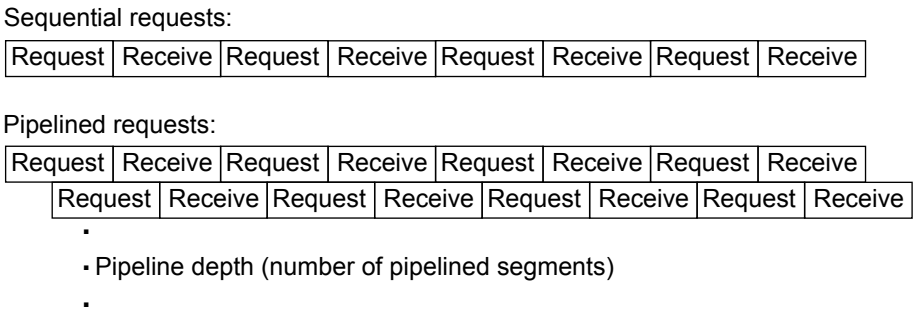


Figure 4.3: With sequential requests, the client has to wait until the previous request has been finished before it can make a new. This requirement is removed when pipelining is used and the next request(s) can be sent at any time, eliminating the time overhead.

4.1.2 Pipelining

HTTP pipelining is, according to the protocol specification [24], a method that "allows a client to make multiple HTTP requests without waiting for each response, allowing a single TCP connection to be used more efficiently, with much lower elapsed time". The benefit of pipelining is depicted in figure 4.3. Without pipelining ("Sequential requests"), each request has to be fully served before the next can be sent. Whenever a client makes a request, it has to wait at least one RTT before the first data arrives. A new request cannot be made before all the data belonging to the previous request has been received, introducing a large time overhead for high-RTT connections. For a large number of subsegments, this overhead significantly impairs the throughput.

Without pipelining, a subsegment approach will cause a reduction in throughput. The reduction depends on the number of subsegments, as well as the RTT of the link(s). First, a higher RTT causes an increase in the time it takes for the request and reply to arrive at the server and client, respectively. If we assume a one-way delay of 50 ms in both directions (a 100 ms RTT), it will take at least 50 ms before the server receives the request. Then, another 50 ms (or more) will pass before that client receives the first bytes. The client and server's idle time (while they wait for the next request) will increase together with the number of subsegments. If links are idle, they are not utilized at their maximum capacity, and, thus, the performance and effect of bandwidth aggregation is reduced.

An example of the performance gain that can be achieved with HTTP pipelining, combined with multiple links, is shown in figure 4.4. Here, the client was connected to one WLAN (average throughput 600 KB/s) and one HSDPA-network (average throughput 300 KB/s), and the aggregated throughput increased significantly compared to requesting subsegments sequentially, even for small values of S (the subsegment size). For example, with pipelining, the ideal throughput (900 KB/s) was reached with a subsegment size of

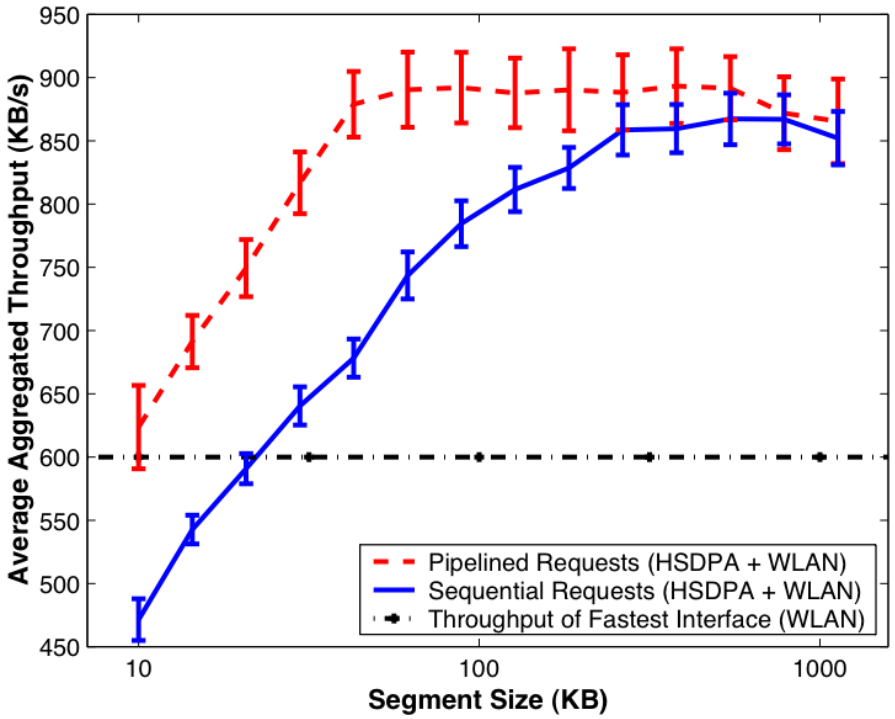


Figure 4.4: An example of the benefit of using HTTP request pipelining. These results were obtained by simultaneously downloading a 50 MB large file over HSDPA (avg. throughput 300 KB/s) and WLAN (avg. throughput 600 KB/s).

50 KB. The throughput when sequential requests was used never reached this level.

Multi-segment pipelining

In order to benefit from pipelining, it must be guaranteed that the amount of pipelined (not yet received) data exceeds a path's bandwidth-delay product. Otherwise, idle periods can occur because the client has received all the requested data before the next request has been processed by the server. This can be solved by either increasing the subsegment size or by pipelining multiple requests. Increasing the subsegment size is in many cases not ideal as it reduces the accuracy of the subsegment approach, in addition to the increased probability of causing the *last subsegment problem*. A better idea is therefore to increase the length of the pipeline. This means to increase the number of pipelined requests, known as the pipeline depth and symbolised by the dots in figure 4.3.

An example of the performance gain offered by pipelining multiple requests is shown in figure 4.5, using an HSDPA-connection with an average throughput of 300 KB/s. Even though the numbers are only valid for this connection, the observations are generic. While single-segment pipelining (a pipeline length of two) suffered from a low throughput for segment sizes below the bandwidth-delay product (the *minimum segment problem* occurs), multi-segment pipelining achieved close to optimal throughput with subsegment

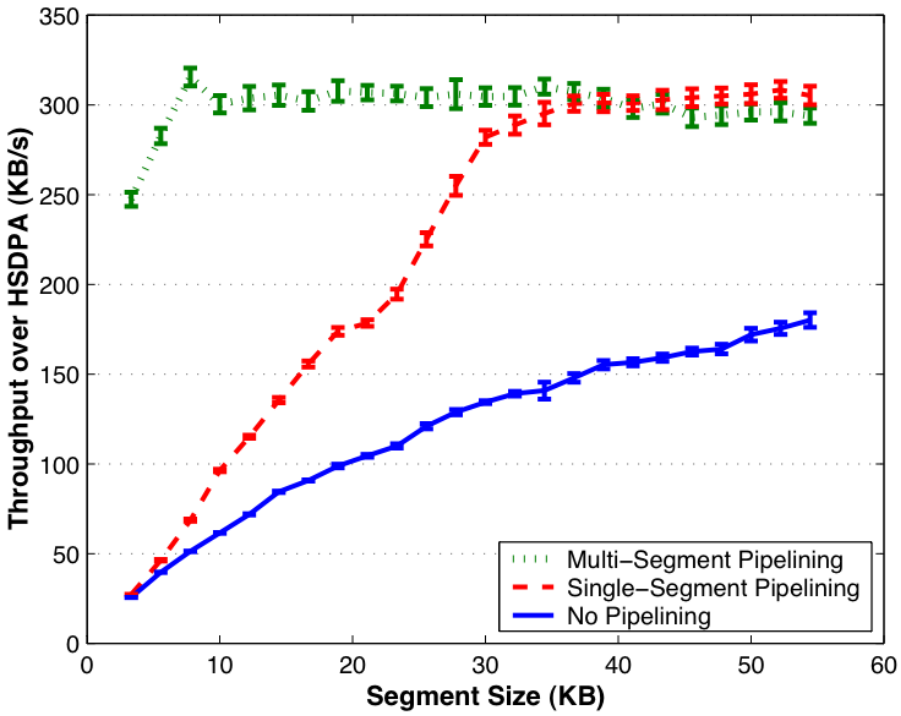


Figure 4.5: Increasing the number of pipelined subsegments results in a more efficient throughput aggregation.

sizes in the order of a few IP packets. The ideal length of the pipeline depends on both the subsegment size and the bandwidth delay product of the links. However, a long pipeline is not always ideal. Because HTTP does not support a client removing or aborting a request without closing down the connection, allocating (pipelining) too many subsegments to one link can have a negative effect on the throughput. For example, the available bandwidth changes frequently in a wireless scenario, and pipelining decisions might not be valid for very long. Restarting a connection involves a large time and processing overhead.

Assuming that an appropriate subsegment size has been chosen, a pipeline length of two (single-segment pipelining) is ideal. This reduces the probability of making a wrong pipelining decision, while still benefiting from pipelining. By requesting a new subsegment as soon as the first bytes of the previous subsegment arrives, the server will always have at least one request to process (pipelined).

Interleaved startup phase

Different link characteristics, like bandwidth and RTT, is not known in advance. Therefore, the technique used to allocated data to the links during the initial phase of a transfer, can have a significant effect on the performance of for example video streaming applications. If consecutive subsegments are requested over the same link, a longer waiting

time before the playback starts might occur. The application has to wait for gaps in the received data to be filled by requests made over a slower link(s).

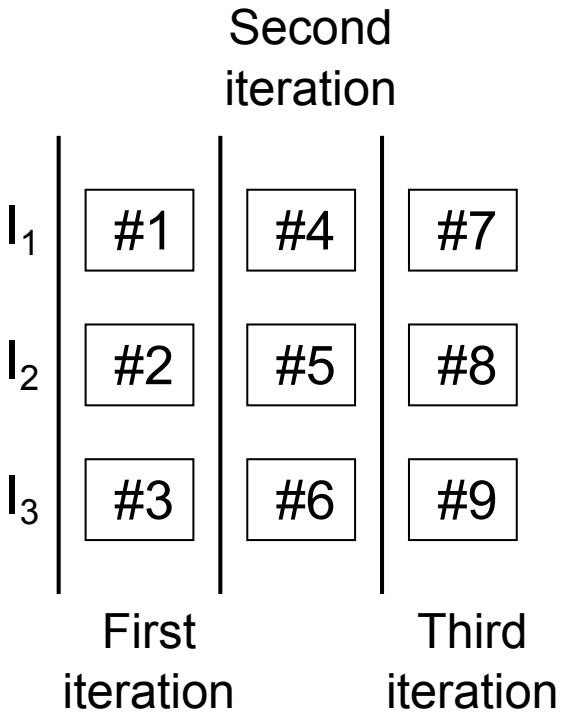


Figure 4.6: Startup phase – Requesting subsegments in an interleaved pattern helps to provide a smooth playback and reduces the initial response time. The figure shows interleaved startup for a client with three interfaces (I_1...,I_3) and a pipeline depth of three.

In order to avoid consecutive subsegments requested over the same link, we recommend using a pre-defined scheduling pattern during the startup phase (see figure 4.6). During the interleaved startup phase, it is guaranteed that consecutive subsegments are not assigned to the same interface. After the interleaved startup phase, subsegments will automatically be requested based on the performance of the link(s). A faster interface will finish receiving subsegments faster and, thus, request more subsegments than slower links (assuming pipelining is enabled).

4.1.3 Persistent connections

The final feature that makes HTTP well-suited for use with multilink networked applications, is persistent connections. In HTTP versions prior to 1.1, a client could only send one request per connection. If for example a web browser were to retrieve a webpage containing 10 pictures, 11 connections would have to be opened (one for retrieving the webpage and ten for the pictures).

Having to open a new connection for every request can introduce a significant time overhead, especially on links with a high RTT. Persistent connections removes this over-

head by allowing multiple requests to be sent over one connection. The server then uses a timeout to determine when a connection should be closed (unless done so by the client).

4.2 Quality-adaptive streaming over multiple links

Video streaming is a generic term that refers to several different types of streaming. The most common type of streaming on the Internet is *progressive download*, which allows a stream to be played out while it is being downloaded. A particular property of progressive download is the use of client-side buffering for achieving smooth playback during periods of congestion. The buffer is also used to compensate for the difference between the required bitrate of the video and the available bandwidth. If the buffer is emptied too fast, the playback pauses until sufficient data is available again. In order to avoid emptying the buffer, the buffer is initially filled up to a certain point. Then, a video streaming client will aim to keep the buffer sufficiently full at all times. From a user's perspective, the required buffer size (in terms of bytes required in memory) is of little relevance. However, when developing multilink streaming solutions on devices with limited memory, the maximum possible buffer represents an important design factor. We assume that the devices always have enough available memory to buffer the requested data. Still, our goal has been to achieve the best possible video playback quality with the smallest possible buffer.

4.2.1 Startup delay

The initial filling of the buffer is known as the startup delay or latency, and is an important factor for the user-perceived quality of service. For example, when starting a full-length movie of two hours, users might be willing to wait a few minutes before playback starts. However, for shorter clips, the startup latency should be perceived as instantaneous. Also, the startup latency affects the *liveness* of a stream. We define liveness as how far behind the playback is the no-delay broadcast. Liveness is used as a parameter in our evaluations, and we try to achieve the best playback with the shortest startup delay (and thus, highest degree of liveness).

4.2.2 Adaptive streaming

Quality-adaptive, segmented video streaming is a more advanced type of progressive download. Earlier, streams were typically encoded in a single bitrate and the client was unable to adapt the stream to the available resources, except for increasing the buffer size, startup delay and rebuffering periods. With quality-adaptive streaming, each video is divided into fixed-length segments, and each segment is encoded at different bitrates. The clients are

given more control and can for example adjust the requested video quality based on the measured throughput, as well as the number of segments currently in the buffer.

By requesting different parts of the video file or video segments over different links, a device should ideally be able to utilize the full capacity of the different links and achieve better performance than the fastest of the single links (for example a higher video quality with fever playback interruptions). Also, using several independent interfaces makes a solution more robust against link variance, congestion and failure.

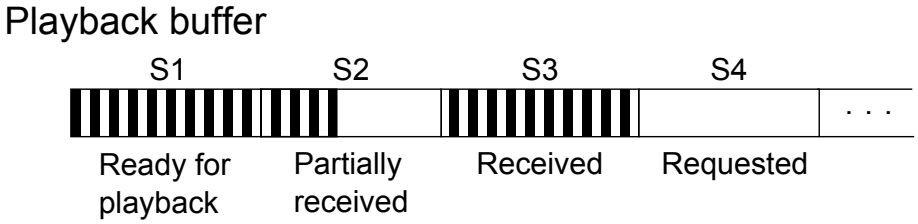


Figure 4.7: Snapshot of a stream in progress. The striped areas represent received data.

However, different techniques are needed to overcome the link heterogeneity challenges described in section 1.2.2. Due to differences in bandwidth and delay, there can be gaps in the received content, as shown in figure 4.7. Video is read sequentially, so any gap must be filled before segments can be played back. If a segment is not ready before it is to be played out, a *deadline miss* occurs. A deadline miss causes interruptions in playback, as the client has to wait for the rest of segment, and reduces liveness even further. If the stream is a live broadcast, each deadline miss causes the client to *lag* further behind the broadcast. One technique used to increase liveness, is to *skip* segments if the client lags too far behind. This will be discussed later.

4.2.3 The DAVVI streaming system

In order to design, develop and evaluate HTTP-based quality adaptive multilink streaming in a real-world environment and for real-world networks, a video streaming system is needed. There exists several different such systems, however, except for one, those we found were all proprietary.

Because it is open-source and provides the features we need, we extended the DAVVI-platform [43]. DAVVI is an HTTP-based streaming system that supports several types of streaming, and offers several of the same features as popular commercial, proprietary solutions (like Microsoft's SmoothStreaming [78]). Each video is divided into fixed-length, independent video segments with constant duration (set to two seconds for the work done related to this thesis). The segments are encoded in multiple qualities (bitrates) and each segment (in each quality) is stored as a separate file. The constant duration of the segments limits the liveness of a stream, i.e., how live a stream can be compared to the

broadcast. At least one segments must be ready and received by the client before playback can start.

DAVVI stores video segments on regular web servers, so a dedicated streaming server is not needed, and the video segments are retrieved using HTTP GET. Because no additional feedback is provided by the server, the client monitors the available resources. This information is used to adapt the video quality and ensure that the buffer is always as full as possible, ideally avoiding deadline misses. The quality can be changed whenever a new segment is requested. However, the change will not be visible to the user before the segment is actually played out. In our case, each segment contains two seconds of video, which has been shown in [57] to be a good segment length. According to their work, changing video quality more frequently than every 1-2 seconds annoys the user.

For this thesis, we have focused on the three main types of streaming. They are as follows:

- **On-demand streaming:** On-demand streaming is used together with full-length movies and similar content, meaning that video quality and continuous playback are the most important metrics. This is the most common type of streaming, and the performance is only limited by the available bandwidth. Because the entire video is available on the server in advance, segments can be requested as soon as there is room in the receive buffer.

- **Live streaming with buffering:** Live streaming with buffering is very similar to on-demand streaming, except that the whole video is not available when the streaming starts. Live in the context of this thesis is liveness, and by delaying playback by a given number of segments (the startup delay), a trade-off between liveness and smoothness is made. Provided that all requested segments are received before their playout deadline, the total delay compared to the broadcast is *startup_delay + inital_segments_transfer_time*. Live streaming with buffering is frequently used to stream sports events, for example cross country or road cycling.

- **Live streaming without buffering:** Live streaming without buffering has liveness as the most important metric. Segments (requests) are skipped if the stream (playback) lags too far behind the broadcast, and a requirement for being as live as possible is that the startup delay is the lowest that is allowed by the streaming system. In our case, this limit is two seconds (one segment), so the client lags *2s + initial_segment_transfer_time* behind the broadcast when playback starts, and skips segments if the lag exceeds the length of one segment. Even though most content provides use a startup delay of some segments, we have also looked at live streaming without buffering as it is the most extreme case and provides valuable knowledge.

In order to better utilise available network resources, DAVVI was extended with multilink support through MULTI. MULTI was used in visible mode, meaning that the application was notified directly of changes in link state. Whenever the client was connected to a new network, the DAVVI client application created a new connection to the video server and started requesting subsegments. In order to use multiple links efficiently, subsegments must be requested according to the available resources. If a slow interface is allocated a large share of a segment, the performance of the whole application might suffer. For example, the segment may not be ready when it is supposed to be played out, causing a deadline miss and an interruption in playback.

The core of DAVVI's multilink extension is the *request scheduler*. The request scheduler is responsible for making and distributing subsegment requests, adjusting the desired video quality and forwarding complete segments to the video player. The size of each subsegment is decided by the *subsegment approach*. Without a good scheduler and a good subsegment approach, adding multiple interfaces can cause a drop in performance, affecting the user experience. For example, the quality adaptation might be too optimistic and select a higher quality than the links can support, or links might not be used to their full capacity. The request scheduler and how it adapts quality is presented in the next section. Then, in the subsequent sections, we combine and evaluate the scheduler together with two different subsegment approaches. In the *static subsegment approach*, each subsegment has a fixed size, while the *dynamic subsegment approach* divides segments into subsegments based on the measured throughput.

4.2.4 Quality adaption mechanism and request scheduler

In order to utilize the available client interfaces efficiently, to avoid video deadline misses and to prove the best possible user experience, we extended DAVVI with a new request scheduler. The client monitors the throughput (both aggregated and per link), and the request scheduler uses this information to adjust the video quality level and request subsegments over the available interfaces.

Our request scheduler is outlined in algorithm 1. The first segment is always requested in the highest quality offered by DAVVI (line 1-3). The highest quality level consists of the largest files, i.e., the files that will be divided into the most subsegments ("samples"). This enables the client to quickly make the most accurate aggregated throughput estimate, at the expense of a longer startup delay. Also, the first segment is the only segment that can not cause a deadline miss, so it is safe to request it in any quality. The aggregated throughput estimate is updated once for every subsegment (line 9), and is estimated by the client measuring how long it takes to receive the subsegments over the different links. The measurements are averaged and then added together.

Algorithm 1 Request Scheduler *[simplified]*

```
1:  quality_level = "super"
2:  request(subsegment over each interface)
3:  request("pipelined" subsegment over each interface)
4:  while (stream not finished) do
5:     data = receive()
6:     I = interface that received data
7:     if (data == complete subsegment) then
8:        estimate aggregated throughput
9:        if (data == complete segment) then
10:          queue_for_playback(segment)
11:          quality_level = quality_adaption_mechanism
12:       end if
13:    end if
14:    if subsegment's pipelined_subsegment then
15:       request(subsegment, I, quality_level)
16:       subsegment's pipelined_subsegment = true
17:    end if
18: end while
```

The request scheduler uses interleaved startup (line 2 and 3), and as long as there are video segments available and space in the buffer, the client will continue to request subsegments. If the subsegment is the first subsegment of the next video segment, the quality is adjusted (line 11) based on the aggregated throughput. Pipelining is done as soon as possible (line 14-17) in order to ensure that the server always has one segment to process. Each subsegment has a *pipelined_subsegment* variable, which is used to determine if the current subsegment has spawned a pipelined subsegment request or not. Algorithm 1 assumes that a pipeline length of two is used, otherwise, *pipeline_subsegments* would be replaced with a counter.

If there are no more segments available (for example if the client has caught up with a live-stream) or the buffer is full, the request scheduler will wait before it requests more data. When a segment is then available, the algorithm will restart, i.e., interleaved startup will be used. However, unlike for the initial segment, which is always requested in the highest quality, the measured throughput is used to select the quality.

The quality adaption mechanism is summarized in algorithm 2. Three parameters are used to select the quality: the amount of buffered data ($num\_buffered\_segments$), the measured aggregated throughput ($aggregated\_throughput$) and the size of the different quality levels of a segment (in bytes, $segment\_size[quality\_level]$). The size of the video segment is parsed from a continuously updated logfile generated by the server. Requesting and receiving the updated parts of this file adds some overhead, however, measurements has shown that it does not have an effect on the overall performance. The requests are

Algorithm 2 Quality adaptation mechanism

```
1: transfer_deadline = time_left_playout + (2s * num_buffered_segments)
2: pipeline_delay = pipelined_bytes_left / aggregated_throughput
3: for quality_level = "super" to "low" do
4:    transfer_time = segment_size[quality_level] / aggregated_throughput
5:    if transfer_time <(transfer_deadline - pipeline_delay) then
6:       return quality_level
7:    end if
8:    reduce quality_level
9: end for
```

made in parallel with the requests for subsegments, and each update is less than one kilobyte large.

The goal of the quality adaption mechanism is to find the highest possible quality that will not cause a playback interruption. When selecting video quality, the client first calculates how much content it has already received and that is ready for playout ($transfer\_deadline$, line 1). This is the sum of what is left of the segment that is currently being played out ($time\_left\_playout$), and how much content has been buffered (2 seconds * $num\_buffered\_segments$. Then, an estimate for how long it will take to receive the data that has already been requested ($pipeline\_delay$, line 2) is calculated.

For each quality level, starting from the highest, the time it will take to receive the segment in the given quality level is calculated ($transfer\_time$, line 4). By subtracting the $pipeline\_delay$ from the $transfer\_deadline$, an estimate of how much additional time can be spent receiving data without causing a deadline miss is obtained (line 5). If $transfer\_time$ is less than this estimate, the segment should be received and ready for playback before a deadline miss occurs, and the quality level is returned to the scheduler (line 6).

4.3 Evaluation method

When evaluating the performance of video streaming, we measure the video quality and deadline misses. The video quality is dependent on the bandwidth aggregation, i.e., an efficient aggregation results in a higher throughput. Thus, the quality increases. Deadline misses are of the highest importance from a user's perspective, with respect to the perceived video quality. The number of deadline misses depends on the subsegment approach. A poor approach allocates too much data to slower interfaces, causing data to arrive late and segments to miss their deadlines.

Initially, we developed the static subsegment approach and evaluated the performance difference between using a single and two links. The subsegment approach was combined with live streaming with buffering, and we measured the effect of both bandwidth and latency heterogeneity. If the heterogeneity is not properly considered, adding multiple links can lead to a reduction in throughput. However, as we discovered, the performance of the static subsegment approach depends on the buffer being large enough to compensate for the bandwidth heterogeneity. Increasing the buffer is not always ideal, so the static subsegment approach was improved by making it more dynamic. In our second set of experiments, we compared the performance of the static and dynamic subsegment approaches, and evaluated them together with all the three types of streaming.

The experiments were performed both in a fully controlled environment and in real-world networks. A fully controlled environment allows us to isolate and analyze the effect of different parameters, while real-world networks give an impression of how the multilink streaming would perform if deployed. To get comparable results from real-world networks, the tests were run interleaved, and the experiments were performed during peak hours (08-16) to get the most realistic network conditions.

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

Table 4.1: Quality levels and bitrates of the soccer movie used to evaluate the performance of video streaming.

The same video clip was used in all the experiments. The clip was Variable Bitrate-encoded (VBR) and shows a football match. It has a total playout duration of 100 minutes (3127 segments of two seconds) and was available in four different qualities. A subset of 100 video segments were used in the evaluations, and the bandwidth requirements for this subset is shown in table 4.1. In all the experiments performed in the controlled network environment, the sum of the available link bandwidth was always equal to the average requirement for super quality, 3 Mbit/s. In other words, the client should on average be able to stream video in the highest quality. Limiting the bandwidth also allows us to evaluate the gains of bandwidth aggregation, otherwise, a single link would in many of the evaluations have been sufficient for receiving the highest quality.

For each experiment, the buffer size and startup delay were always equal, forcing the client to fill up the buffer before starting playback.

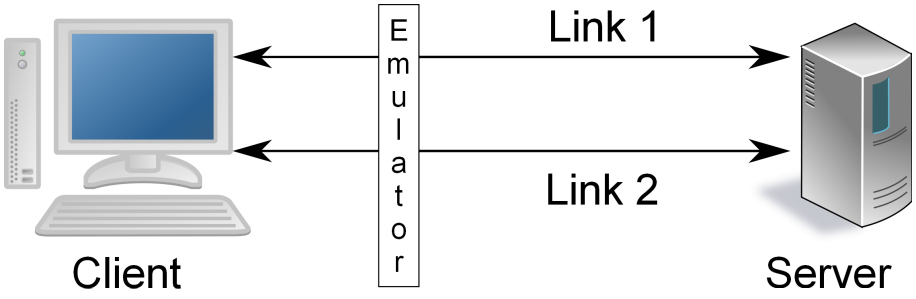


Figure 4.8: The controlled environment-testbed used to evaluate the performance of video streaming over multiple links.

4.3.1 Controlled network environment

The controlled environment testbed, shown in figure 4.8, consisted of a client and a server (Apache 2 [5]) connected using two independent 100 Mbit/s Ethernet links. Both client and server ran Linux 2.6.31, and to control the different link characteristics, the network emulator *netem* was used with a hierarchical token bucket queueing discipline.

In the controlled network environment, we measured how different levels of bandwidth and latency heterogeneity affected the video streaming. In addition, we performed experiments where we emulated link dynamics. This allowed us to expose the subsegment approaches to dynamic links, while still having some control over the parameters.

Bandwidth heterogeneity

Bandwidth heterogeneity is one of the two challenges that are especially important when aggregating multiple links. Unless the traffic is balanced properly, the links will not be used at their full capacity.

In order to evaluate the effect of bandwidth heterogeneity, the combined bandwidth of the two links was always 3 Mbit/s, emulated using the hierarchical token bucket. This is equal to the average bandwidth requirement for the highest video quality (see table 4.1). The latency was that of the links, i.e., it was homogeneous.

Latency heterogeneity

Different types of networks often have significantly different latencies, causing requests to arrive and be processed at different times. When measuring the effect of latency heterogeneity on video quality and deadline misses, one link had a constant RTT of 10 ms, while the other link was assigned an RTT of r ms, with $r \in \{10, 20, \ldots, 100\}$. The bandwidth of each link was limited to 1.5 Mbit/s, to avoid bandwidth heterogeneity affecting the results.

[5] http://www.apache.org

Emulated link dynamics

Dynamic links impose different challenges than static links do, as the scheduler has to adapt to often rapid changes in the network. To expose the two subsegment approaches to dynamic links while still having some control over the parameters, we created a script which adjusts the link characteristics based on observed real-world network behavior. The purpose of this script is not to recreate reality, but to give an impression of real-world link behavior.

The sum of the bandwidth of the two links was always 3 Mbit/s, but at random intervals of t seconds, $t \in \{2, \ldots, 10\}$, the bandwidth bw Mbit/s, $bw \in \{0.5, \ldots, 2.5\}$ of each link was updated. The RTT of link 1 was normally distributed between 0 ms and 20 ms, while the RTT of link 2 was uniformly distributed between 20 ms and 80 ms. The script was used in the experiments conducted to gather the results presented in section 4.4.2 and 4.5.3.

4.3.2 Real-world networks

	WLAN	HSDPA
Average experienced throughput	600 KB/s	250 KB/s
Average RTT	30 ms	220 ms

Table 4.2: Observed characteristics of the real-world links that were used when comparing the static subsegment approach to using a single link.

	WLAN	HSDPA
Average experienced throughput	287 KB/s	167 KB/s
Average RTT	30 ms	220 ms

Table 4.3: Observed characteristics of the real-world links that were used when comparing the performance of the static and dynamic subsegment approach.

Measuring the performance of the subsegment approaches in real-world networks, gives an impression of how they will perform if deployed. We made experiments in a wireless scenario where the client was connected to a public WLAN (IEEE 802.11b) and an HSDPA network. The characteristics of these networks are summarized in table 4.2. The reason we worked with wireless networks is that they present a more challenging environment than fixed links. When comparing the static to the dynamic subsegment approach, the characteristics of the wireless networks had changed. The performance of the wireless networks when performing the second set of evaluations is summarized in table 4.3. That the performance of the networks had changed means that the results cannot be compared directly. However, any general trends and observations are still valid.

4.4 Static subsegment approach

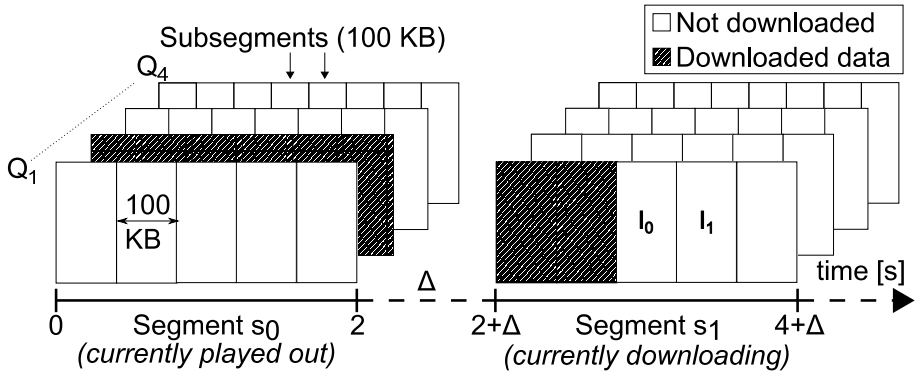


Figure 4.9: An example of the static subsegment approach. The two interfaces I_0 and I_1 have finished downloading segment s_0 of quality Q_2. As the throughput has dropped, they currently collaborate on downloading a lower-quality segment.

The first of the two subsegment approaches, the static subsegment approach, divides each two-second video segment into fixed-sized, logical subsegments, as shown in figure 4.9. In the figure, a client equipped with two interfaces will request a new video segment. This segment is divided into fixed-size subsegments that are requested interleaved over the two interfaces. The performance of the static subsegment approach was measured together with live streaming with buffering and was compared to that of the single, faster link. We did not want to add additional time overhead by measuring the capacity of each link and then calculating a subsegment size, so a size of 100 KB was used. Based on experimental evaluation and the formulas for calculating the ideal subsegment size, 100 KB was found to give a good trade-off between performance and accuracy (in terms of how much data is allocated to one interface). It provides sufficient slack and is large enough to not cause the minimum segment problem, while still being small enough to avoid the last segment problem for any link used in the evaluations. Even the slowest link is able to receive 100 KB within a few seconds.

4.4.1 Static links

The emulation testbed introduced in section 4.3.1 provides a static, fully controllable network environment, where it is possible to isolate, adjust and analyze different parameters. The purpose of the experiments with static links was to evaluate how bandwidth and latency heterogeneity affects the video quality. The total available bandwidth was always equal to 3 Mbit/s.

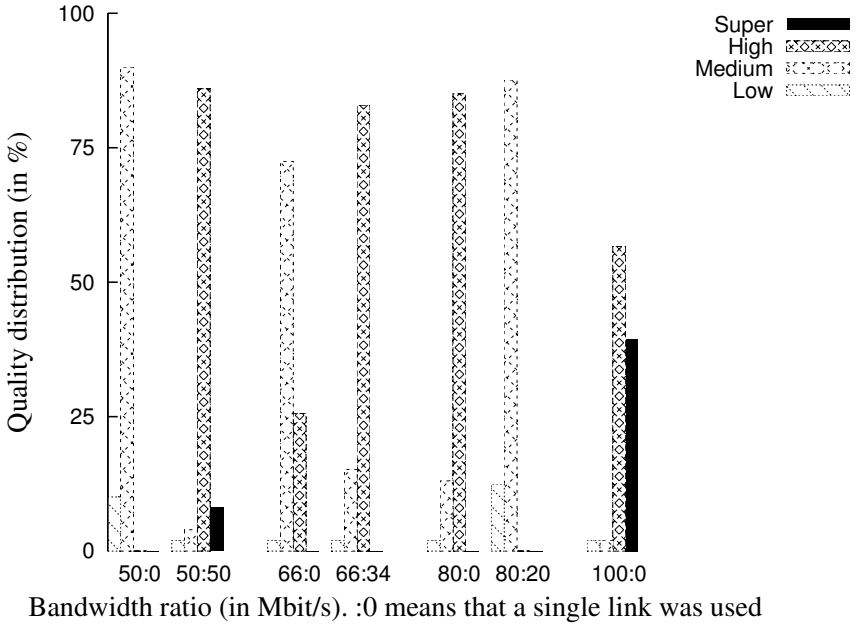


Figure 4.10: Video quality distribution when the scheduler was faced with bandwidth heterogeneity in a fully controlled environment (0.1 ms RTT on both links), using the static subsegment approach.

Bandwidth heterogeneity

Figure 4.10 shows the video quality distribution for various levels of bandwidth heterogeneity. The bandwidth ratio is shown along the x-axis, and the X:Y notation means that one link was allocated X % of the bandwidth, while the other link was allocated Y %. The bars represents the four video qualities, and the y-value of each bar is its share of the received segments.

In this experiment, a buffer size of two segments was used, meaning that the client lagged at least 4 seconds behind the live stream when the video started playing. With a single link (the X:0 bars), the quality, as expected, increased as more bandwidth became available. Adding a second link increased the quality even further, however, only for some levels of bandwidth heterogeneity. When the bandwidth ratio was 80:20, the performance was worse than when a single link was used (80:0).

This was caused by the buffer being too small to compensate for the link heterogeneity, and the client was unable to reach maximum performance. With a short startup delay and small buffer, the request scheduler is only allowed a little slack when requesting the first segment after the playout has started. Assuming that the links are heterogeneous and none exceeds the bandwidth requirement for the stream by a large margin, the small amount of buffered content forces the scheduler to pick a segment of lower quality. Smaller segments consist of fewer subsegments, so the slowest link is allocated a larger share of the

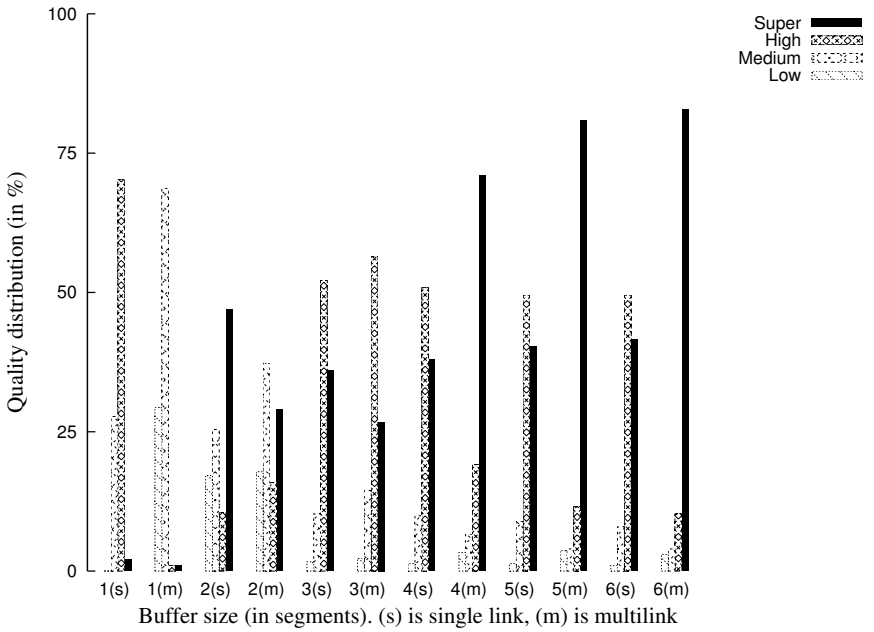


Figure 4.11: The number of buffered segments plotted against video quality distribution (bandwidth ratio 80:20), using the static subsegment approach.

requested data, and the link has a more significant effect on the throughput measurements. This quality reduction continues until the throughput and quality stabilizes at a lower level than the links might support.

Increasing the receive buffer size and startup delay improves the situation, as can be seen in figure 4.11. This figure shows how a larger buffer size increased the efficiency of throughput aggregation for a high bandwidth ratio (80:20). A larger receive buffer allows the scheduler more slack, so the first segment after the startup delay (the buffer has been filled completely) is requested in a higher quality than with a small buffer. Larger segments consist of a higher number of subsegments than smaller ones, so fewer subsegments are requested over the slowest interface, i.e., it is made responsible for less data (relative to the total amount of requested data). Provided that the buffer is large enough, the links are allocated their correct share of subsegments (or at least close to). Thus, the throughput measurements are more accurate and a better video quality distribution is achieved.

A rule of thumb with the static subsegment approach is that the buffer size must be equal to the bandwidth ratio. For example, a bandwidth ratio of 80:20, requires a buffer size of five segments (the low bandwidth link can receive one segment for every fourth the hight bandwidth one receives). However, this is only correct for a video with constant bit ratio. With a VBR-video, which was used in the experiments presented in this chapter, the segments are of different sizes and have different bandwidth requirements. The latter explains why a buffer size of four was sufficient for the multilink performance to exceed that of a single link, as seen in figure 4.11.

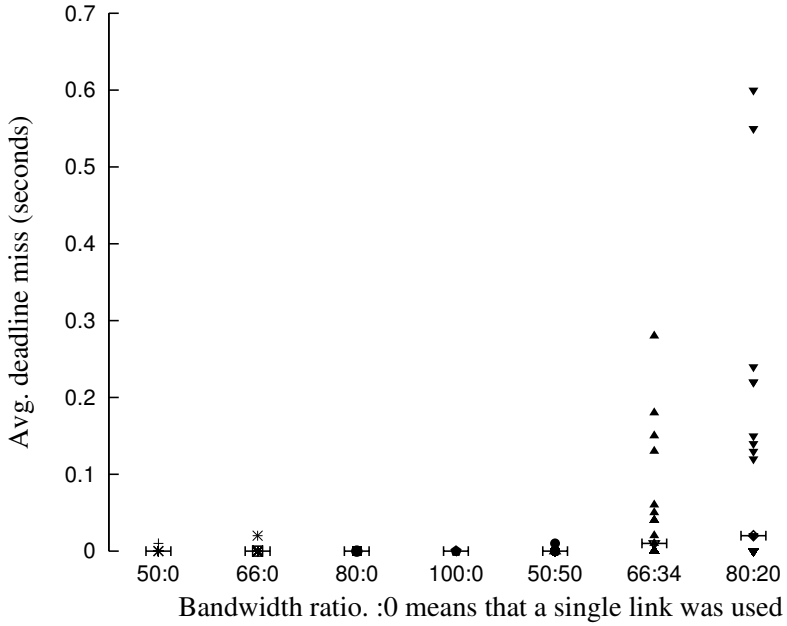


Figure 4.12: Deadline misses for 2-segment buffers and various levels of bandwidth heterogeneity, using the static subsegment approach.

Another interesting observation made in figure 4.11, is that limiting the buffer size also affects the single link performance. Even though the connections to the server were kept alive (persistent connections), the cost of having to wait for room in the buffer and then request a new segment is high.

Figure 4.12 shows the deadline misses for the different bandwidth ratios. The RTT was close to constant at 0.1 ms, and the majority of deadline misses occurred when the buffer was not large enough to compensate for the bandwidth heterogeneity. Thus, another consequence of the buffer being to small, is that the slow interface caused the reception of the complete video segment to be delayed.

Latency heterogeneity

For the results presented in figure 4.13, the latency heterogeneity was varied between different levels. The bandwidth of each link was limited to 1.5 Mbit/s, to avoid bandwidth heterogeneity affecting the results, and the buffer size was set to two segments.

As depicted in Figure 4.13, video quality was not significantly affected by latency heterogeneity. Through efficient use of pipelining and adding a startup latency, the latency heterogeneity was compensated for.

However, not taking the latency heterogeneity into account can have an effect on deadline misses, as seen in figure 4.14. The buffer size limits the frequency of subsegment requests, and after an interface has been idle, it will take at least one RTT before the

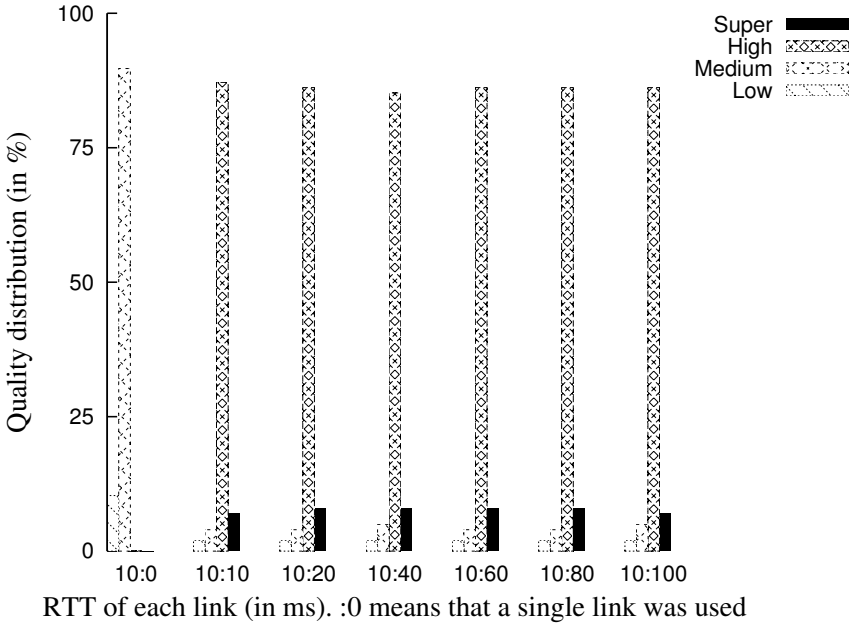


Figure 4.13: Video quality distribution when the scheduler was faced with latency heterogeneity in a fully controlled environment, buffer size of two segments and using the static subsegment approach.

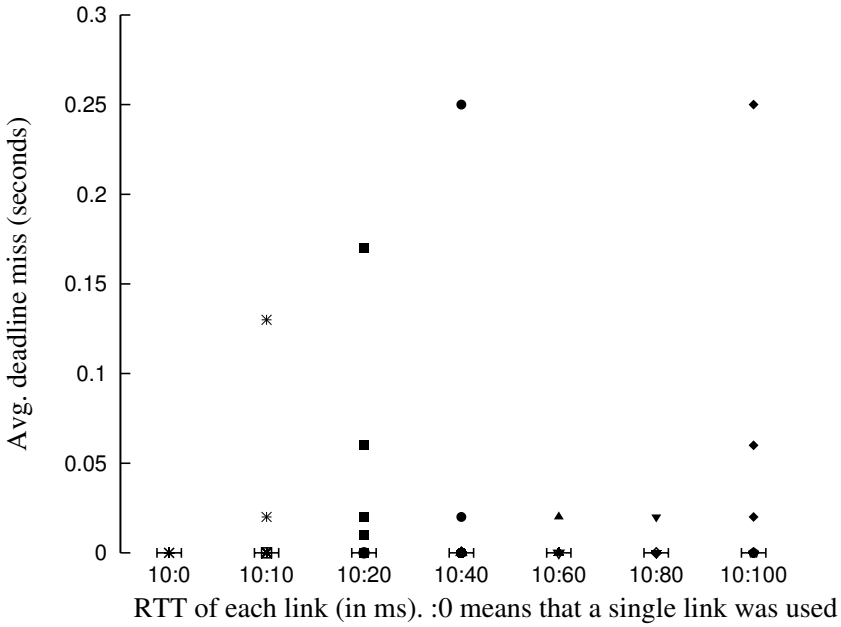


Figure 4.14: Deadline misses when the scheduler is faced with latency heterogeneity in a controlled environment, using the static subsegment approach.

first bytes of a requested subsegment is received. Then, for example packet loss or an overestimation of bandwidth by the scheduler can lead to deadline misses. However, the observed lateness was not significant compared to the complete segment length. The average lag for all bandwidth ratios was close to 0 s, and the maximum observed reduction

in liveness was less than 0.3 s.

Buffering is one way of avoiding deadline misses. By adding a larger buffer and sacrificing the liveness, the client has enough stored data to compensate for deadline misses. The same test performed with a buffer size of one and three segments confirm this. A buffer size of one caused a significant increase in the number of deadline misses, and they were more severe. When the buffer was increased to three segments, all deadline misses were eliminated.

4.4.2 Dynamic links

Dynamic links impose different challenges than static links do. Their behaviour requires that the request scheduler adapts to changes in the network, often rapidly. We evaluated the scheduler and static subsegment approach in two different scenarios. In the first, we emulated dynamic network behaviour to control all parameters and analyze how the scheduler and subsegment approach performs. In the second, they were evaluated using public WLAN and HSDPA networks to get an impression of their performance in the real world.

Emulated network dynamics

In order to emulate network dynamics, a script that at random intervals updated the bandwidth of the links was used. In addition, the RTT of the two links followed different distributions. The worst case bandwidth heterogeneity was 5:1 (2.5 Mbit/s and 0.5 Mbit/s), and a buffer size of six segments was used, according to the rule of thumb discussed in section 4.4.1.

Figure 4.15 shows the average achieved throughput for every run (40 runs), both when two different single links were used and when the client aggregated the bandwidth over multiple links. When both links were used at the same time, the throughput was most of the time equal to the bitrate-requirement for the highest quality. In total, 95 % of segments were in "Super" quality. When single links were used, the achieved throughput stayed between "Medium" and "High". With single links, 35 % of the segments had "Medium" and 20 % "High" quality, 26 % "Super" and 18 % "Low".

Figure 4.16 shows that deadline misses did occur when each of the links were used alone, but never when they were used together. This is as expected. When a single link was used, the link was often unable to meet the bandwidth requirement of the requested quality because of the fluctuating bandwidth. When the bandwidth was aggregated, the two links together were always able to provide the required bandwidth.

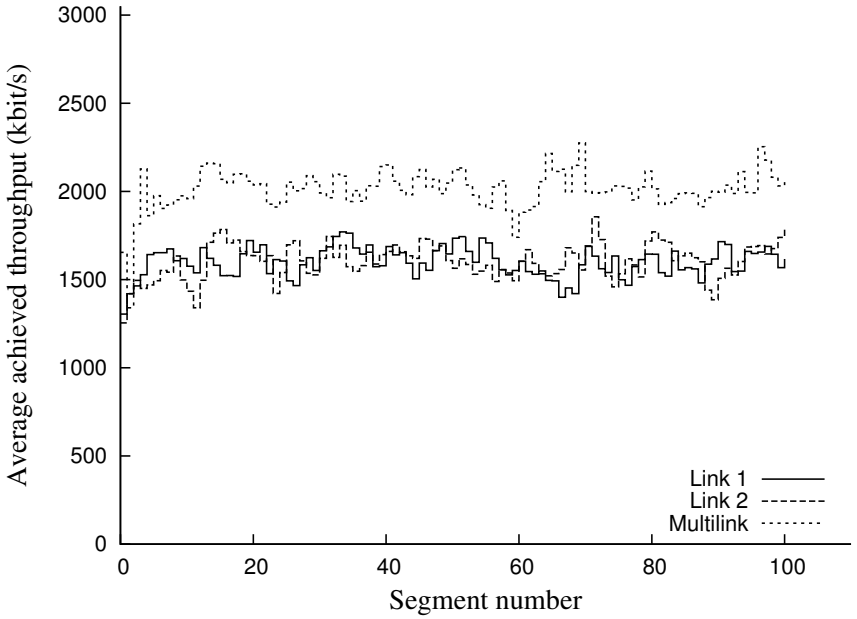


Figure 4.15: The average achieved throughput (for every segment) of the scheduler with emulated dynamic network behaviour, using the static subsegment approach.

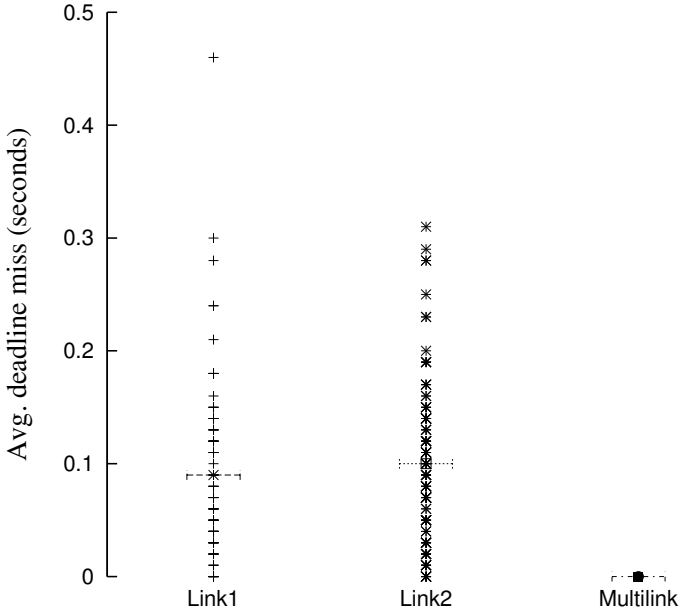


Figure 4.16: Deadline misses of the scheduler with emulated dynamics, using the static subsegment approach.

Real-world networks

To get an impression of the scheduler's performance over real wireless networks, we experimented with the multilink-enabled DAVVI player using WLAN and HSDPA networks summarized in table 4.2. The observed worst case heterogeneity was also here around 5:1,

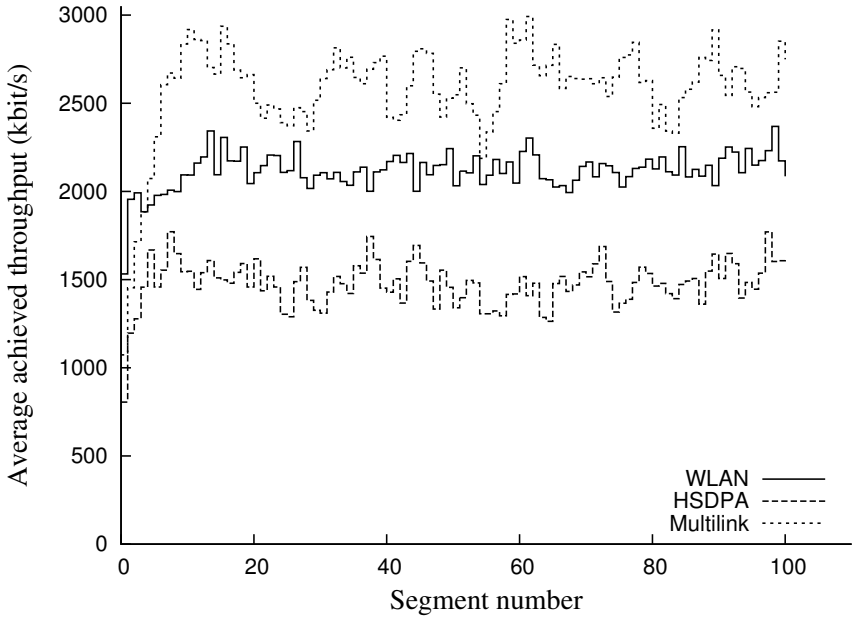


Figure 4.17: Average achieved throughput of the scheduler in real-world wireless networks, using the static subsegment approach.

so a buffer size of six segments was used.

Figure 4.17 shows the average achieved throughput (also over 40 runs) for every requested segment. The scheduler improved the performance and thereby the video quality significantly. With the fastest of the two interfaces, WLAN, 45 % of the segments were in "Super" quality, compared to 91 % when both links were used. The worst observed deadline miss over both links was only 0.3 s.

4.4.3 Summary

The static subsegment approach divides the two-second video segments generated by DAVVI into fixed-size 100 KB subsegments. These subsegments are requested in parallel and are distributed over the multiple links by the request scheduler. The multilink performance of the request scheduler and the static subsegment approach was evaluated with stable and dynamic link behavior, and was compared to that of the single, faster link. Using the static subsegment approach, adding a second link gave a significant increase in video quality, provided that the buffer was large enough to compensate for the bandwidth heterogeneity. Buffering and smart use of HTTP pipelining was able to compensate for the latency heterogeneity, i.e., the heterogeneity did not have an effect.

Increasing the buffer size, in order to compensate for the bandwidth heterogeneity, involves trading liveness for playback quality. This is in many cases not desirable or possible. In the next section, we present an improved subsegment approach which is able

to achieve the same performance independent of bandwidth heterogeneity (for a given buffer size).

4.5 Dynamic subsegment approach

The static subsegment approach does not handle the challenges introduced by limited receive buffers and timeliness optimally. It is unable to reach maximum performance unless the receive buffer is large enough to compensate for the bandwidth heterogeneity. Increasing the buffer size is in many cases not acceptable, desirable or even possible, as it involves reducing the liveness of a stream. We therefore designed the dynamic subsegment approach, which allocates data to the links in a more dynamic fashion. With the dynamic subsegment approach, each link is allocated their correct share of a segment (according to the measured throughput at the time of a request), so the slower links are made responsible for less data.

Algorithm 3 Dynamic subsegment approach *[simplified]*

```
1: block_length = number_of_interfaces * 100 KB
2: share_interface = throughput_link / aggregated_throughput
3: size_allocated_data = share_interface * block_length
4: if size_allocated_data >left_block then
5:     size_allocated_data = left_block
6: end if
7: left_block -= size_allocated_data
8: request new Subsegment(size_allocated_data)
```

Using the dynamic subsegment approach, as with the static subsegment approach, 100 KB is regarded as a well suited share of data to request over one link. However, the video segments are now divided into blocks of $number\_of\_interfaces$ * 100 KB (limited by the total segment size), and not fixed-size subsegments of 100 KB. These blocks are then divided into subsegments, and the size of each subsegment is decided by the measured throughput of the interface it will be requested through. Pipelining is still done as soon as possible. The dynamic subsegment algorithm is outlined in algorithm 3. First, the share of the throughput of the interface is calculated (line 2). Then, this value is used to decide the size of the subsegment (line 3). The size of the subsegment is adjust in case it exceeds the amount of data left in the block (line 4-6), before the block is updated and the subsegment is requested (line 7-8).

By allocating the data dynamically based on performance, the need for a big buffer is removed. A link should never be allocated more data than it can receive within the time limit. When dividing segments dynamically and based on the throughput, the perfor-

mance for a given buffer size should ideally be the same for all bandwidth heterogeneities. This approach is hereby referred to as the dynamic subsegment approach.

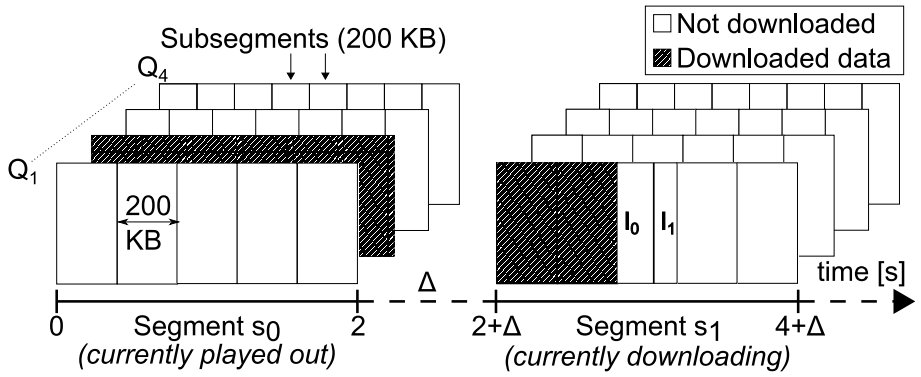


Figure 4.18: An example of the dynamic subsegment approach. The two interfaces I_0 and I_1 have finished downloading segment s_0 of quality Q_2. As the throughput dropped, the links currently collaborate on downloading the third subsegment of a lower quality segment.

Figure 4.18 shows an example of how the dynamic subsegment approach works. A block size of 200 KB is used, and the bandwidth ratio between the two links is 3:2. Instead of allocating a fixed size subsegment to either link, interface zero requests 120 KB and interface one 80 KB.

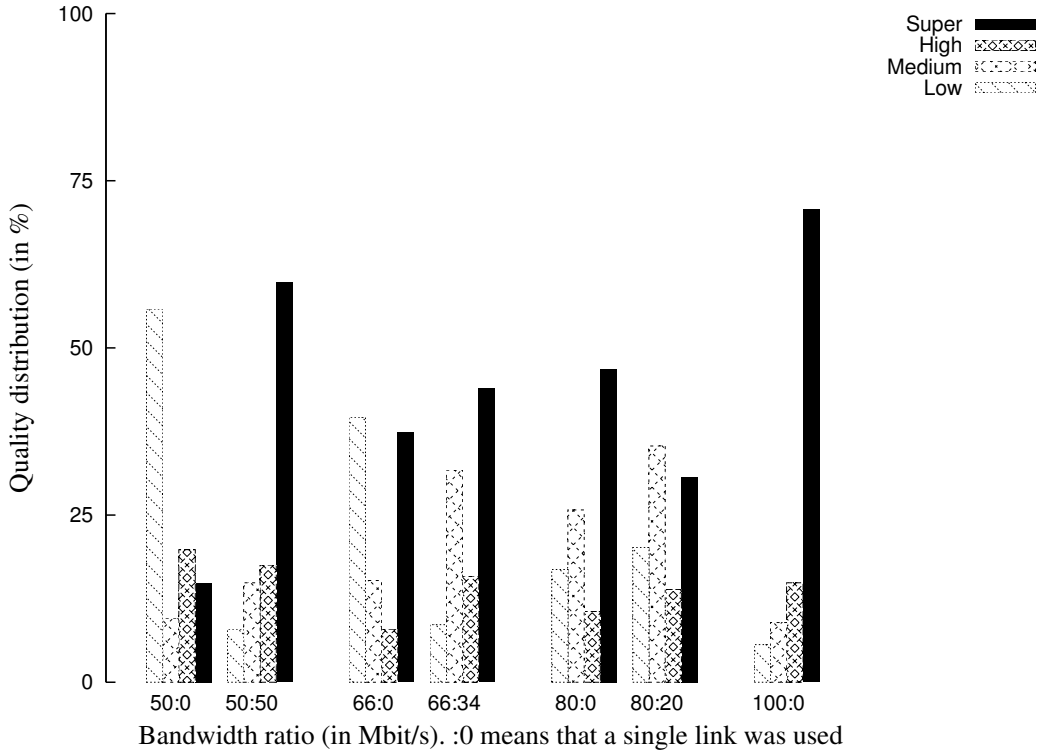
4.5.1 Bandwidth heterogeneity

In order to measure the effect of bandwidth heterogeneity on the two subsegment approaches, the controlled testbed was used and configured to provide different levels of bandwidth heterogeneity. The goal with using multiple links simultaneously, was that the performance should match that of a single 3 Mbit/s link.

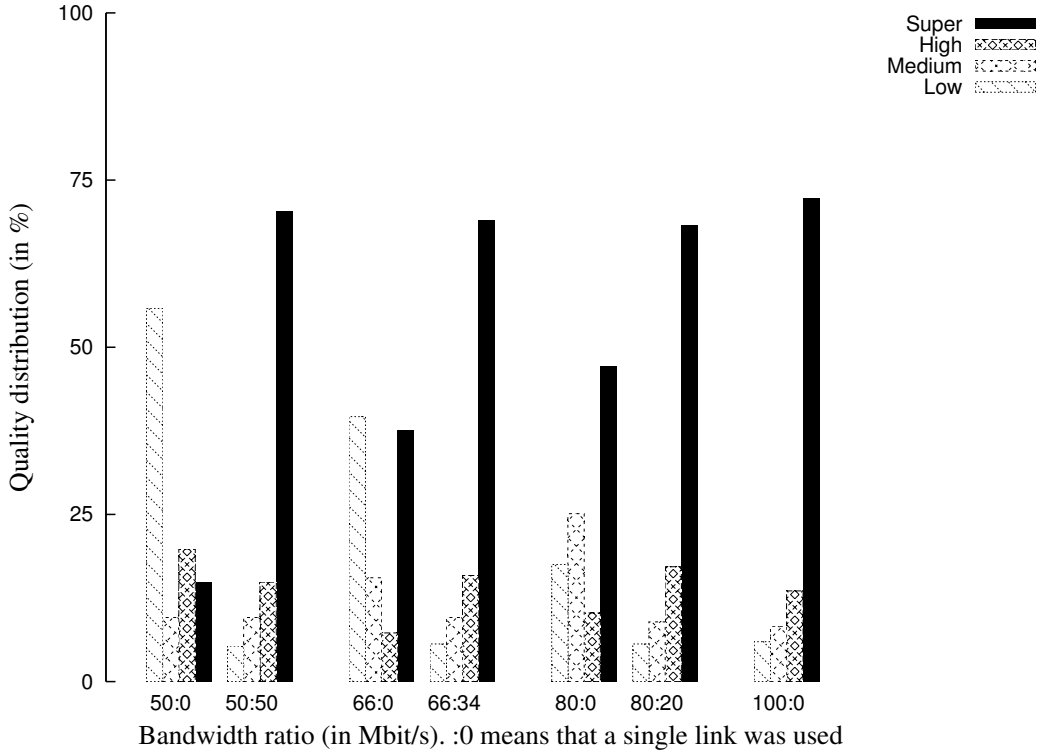
On-demand streaming

Figure 4.19 shows the quality distribution for the two subsegment approaches when evaluated together with on-demand streaming, with a buffer size of two segments. When a single link was used, the expected behavior can be observed. As the available bandwidth increased, so to did also the video quality.

With multiple links, the static subsegment approach behaved as earlier. At a bandwidth ratio of 80:20, using multiple links resulted in a worse quality distribution than when a single link was used alone. The dynamic subsegment approach, on the other hand, adapted to the heterogeneity. The performance was almost the same irrespective of link heterogeneity, and significantly better than when a single link was used. However,



(a) Static subsegment approach



(b) Dynamic subsegment approach

Figure 4.19: Video quality distribution for different bandwidth heterogeneities, buffer size/startup delay of two segments (4 seconds) and on-demand streaming.

the performance never reached the level of a single 3 Mbit/s link. This was caused by the initial segment, which was requested in the highest quality and had a higher bandwidth requirement than 3 Mbit/s. The low bandwidth link was allocated too much data, and, due to the limited buffer, this caused an idle period for the high bandwidth link. In order to avoid further deadline misses, the scheduler requested the next segment(s) in a lower quality. This observation is valid for all the results presented in this section. In other words, the multilink performance was never as good as a single 3 Mbit/s link.

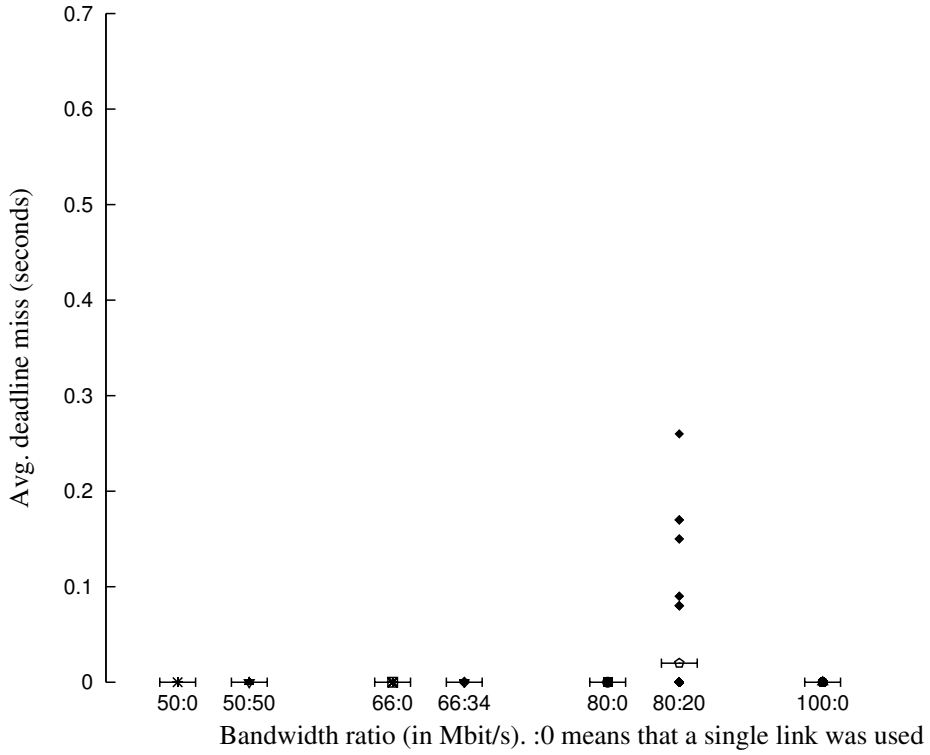
Figure 4.20 shows the average number of deadline misses for the bandwidth ratios, and both subsegment approaches performed well. The bandwidth measurements and quality adaption were accurate, there were close to no deadline misses, except when the buffer was unable to compensate for the heterogeneity. The deadline misses when the bandwidth ratio was 80:20 and the static scheduler was used, were caused by the slow interface not receiving the data fast enough. However, all deadline misses were significantly shorter than the segment length of two seconds. The worst observed miss was only of ˜0.3 seconds.

Live streaming with buffering

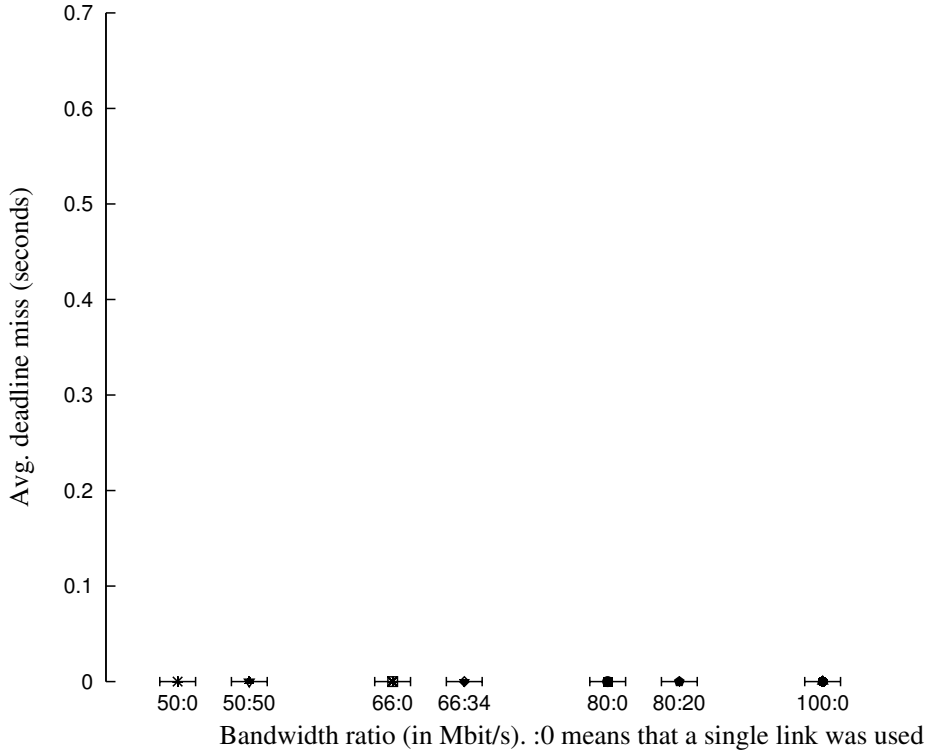
With live streaming with buffering, the results were similar to those of the on-demand streaming tests. When multiple links were used, the dynamic subsegment approach showed similar performance irrespective of bandwidth heterogeneity, while the performance of the static subsegment approach suffered from the buffer being too small to compensate for the link heterogeneity. The number of deadline misses were also the same as with on-demand streaming. The reason for these similar results is that segments were always ready also when live streaming with buffering was used. The client was never able to fully catch up with the no-delay broadcast.

With on-demand streaming, it makes no sense to discuss liveness. However, in live streaming with buffering, liveness is one of the most important criteria. With a buffer size of two segments, the static subsegment approach added an additional worst-case delay of 4 seconds compared to the no-delay broadcast. This means that, in addition to the delay caused by the startup latency and retrieval of the initial segments, the deadline misses caused the stream to lag an additional 4 seconds behind the broadcast. The dynamic subsegment approach resulted in an additional worst-case delay of 2.5 seconds.

Figure 4.21 shows the effect of increasing the liveness to the maximum allowed by DAVVI. Both the startup delay and buffer size was set to one segment (two second delay). The dynamic subsegment approach was able to cope well with the increased liveness requirement, and showed a significant increase in performance compared to using a single link. Also, the performance was independent of the bandwidth heterogeneity. The static subsegment approach, on the other hand, struggled because of the small buffer.

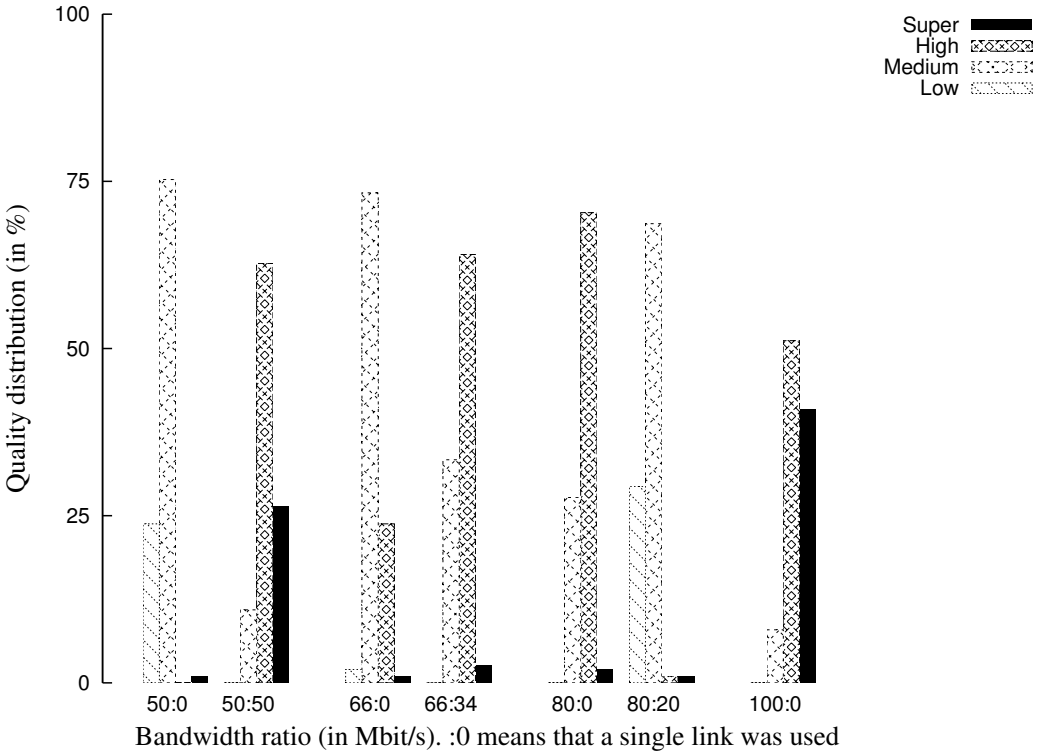


(a) Static subsegment approach

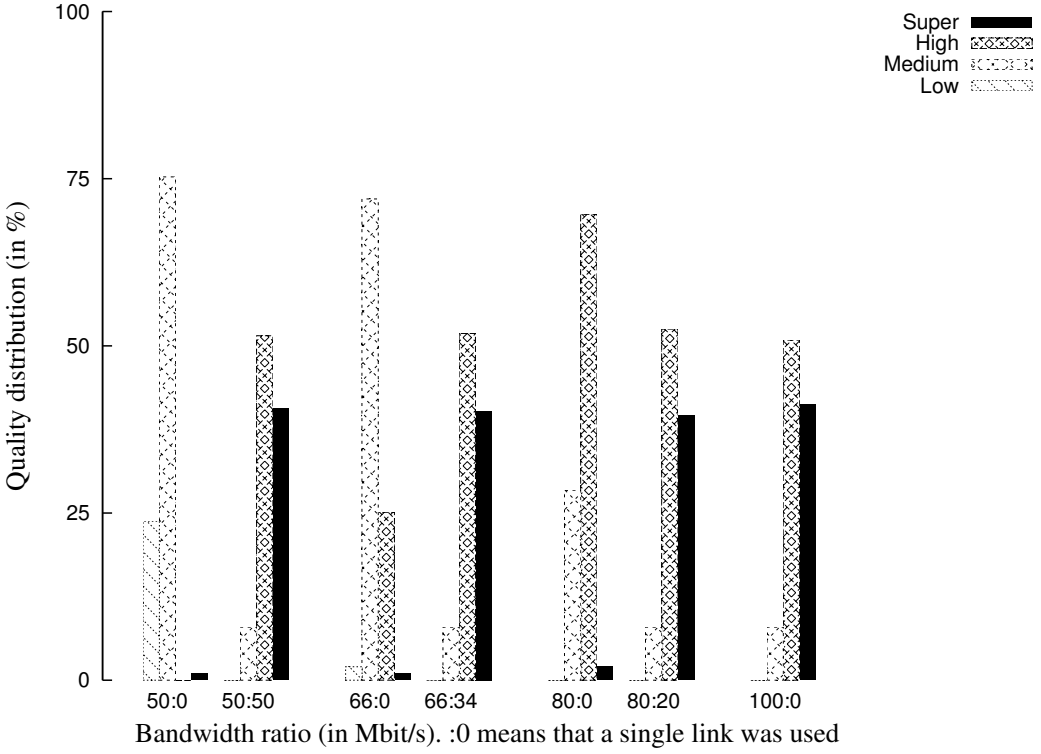


(b) Dynamic subsegment approach

Figure 4.20: Deadline misses for different levels of bandwidth heterogeneity with on-demand streaming, buffer size/startup delay of two segments (4 seconds).

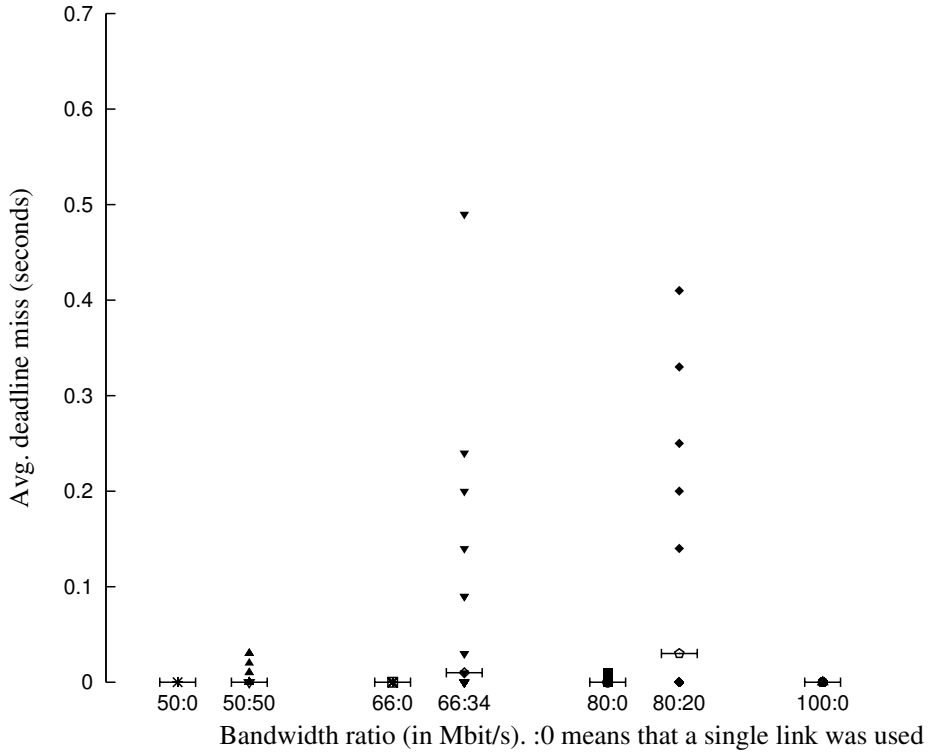


(a) Static subsegment approach

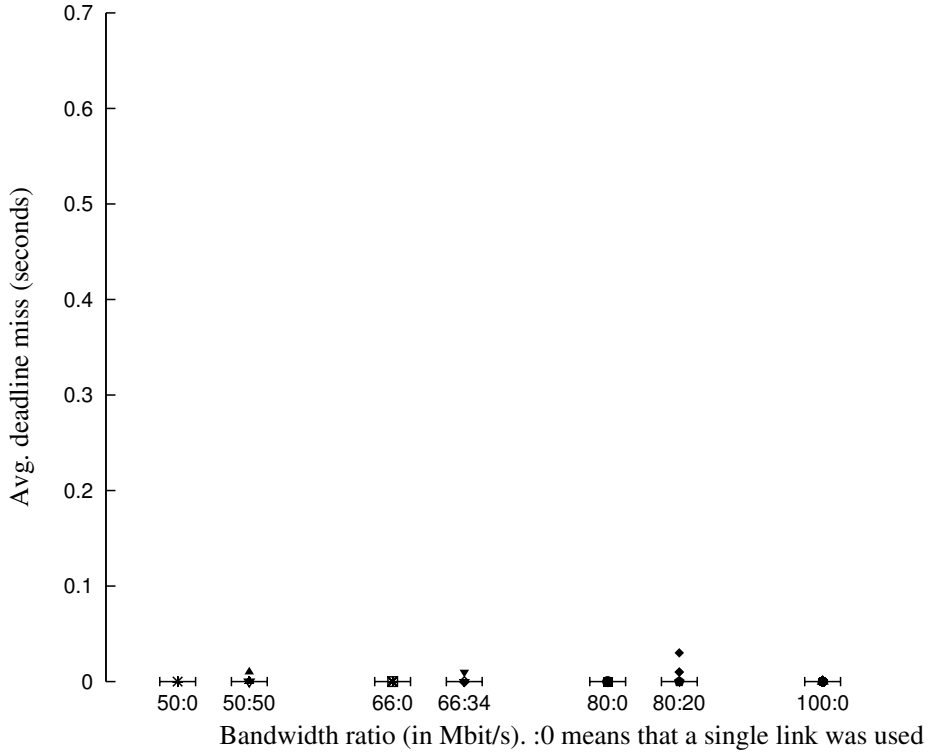


(b) Dynamic subsegment approach

Figure 4.21: Video quality distribution for different levels of bandwidth heterogeneity, buffer size/startup delay of one segment (2 second startup delay) and live streaming with buffering.



(a) Static subsegment approach



(b) Dynamic subsegment approach

Figure 4.22: Deadline misses for a buffer size of one segment (2 second startup delay) and various levels of bandwidth heterogeneity, live streaming with buffering.

In addition to pipelining only being effective within a segment, the buffer size problem became even more apparent. The performance hit was reflected in the deadline misses, shown in figure 4.22. While the dynamic subsegment approach was able to avoid almost all deadline misses, the static subsegment approach caused several misses. When the dynamic subsegment approach was used, a worst-case additional delay of 2.3 seconds was observed, compared to 6 seconds with the static subsegment approach.

Live streaming without buffering

By skipping segments, i.e., only requesting the most recent segment, a client can try to catch up with the broadcast, and skipping segments leads to interruptions in playback. However, prioritizing liveness did not affect the video quality, as shown in figure 4.23. The results were the same as for live streaming with buffering and a buffer size/startup delay of one segment (figure 4.21) - the dynamic subsegment approach improved the performance significantly compared to a single link, while the performance of the static subsegment approach suffered due to the limited buffers. The deadline misses were similar to figure 4.22. Unlike the static subsegment approach, the dynamic subsegment approach was able to avoid most deadline misses.

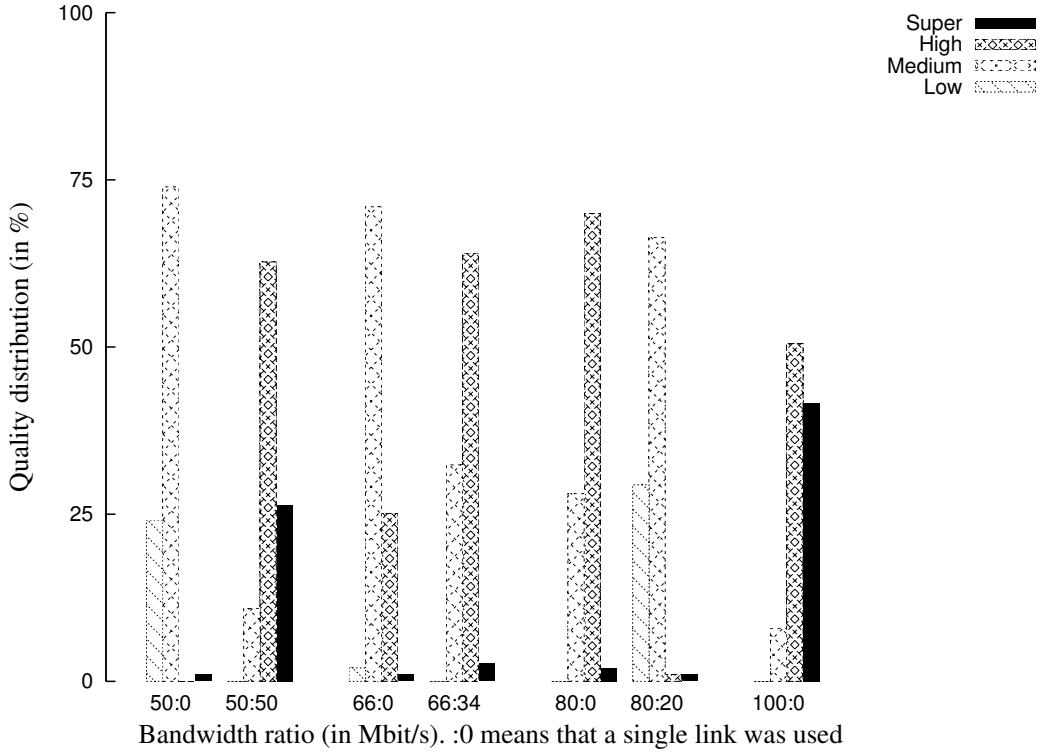
However, the number of skipped segments were the same for both subsegment approaches, with a worst case of two segments. This was because of the first segment, which is requested in the highest quality to get the most accurate throughput measurements. Both subsegment approaches assume that all links are equal and initially allocate the same amount of data to each. If the links are heterogeneous, which was the case in almost all of our experiments, or unable to support the video quality, the first segment will often take longer than two seconds to receive. As a new segment is generated every two seconds, the transfer time causes the client to skip one or more segments.

4.5.2 Latency heterogeneity

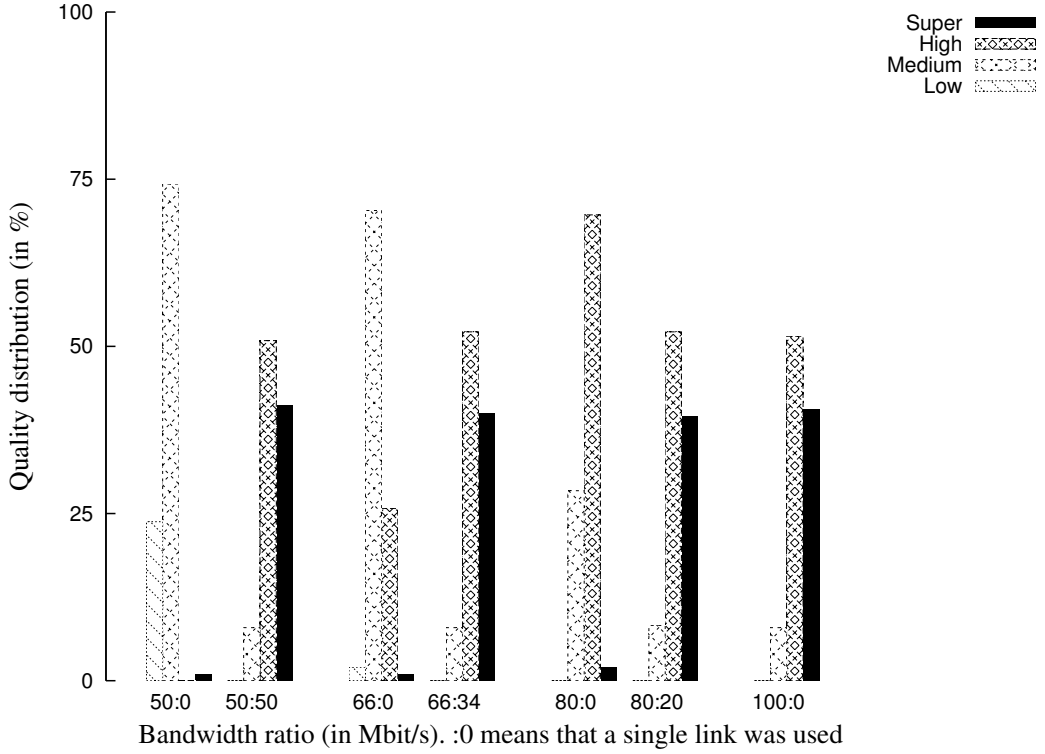
When measuring the effect of latency heterogeneity on video quality and deadline misses, the controlled network environment testbed was used. The bandwidth of each link was limited to 1.5 Mbit/s to avoid bandwidth heterogeneity affecting the results, and a buffer size of two segments was used.

On-demand streaming

Figure 4.24 depicts the video quality distribution for different levels of latency heterogeneity. As shown, latency heterogeneity did not have a significant effect on video quality, independent of subsegment approach. Smart use of HTTP pipelining and the buffering

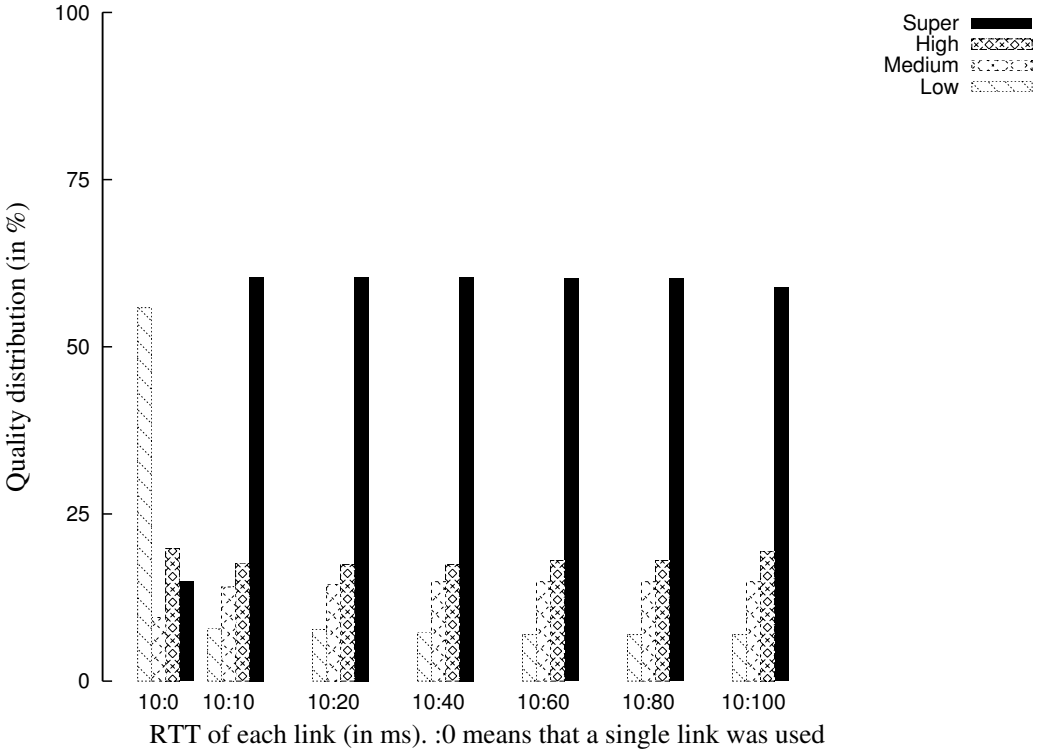


(a) Static subsegment approach

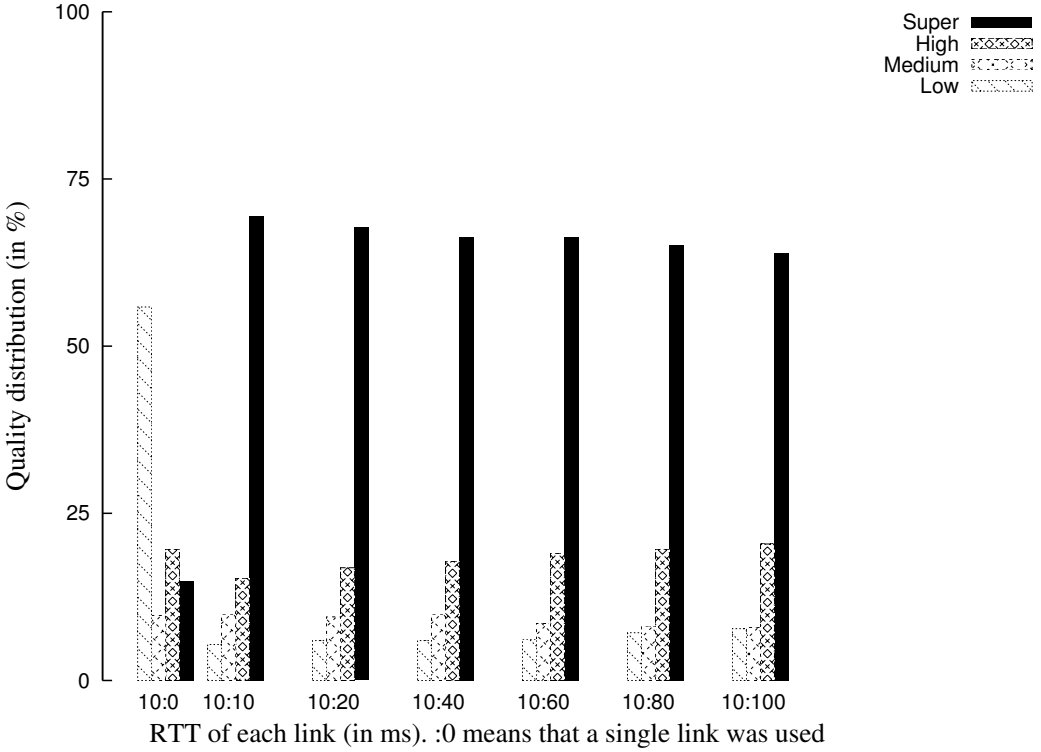


(b) Dynamic subsegment approach

Figure 4.23: Video quality distribution for a buffer size of one segment (2 second startup delay), live streaming without buffering and bandwidth heterogeneity.



(a) Static subsegment approach



(b) Dynamic subsegment approach

Figure 4.24: Video quality distribution for two-segment buffers and various levels of latency heterogeneity.

compensated for the heterogeneity. The bandwidth ratio was 50:50, and both subsegment approaches achieved close to the same quality distribution as in the on-demand bandwidth heterogeneity experiments (for a 50:50 bandwidth ratio), shown in figure 4.19, for all latency heterogeneities. The reason for the performance difference between the two subsegment approaches, is that the dynamic subsegment approach is able to use the links more efficiently.

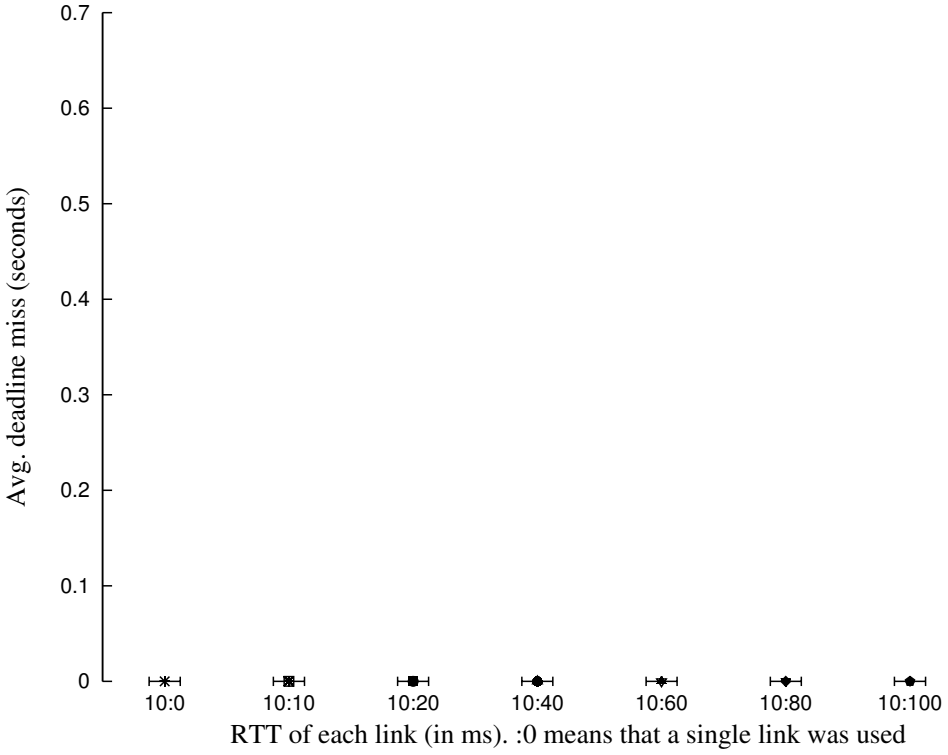
A slight decrease in video quality as the heterogeneity increased can be observed for both subsegment approaches. This indicates that the latency heterogeneity at some point will have an effect. The reason for the quality decrease is that it takes longer to request, and thereby receive, the subsegments over the high RTT link. The client then measures a lower throughput and potentially reduces the quality of the requested segments. Even though HTTP pipelining in most cases is able to compensate for the RTT, it is not possible when the buffer is full and the next segment cannot be requested immediately. When space is again available in the buffer, it will take at least one RTT before the first bytes of a subsegments arrives. In order to get accurate throughput measurements, the client starts measuring when the request is sent. Thus, a high RTT will lead to a lower measured throughput. Also, the TCP throughput is lower for short transfers over high delay links. As we use TCP as the underlying transport protocol, this also affects the performance. The congestion window grows slower, and it will take longer time to recover from packet loss.

The deadline misses, shown in figure 4.25, were also similar to the 50:50-case from the bandwidth heterogeneity experiments. As expected in a stable network environment, both subsegment approaches made accurate decisions and no deadline misses were observed.

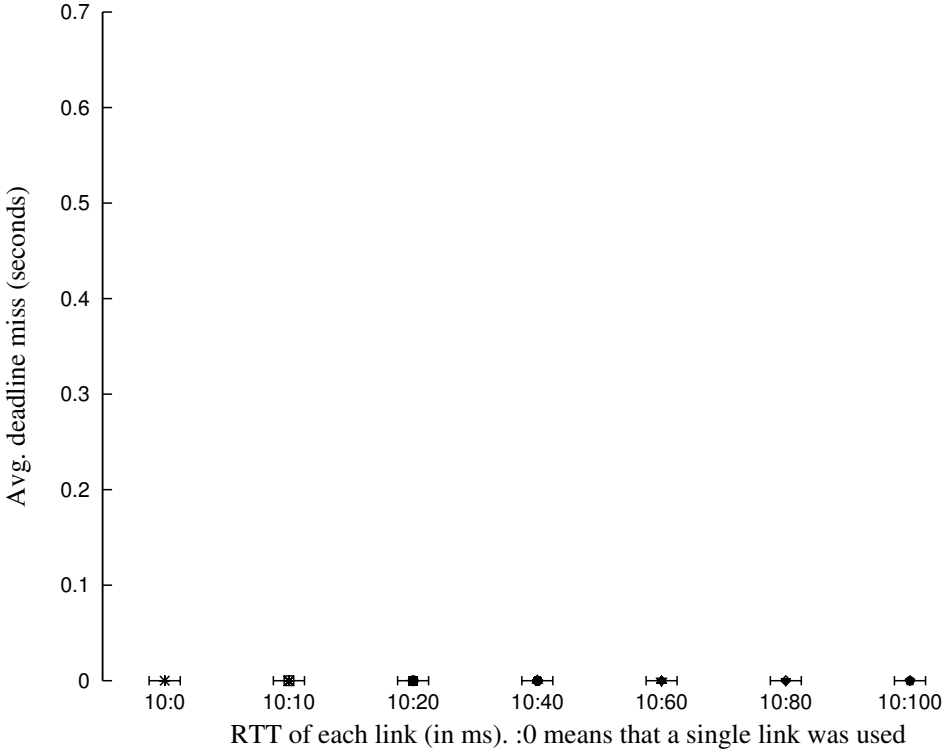
Live streaming with buffering

As with bandwidth heterogeneity, the results when measuring the effect of latency heterogeneity on live streaming with buffering were very similar to those with on-demand streaming. The quality distribution and deadline misses were not affected for the levels of heterogeneity used in this experiment. However, a slight decrease in video quality as the latency heterogeneity increases was seen also here. The worst case observed additional delay compared to the no-delay broadcast was 2 s for both subsegment approaches. This was, as with bandwidth heterogeneity, caused by the first segment being requested in the highest quality.

Reducing the buffer size to one, caused a similar reduction in performance to the ones seen in figures 4.21 and 4.22 (for a 50:50 bandwidth ratio). However, as for a buffer size of two segments, the latency heterogeneity did not affect the quality distribution or deadline misses. Both subsegment approached caused a worst case additional delay of 2.5 s.



(a) Static subsegment approach



(b) Dynamic subsegment approach

Figure 4.25: Average deadline misses for a buffer size of two segments (4 second startup delay), with latency heterogeneity.

Live streaming without buffering

The observed video quality and deadline misses using live streaming without buffering, were similar to the earlier latency heterogeneity experiments. Latency heterogeneity did not have a significant impact on video quality, however, a slight decrease can be observed, indicating that the latency heterogeneity will affect the performance at some point. As in the bandwidth heterogeneity experiments for live streaming without buffering, the number of skipped segments and the total delay compared to the no-delay broadcast were the same for both approaches. When multiple links were used, zero segments were skipped, and a worst case additional delay of 1.86 seconds was observed for both subsegment approaches. This was caused by the first segment. Even though the request scheduler's initial assumption that the links are homogeneous, was correct, the links were unable to support the bandwidth requirement for this segment.

4.5.3 Emulated dynamics

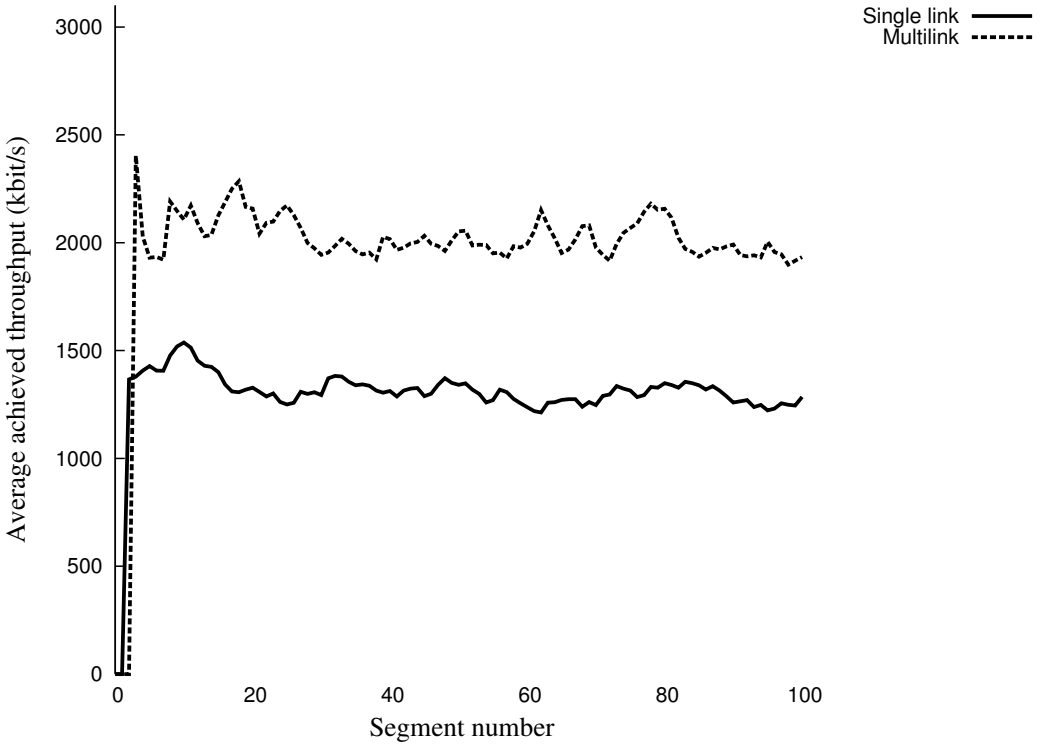
Dynamic links impose different challenges than static links, the scheduler has to adapt to often rapid changes in the network. To expose the two subsegment approaches to dynamic links while still having some control over the parameters, the script described in section 4.3.1 was used. A buffer size of six segments was used to compensate for the worst case bandwidth heterogeneity, except for in the live streaming without buffering experiments. Each subsegment approach was tested 30 times for each type of streaming, and the results shown are the averages of all measurements.

On-demand streaming

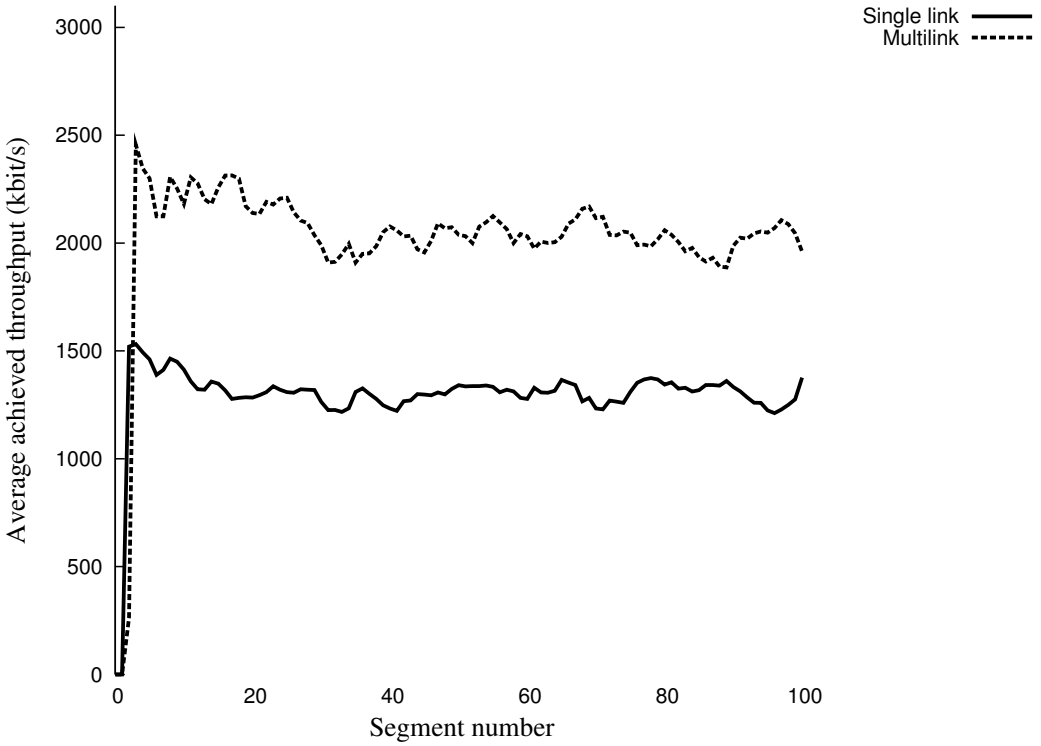
Subsegment approach	Low	Medium	High	Super
Static, single-link	31%	27%	28%	15%
Static, multilink	4%	4%	11%	81%
Dynamic, single-link	30%	26%	29%	15%
Dynamic, multilink	3%	3%	10%	83%

Table 4.4: Quality distribution for emulated dynamics and on-demand streaming.

The aggregated throughput when combining emulated link dynamics with on-demand streaming, is shown in figure 4.26. With both subsegment approaches, adding a second link gave a significant increase in throughput, and thereby achieved video quality. Also, as in the other experiments where the buffer size was large enough to compensate for link heterogeneity, both approaches gave close to the same video quality distribution, with a slight advantage to the dynamic subsegment approach. The average aggregated



(a) Static subsegment approach



(b) Dynamic subsegment approach

Figure 4.26: Average achieved throughput of the schedulers with emulated dynamic network behaviour, on-demand streaming.

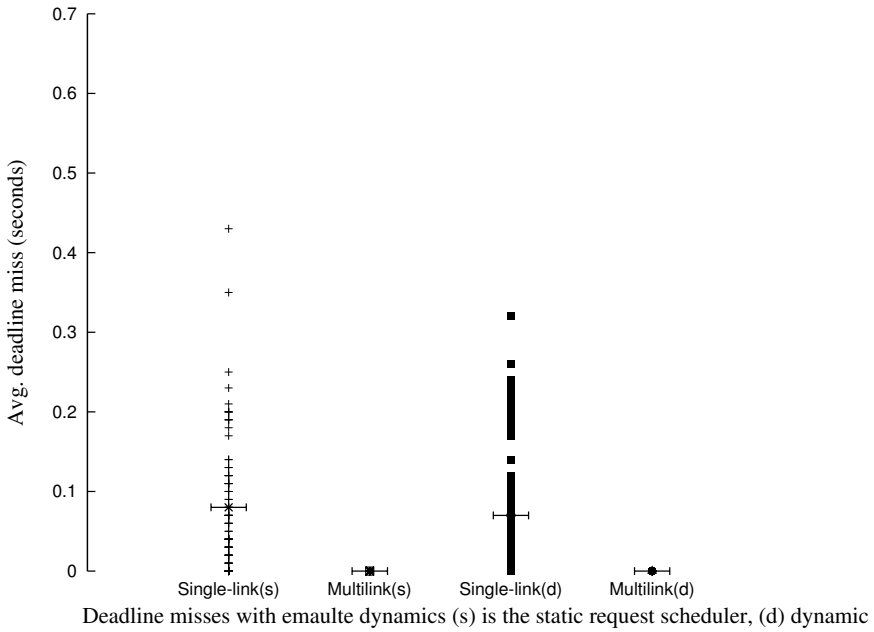


Figure 4.27: Deadline misses with on-demand streaming and emulated dynamics.

throughput oscillated between the average bandwidth requirement for "High" and "Super" quality. The quality distribution is presented in table 4.4.

In terms of deadline misses, shown in figure 4.27, both approaches were equally accurate. When a single link was used, misses occurred due to a single link being more vulnerable to the fluctuating link characteristics, however, none was severe. The worst case observed miss for both approaches was of less than 0.5 seconds. With multiple links, both approaches avoided all deadline misses.

Live streaming with buffering

Subsegment approach	Low	Medium	High	Super
Static, single-link	30%	26%	28%	16%
Static, multilink	4%	4%	11%	81%
Dynamic, single-link	29%	26%	29%	15%
Dynamic, multilink	3%	3%	11%	82%

Table 4.5: Quality distribution for emulated dynamics and live streaming with buffering.

As with both bandwidth and latency heterogeneity, the performance of live streaming with buffering was similar to the on-demand streaming experiments, seen in figure 4.26. A significant increase in performance was seen when a second link was added, and the quality distribution is found in table 4.5. The deadline misses were also the same as in the on-demand experiments (figure 4.27), and when multiple links were used, no misses

occurred. The worst-case additional delay compared to the no-delay broadcast was of 2.3 seconds, caused exclusively by the initial segment transfer time.

Live streaming without buffering

The live streaming without buffering experiments were performed with the same settings as used in the other emulated dynamics experiments, except that a buffer size and startup delay of one segment were used. This was, as discussed earlier, done to increase the liveness to the maximum that DAVVI allows (one segment).

Subsegment approach	Low	Medium	High	Super
Static, single-link	41%	44%	14%	1%
Static, multilink	15%	45%	35%	5%
Dynamic, single-link	44%	41%	14%	1%
Dynamic, multilink	2%	28%	55%	15%

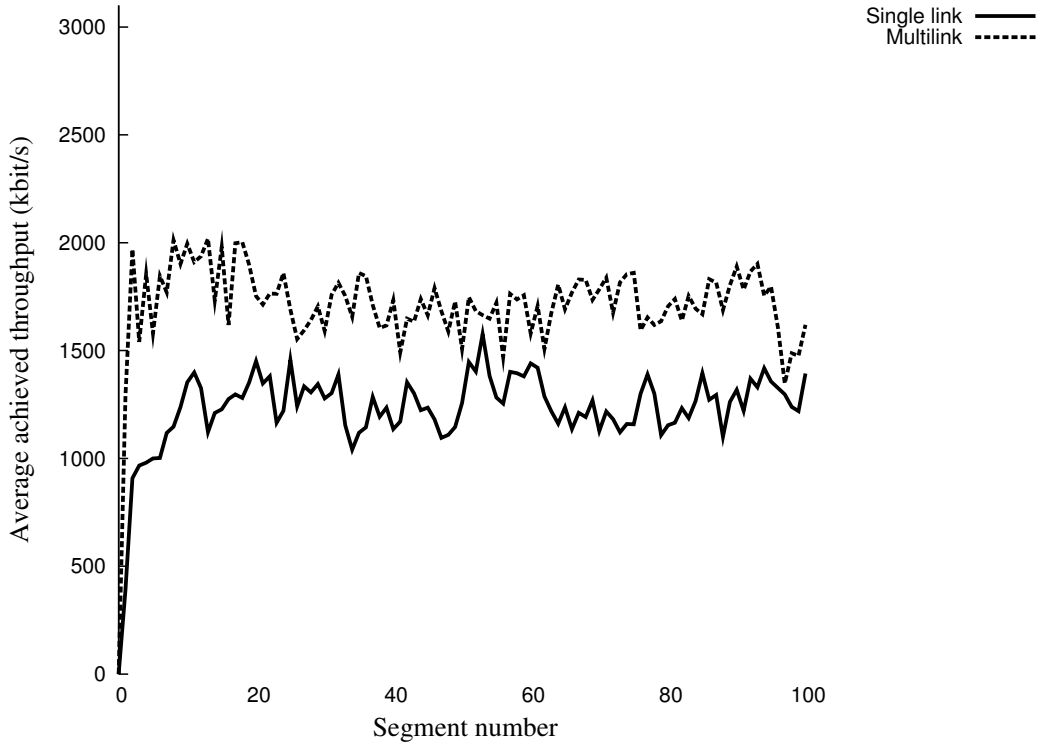
Table 4.6: Quality distribution for emulated dynamics and live streaming without buffering.

As in the earlier live streaming without buffering experiments, the two subsegment approaches performed differently, the static approach was outperformed by the dynamic approach. The reason is that the dynamic subsegment approach adapts better to smaller buffers, and the performance difference is reflected in the quality distribution, presented in table 4.6, and seen in figure 4.28. While the static subsegment approach most of the time achieved a throughput that exceeded the average requirement for "Medium" quality, the dynamic subsegment approach exceeded the requirement for "High" quality.

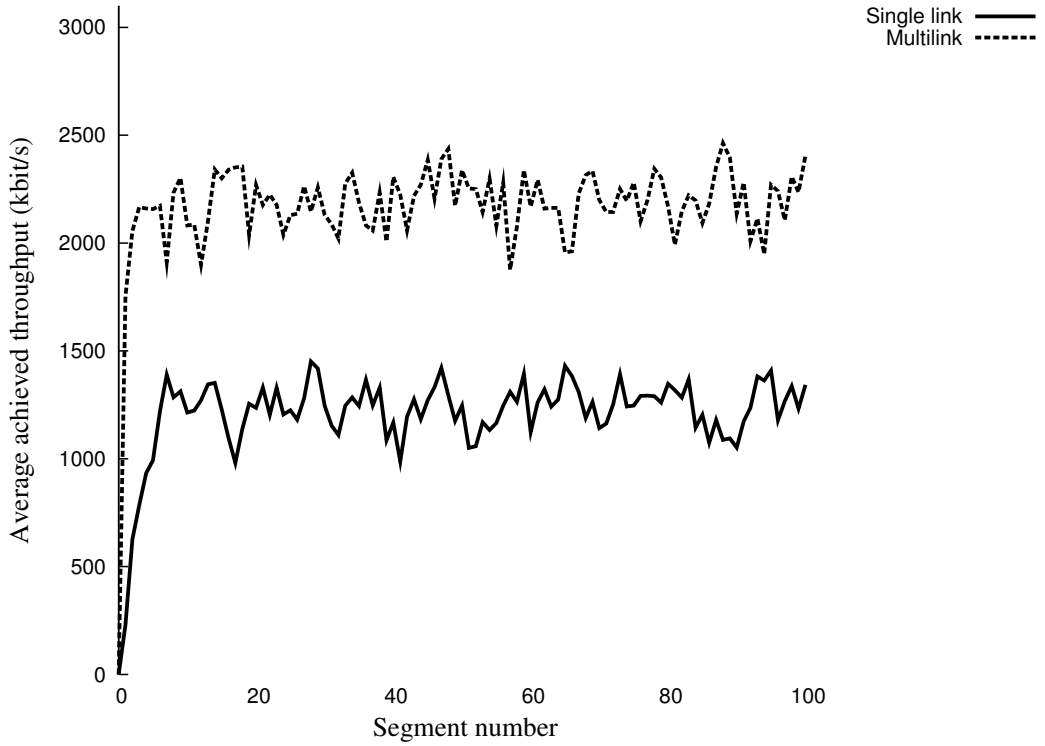
However, both subsegment approaches experienced deadline misses, as shown in figure 4.29. No misses were severe. As before, the worst case observed miss was around 0.5 second. However, if continuous playback had been important, a bigger buffer and startup delay should have been used. This, of course, would involve making a trade-off between liveness and quality of the user experience. The deadline misses are also reflected in the number of skipped segments. For each deadline miss, the liveness is reduced and on average both subsegment approaches had to skip five segments in order to catch up to the broadcast.

4.5.4 Real world networks

Our real world experiments were conducted with the networks described in table 4.3, and a buffer size of three segments was used to compensate for the worst-case measured bandwidth heterogeneity. The only exception was when measuring the performance for live streaming without buffering.



(a) Static subsegment approach



(b) Dynamic subsegment approach

Figure 4.28: Average achieved throughput of the schedulers with emulated dynamic network behaviour, live streaming without buffering.

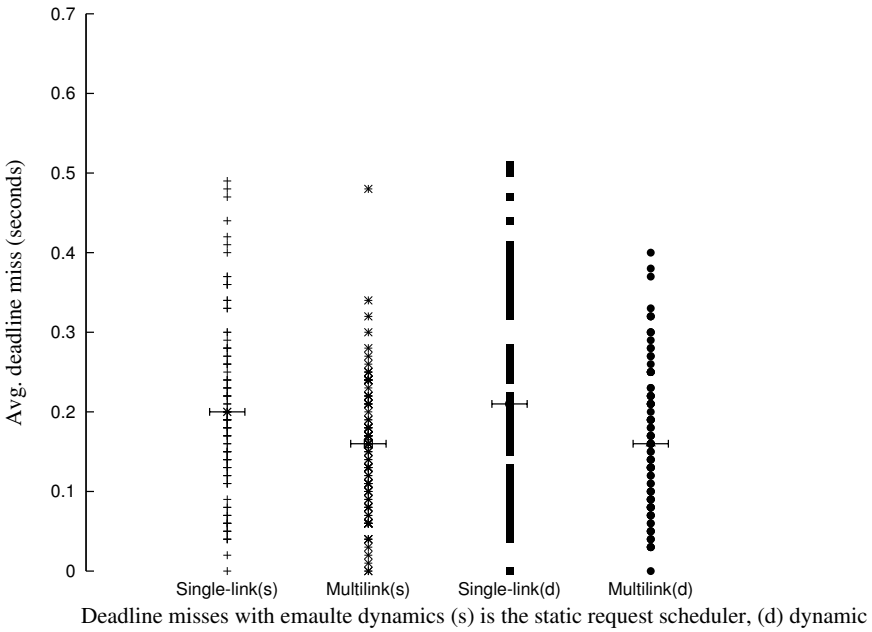


Figure 4.29: Deadline misses with live streaming without buffering and emulated dynamics.

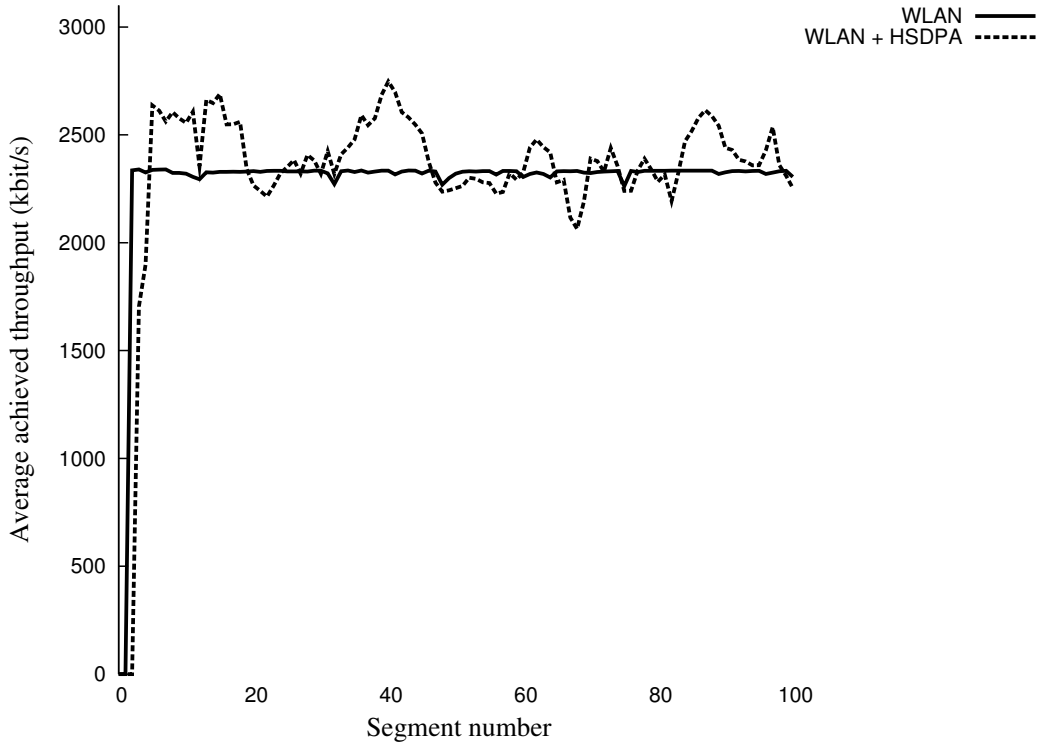
On-demand streaming

Subsegment approach	Low	Medium	High	Super
Static, single-link	1%	8%	51%	40%
Static, multilink	5%	6%	10%	79%
Dynamic, single-link	3%	11%	46%	41%
Dynamic, multilink	3%	2%	9%	86%

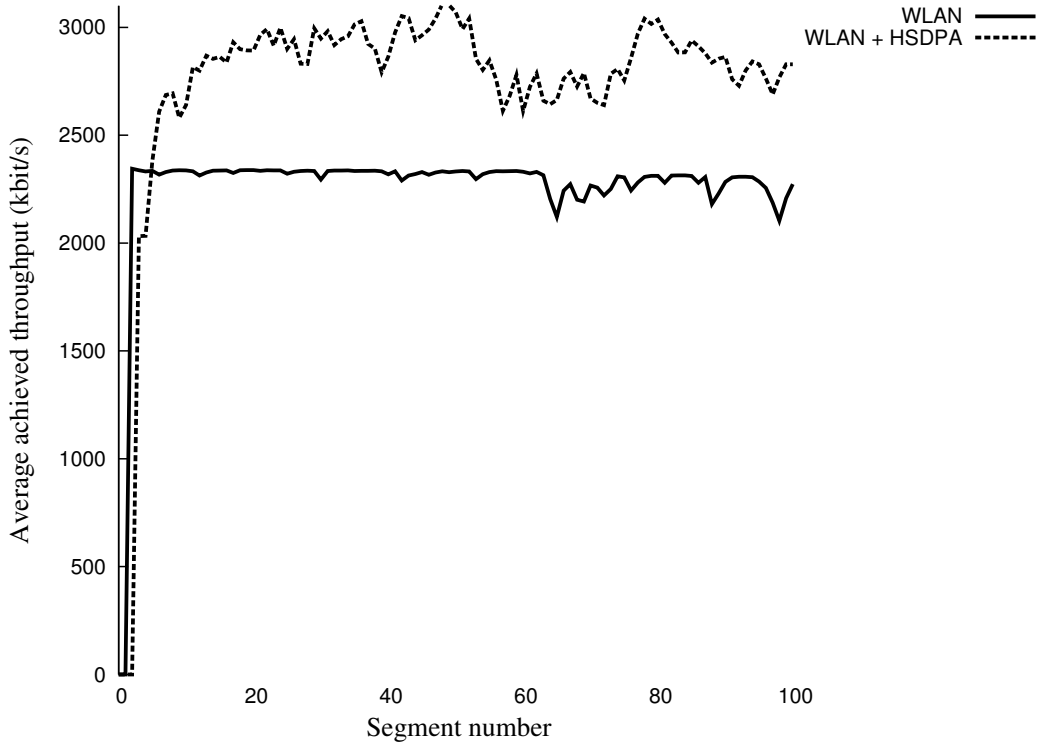
Table 4.7: Quality distribution for real world networks and on-demand streaming.

The average aggregated throughput for on-demand streaming and real world networks can be found in figure 4.30. There was a significant difference in performance between the two subsegment approaches. While the dynamic subsegment approach showed an increase in performance when a second link was added, the static subsegment approach did not benefit that much. In fact, sometimes the aggregated throughput was less than when a single link was used. The reason for the performance difference was, as earlier, that the dynamic subsegment approach is able to utilize the links more efficiently, it adapts better to the buffer size. Because the link's performance are used when calculating the subsegment size, the links should never be allocated more data than they can receive within the given time. The performance difference is also reflected in the quality distribution, shown in table 4.7.

In terms of deadline misses, both subsegment approaches performed equally. Except for some misses caused by significant and rapid changes in the network conditions, like



(a) Static subsegment approach



(b) Dynamic subsegment approach

Figure 4.30: Average achieved throughput of the schedulers with real-world networks, on-demand streaming.

Subsegment approach	Low	Medium	High	Super
Static, single-link	1%	10%	49%	40%
Static, multilink	5%	4%	7%	84%
Dynamic, single-link	1%	9%	49%	41%
Dynamic, multilink	3%	2%	5%	91%

Table 4.8: Quality distribution for real world networks and live streaming with buffering.

congestion and interference, both approaches were able to avoid deadline misses when multiple links were used.

Live streaming with buffering

The performance with live streaming with buffering was, as in the other live streaming with buffering experiments, similar to the on-demand performance. The quality distribution is shown in table 4.8 (for emulated dynamics), and both approaches avoided almost all deadline misses when multiple links were used. A worst-case additional delay compared to the no-delay broadcast of 4 seconds was observed for both subsegment approaches.

Live streaming without buffering

Subsegment approach	Low	Medium	High	Super
Static, single-link	0%	27%	68%	5%
Static, multilink	10%	12%	45%	32%
Dynamic, single-link	0%	27%	68%	5%
Dynamic, multilink	1%	10%	35%	55%

Table 4.9: Quality distribution for real world networks and live streaming without buffering.

When live streaming without buffering was combined with our real world networks, the performance was similar to the results presented in section 4.5.3. The static subsegment approach struggled with the small buffer, while the dynamic approach adapts better, which resulted in a significantly improved performance. The only significant difference compared to section 4.5.3, is that the quality distribution for both approaches were better due to more available bandwidth and more stable links, as can be seen in table 4.9. This was also reflected in the deadline misses and a lower number of skipped segments.

4.5.5 Summary

The static subsegment approach was able to improve the video quality significantly when a second link was added. However, it depends on the buffer being large enough to compensate for bandwidth heterogeneity. However, increasing the buffer size is in many cases

not desirable, as it reduces the liveness of the stream and increases the memory footprint of the client application. To avoid the buffer requirement and allow quasi-live streaming at high quality, we developed a dynamic subsegment approach that calculates the subsegment size dynamically, based on the current throughput of the interface. By doing this, a link should ideally never be allocated more data than it can receive within a given time.

In this section, we have compared the performance of the static and dynamic subsegment approach in both a fully controlled network environment, and with real-world networks. The two approaches were evaluated in the context of on-demand streaming and live streaming with and without buffering, and the dynamic subsegment approach always performed better. It was able to alleviate the buffer problem and showed similar performance independent of link heterogeneity for a given buffer size.

4.6 Conclusion

Application-specific bandwidth aggregation solutions allows for approaches that are tailored to a specific set of needs. We have focused on improving the performance of HTTP-transfers, using quality adaptive video streaming over HTTP as a case study. HTTP supports three features which make it suitable to combine with multiple links - range requests enables a client to request specific byte ranges (subsegments) of a file over independent interfaces in parallel, pipelining reduces (and ideally removes) the time overhead between subsegment requests, and persistent connections removes the need to open a new connection for every request. We would like to point out that even though we have used HTTP, similar features are offered by other protocols. Thus, our application layer bandwidth aggregation technique is also compatible with for example FTP.

In this chapter, we have evaluated two different subsegment approaches, the static and dynamic subsegment approach, together with on-demand streaming and live streaming with and without buffering. A subsegment approach decides how a file is divided into subsegments, and the approaches were evaluated in a controlled network environment and real-world networks. The approaches were implemented as extensions to the HTTP-based, quality-adaptive, segmented DAVVI streaming platform [43]. Each video is divided into fixed-length segments, and each segment is encoded at different bitrates in order to allow for quality adaption. In terms of video encoding and data retrieval, DAVVI offers the same features as many popular commercial and proprietary systems (for example Microsoft's SmoothStreaming [78]).

The static subsegment approach divides segments into fixed-size subsegments and improved the video quality significantly compared to that of a simple link. However, for the static subsegment approach to improve performance, the buffer size has to be large

enough to compensate for the bandwidth heterogeneity. Increasing the buffer size is in many cases not desirable or possible, as it for example reduces the liveness of a stream and increases the memory footprint of an application.

The dynamic subsegment approach was motivated by the buffer challenge. By basing the subsegment size on the capacity of the different links, the dynamic subsegment approached performed the same independent of bandwidth heterogeneity (for a given buffer size). Using this approach, the client was able to utilize the links more efficiently than the static subsegment approach, and a considerable increase in quality was seen. Both compared to a single link and the static subsegment approach.

Application-specific bandwidth aggregation techniques requires extending existing applications. This is not always possible or desirable (for example with closed-source applications), and, therefore, in the next chapter, we present techniques for transparent bandwidth aggregation of UDP and TCP.

Chapter 5

Transparent bandwidth aggregation

In many cases, extending an application with multilink support and bandwidth aggregation is not possible. For example, the source code might not be available or changes will incur a too large development and time overhead. Also, it is unrealistic to assume that most users will be willing to update the code themselves. A similar challenge exists on the server-side. Servers are often operated by third-parties that have no incentive to update their machines or software. If application-specific bandwidth aggregation is not possible, then transparent bandwidth aggregation can be used.

Bandwidth aggregation techniques that are transparent to the applications can be designed to operate at any layer of the IP stack. We have proposed techniques at the network layer. The network layer receives packets from and passes packets to the transport layer. IP is the default addressing protocol in the Internet today, is connectionless and IP packets will be routed through the Internet according to the information in the header. By intercepting packets at the network layer and updating the IP header (for example the destination address) or by tunneling packets, packets can transparently be routed to different interfaces at a multihomed device.

Network layer bandwidth aggregation techniques also has the advantage that transport protocol modifications are not needed. Every transport protocol in use on the Internet creates a layer 3 protocol data unit, consisting of the transport protocol header and the payload. This unit is sent to the network layer, where it is encapsulated inside an IP header. Updating or changing a transport protocol, if at all possible, is not desirable as the change must be reflected in all machines that will communicate. External machines are often controlled by independent third parties without an incentive to update their systems. In addition, new transport protocols or transport protocol modifications take a long time until they reach wide-spread deployment. For example, SCTP, which was standardized already in 2000, is still not implemented in Windows.

In addition, network layer techniques do not have to overcome the deployment chal-

lenges faced by link layer bandwidth aggregation techniques. Link layer bandwidth aggregation techniques require the different interfaces at a client to be directly connected to the same endpoint. This is not feasible in our scenario, as we aggregate links using different technologies and that are connected to different networks. The deployment challenges faced by a network layer technique, especially the limited connectivity of a device caused by NAT, is solved by each transparent bandwidth aggregation technique being designed on top of the features offered by MULTI's invisible mode.

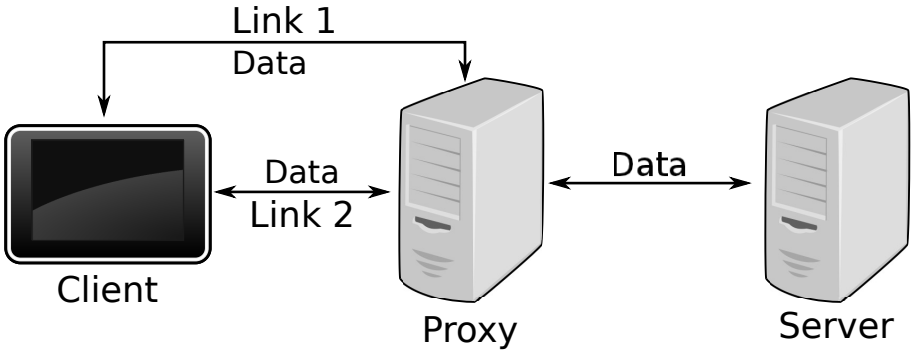


Figure 5.1: An example of a transparent bandwidth aggregation solution built around a proxy, for a client with two interfaces The stream is split/merged at the proxy and internally in the client, as unmodified hosts expects the original stream (the transport protocol).

A common way to enable bandwidth aggregation at the network layer, is to make use of a proxy in order to avoid changing the server, as shown in figure 5.1 for a client with to interfaces. Unmodified hosts are only able to receive/send the original packet streams, the streams must at some point be merged/split across the multiple links. The network layer bandwidth aggregation techniques we have found, are all based on incorrect assumptions or unrealistic requirements. For example, the technique presented in [59] requires changes to the protocol headers and protocol tuning, and assumes that latency heterogeneity can be compensated for by adjusting the packet size. Our observations contradict this, the latency heterogeneity depends on several factors (for example queueing delay in intermediate routers), and reducing the packet size will cause a higher network load (due to the larger number of packets that will be in transit). In [12], a technique based on an algorithm called *Earliest delivery path first* is introduced. The algorithm runs on a proxy, and the proxy is aware of the queue length at the different base stations that a client is connected to. However, this information is typically not available in real-world networks. In addition, in our scenario, there can be multiple hops (and thereby queues) between the proxy and the client, since they often belong to different networks. Finally, the authors have not considered the impact of links with fluctuating performance. Lastly, ECMP [34] allows users to assign static weights to different interfaces, and then these

weights are used to distribute connections across the available interfaces. However, for ECMP to be efficient, the available bandwidth of each link has to be static (so that the ratio between the weights are correct), and applications must be designed so that they open and make use of several simultaneous connections.

Unlike the existing network layer bandwidth aggregation techniques, our technique does not require modifications to existing protocols, depend on a certain application behavior or make unrealistic assumptions about available information. Also, link heterogeneity has been considered properly. No changes are needed at the server, because of the proxy, and only a small userspace application has to be installed on the client. Thus, neither the OS nor the transport protocols have to be modified. Because none of the existing network layer bandwidth aggregation techniques can be applied to our scenario, the performance of our technique is only compared to that of itself, i.e., the performance of multiple links versus a single link with the same bandwidth as the ideal, aggregated bandwidth.

Transport protocols provide a different, but sometimes partly overlapping set of features. For example, TCP guarantees a reliable connection and in-order delivery of the packets to the application, while UDP provides a best-effort transmission of data. In order to achieve efficient bandwidth aggregation, a transparent technique must support the behavior of the targeted transport protocol(s). Otherwise, the links will not be fully utilised. For example, incorrect distribution of traffic will limit the growth of TCP's congestion window and thereby the throughput of the connection (see figure 1.6). There exists a large number of transport protocols, but TCP and UDP are the main transport protocols in the Internet today. Thus, in this chapter, we have focused on improving the performance of TCP- and UDP-streams using multiple links.

5.1 Transparent bandwidth aggregation of UDP-streams

UDP provides a best-effort way for applications to communicate. A data packet is referred to as a datagram, and UDP does not provide any additional functionality like reliability or congestion control. In other words, there is by default nothing stopping a UDP sender from overflowing a network. UDP is mostly used by applications that prioritize low latency. For example, waiting for a retransmission might have a larger effect on a computer game than just ignoring the lost packet. The game might not be able to progress until the lost packet has arrived, causing interruptions in the gameplay.

Increasing the performance of UDP-based applications involves solving two challenges. First, packets have to be striped efficiently over the often heterogeneous network paths (between the client and the proxy) corresponding to each link at the client, and in a

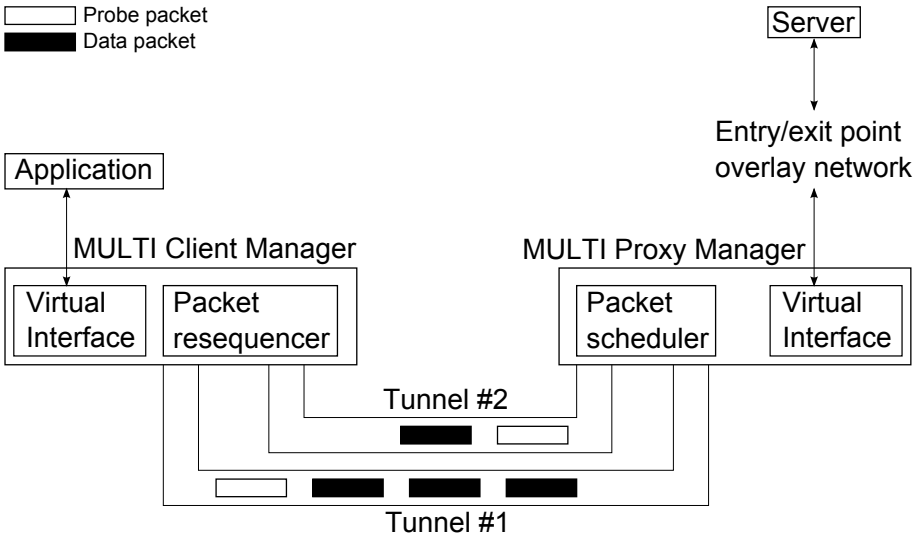


Figure 5.2: An overview of our multilink proxy architecture running on a client with two active interfaces.

manner which avoids congesting the paths. If packets are not striped according to path capacity, the paths will not be utilized to their fullest, while ignoring congestion will be unfair to other traffic. For example, if the bandwidth ratio between two paths is 3:1 and packets are scheduled round-robin, only a third of the capacity of the fastest path is used. Also, a UDP sender by default have no incentive to reduce its send rate. If the slowest path is not able to support its allocated share of the bandwidth, the congestion will reduce the performance of other streams that share the same network path.

Second, packet reordering caused by latency heterogeneity has to be compensated for. When packets are striped over paths with different latencies, they will in many cases arrive out of order at the client. Most applications process data sequentially and, thus, require packets to arrive in-order. Significant packet reordering might for example cause an increase in the buffer usage of a video streaming application.

In the rest of this section, we first introduce our proposed bandwidth aggregation technique, before continuing with the results from the evaluations we performed. To properly evaluate the performance, experiments were performed in a fully controlled environment and with real world networks. The latter gives an impression of how the proxy will benefit potential users, as well as how it performs with a combination of dynamic bandwidth and latency heterogeneity. A controlled environment allows us to isolate and control the level of bandwidth and latency heterogeneity, and study how they affect the performance. Finally, we summarise our contributions.

5.1.1 Architecture

Figure 5.2 summarizes our proposed bandwidth aggregation technique targeted at UDP-based applications. MULTI is used in invisible mode. In invisible mode, MULTI (described in chapter 3) makes use of a globally reachable proxy and a private overlay network is built between a client and the proxy. The network consists of one IP tunnel for each active network interface at the client. The MULTI managers create virtual interfaces, and the clients configures desired routes to go through this interface. The routing subsystem of the OS will then route the traffic through the virtual interface, and it will be intercepted by the managers. Packets are then encapsulated and sent through a tunnel. No changes have to be made to either application or transport layer protocol.

The proxy uses SNAT to ensure that all packets destined for the multihomed client passes through the proxy. When a packet destined for a client arrives at the proxy, the operating system looks up the mapping. After rewriting the address information in the header, the packet is forwarded to the client through the overlay network. Probe packets are sent at a given interval to maintain the overlay network and keep the NAT hole open (if any).

In order to solve the two challenges described earlier, compensating for packet reordering and efficient packet striping without causing path congestion, three different components are needed. *Congestion control* is used to avoid congesting the paths. The information gathered by the congestion control is also used to estimate the available path capacity. The *packet scheduler* uses the capacity estimates to efficiently stripe the packets. Finally, a *packet resequencer* at the client buffers out of order packets until reordering is resolved. The packet scheduler was implemented in the MULTI Proxy manager, the resequencer in the MULTI Client manager, while the congestion control requires support from both the client and the proxy manager. The rest of this subsection describes the three components in more detail.

Congestion control

Congestion control is a mechanism used by several different transport protocols. The purpose, as mentioned in related work, is to limit the send rate and avoid congestion collapse, a state in which little or no useful communication can happen because the network is congested. Congestion causes high loss rate and low throughput. There are several different congestion controls, and each is designed based on the features offered by the targeted transport protocol. For example, the different TCP variations all have rules for how to deal with packet loss and retransmission (due to the reliability requirement).

A transport protocol that includes congestion control and closely resembles UDP, is the Datagram Congestion Control Protocol [49] (DCCP). DCCP is a datagram based,

unreliable protocol that supports reliable connection setup and teardown, feature negotiation, Explicit Congestion Notification [64] and congestion control. The main purpose of DCCP is to provide applications using UDP with standardised congestion controls below the application layer. Earlier, if congestion control was to be used, it had to be implemented in each UDP-based application.

DCCP currently provides three different congestion controls, Congestion Control ID (CCID) 2 [28], CCID3 [30] and CCID4 [29]. CCID3 and CCID4 are targeted at UDP streams that sends fixed-sized packets and where a smooth send rate is important. CCID2, on the other hand, is, according to the RFC, not dependent on a fixed packet size and is recommended for applications that need to transfer as much data as possible as fast as possible. All three congestion controls are TCP fair (also known as friendly). This is important, as TCP is the dominating transport protocol, and no stream should claim more than its fair share of the resources.

Since DCCP is datagram-based and unreliable, the congestion controls can also be viewed as general congestion control mechanisms that can be used by UDP-based applications. The proxy applies congestion control to each active tunnel. Because the goal of the bandwidth aggregation technique presented in this section is to be completely transparent, no assumptions about packet size or desired send rate can be made. Also, the focus of this work has been on bandwidth intensive applications that need to send as much data as possible. For these reasons, we have used CCID2 rather than CCID3 or CCID4.

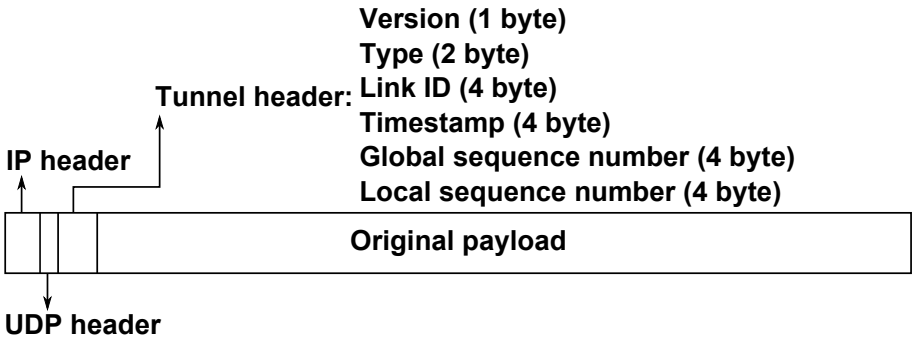


Figure 5.3: The packet format used with the transparent UDP bandwidth aggregation technique.

CCID2 closely resembles TCP's congestion control, described in section 2.1.2. CCID2 follows TCP's AIMD-rules (Additive Increase/Multiplicative Decrease) for congestion window development and uses slow-start, fast recovery, delayed ACKs and an equivalent of the retransmission timeout. The proxy maintains a congestion window for the tunnel established over each path, updated according to CCID2. The congestion window is updated based on ACKs (or lack of ACKs) sent from the client. Each tunnel data

packet, shown in figure 5.3, has two sequence numbers. One is a global sequence number that is valid for all tunnels and used by the packet resequencer, while the other is local to each tunnel. The local sequence number is the one used by CCID2 to detect packet loss and do congestion control. Packet loss is detected by checking for gaps in the acknowledged sequence numbers.

Packets occupy a certain amount of space in the congestion window, and the client acknowledges the packets it has received. The acknowledgements are used to free up space in the window and increase its size. If packet loss has occurred, the size of the congestion window will be reduced. The overall size of a congestion window gives a good estimate of the current path capacity. By comparing the total window size with the free space, an estimate of the available capacity can be derived.

One could also have used TCP as the tunneling mechanism, as this would have ensured fair utilization of the paths and removed the need for a new congestion control. However, TCP adds a significant overhead. In addition to a larger packet header, leaving less room for payload, TCP's reliability feature would increase the latency of the UDP stream. For example, if packet loss occurs, a TCP-based tunnel will not deliver more packets to the manager before the lost packet has been retransmitted and received. As UDP by definition is unreliable, adding reliability would alter the expected behavior of the protocol. Also, because UDP provides no additional features than best-effort delivery, it ensures that the behavior of the encapsulated transport protocol is preserved.

Packet scheduler

A packet scheduler is responsible for achieving efficient bandwidth aggregation by striping packets according to the available path capacity. Otherwise, the paths might not be fully utilized or congestion can occur. The packet scheduler used by our transparent bandwidth aggregation proxy makes its decisions based on the information gathered by the congestion control. Namely, information related to the each tunnel's congestion window size.

Instead of designing a new packet scheduler, we have used the scheduler introduced by TCP PRISM [48]. To the best of our knowledge, this is the only window-based packet scheduler that exists that has been shown to work well in scenarios similar to ours (a heterogeneous network environment). Even though the scheduling part of PRISM is built for TCP, it can be applied to any bandwidth aggregation approach based on window-based congestion control.

The PRISM scheduler provides a good technique for scheduling packets according to available path capacity, without requiring any a priori knowledge. The metric used for scheduling by PRISM is the path utilization. A high bandwidth path will support a larger congestion window than a low bandwidth path, and a tunnel established over a

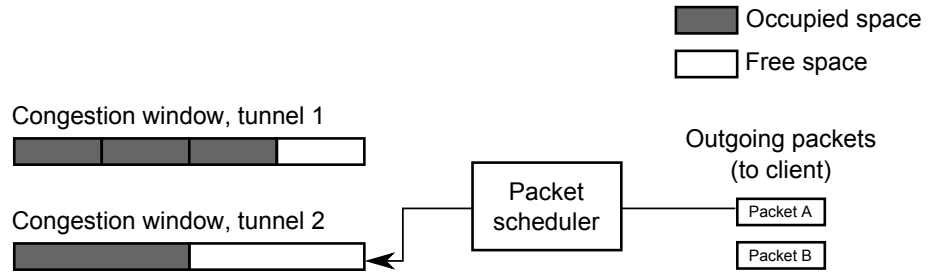
low RTT path will have packets removed from its congestion window faster than a tunnel established over a high RTT path. By dividing the number of packets currently in flight (i.e., not yet acknowledged) with the congestion window size, one gets the utilization of that path. By dividing each tunnel's congestion window size with the sum of the size of all congestion windows, a globally valid path utilization value is calculated. The scheduler picks the tunnel with the lowest utilization, i.e., the path with the most available capacity. If two or more paths have equal capacity, the least recently used tunnel is chosen. If every congestion window is full, the packet is dropped.

Algorithm 4 Packet scheduler

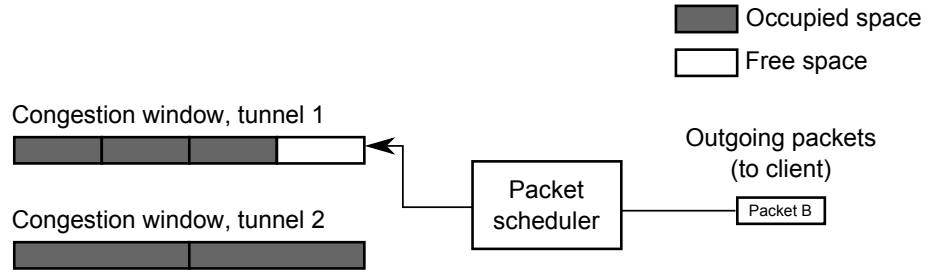
```
1:  min_utilisation = INFINITE
2:  scheduled_tunnel = None
3:  tunnels = [set of tunnels with an open congestion window]
4:
5:  if tunnels == Empty then
6:     drop packet
7:     return None
8:  end if
9:
10: for all tunnels do
11:    if utilisation_path[tunnel] <min_utilisation then
12:       min_utilisation = utilisation_path[tunnel]
13:       scheduled_tunnel = tunnel
14:    end if
15: end for
16: return scheduled_tunnel
```

The scheduling algorithm is summarized in algorithm 4. When a packet destined for the client arrives at the proxy, the proxy first initialises the required variables and creates a set containing all the tunnels with an open congestion window (line 1-3). If the set is empty, then the packet is dropped in order to avoid causing congestion (line 5-6). Otherwise, the tunnel with the lowest path utilization is chosen (line 10-16). The capacity metric is the global path utilization value for each tunnel (*utilisation_path[tunnel]*). This value is currently recalculated every time a scheduling decision has to be made received.

Figure 5.4(a) shows an example of how the packet scheduler works. A client equipped with two active network interfaces has established two tunnels to the proxy. Tunnel 1 has a congestion window of size four, while tunnel 2 has a window size of two. The number of unacknowledged packets in tunnel 1 is three, while tunnel 2 has one unacknowledged packet. When the proxy decides which tunnel to send packet A through, it first calculates the path utilisation. For the path corresponding to tunnel 1, this is $\frac{3}{4}$, while for tunnel 2 the utilisation is $\frac{1}{2}$. In order to get a globally valid utilisation value, the window size



(a) The proxy schedules packet A to go through tunnel 2, as this tunnel has most available capacity.



(b) The proxy schedules packet B to go through tunnel 1.

Figure 5.4: Snapshots of the packet scheduler.

has to be divided with sum of the window size of every tunnel. This gives $\frac{3}{\frac{4}{6}} = \frac{9}{2} = 4.5$ for tunnel 1 and $\frac{1}{\frac{2}{6}} = 3$ for tunnel 2. Because tunnel 2 has the lowest path utilisation (most available capacity), it is chosen by the scheduler. Assuming that no ACKs arrive before packet B will be sent from the proxy, tunnel 1 will be chosen because there is no free space in the congestion window for tunnel 2 (shown in figure 5.4(b).

Packet resequencer

Even though bandwidth aggregation increases the available bandwidth, it does not guarantee a higher throughput for the application. That packets arrive in-order is important because most applications process data sequentially. Therefore, a bandwidth aggregation technique should also support reducing the degree of reordering exposed to the higher layers.

In a multilink scenario, packet reordering is mainly caused by latency heterogeneity. With the bandwidth aggregation technique presented in this section, reordering is compensated for at the client by resequencing packets. Resequencing is a common approach and is for example used by MPTCP [36] and by the link layer technique introduced in [2]. The only difference between ours and the default packet resequencer behavior, is that we have added timers to avoid unlimited blocking. Because UDP is unreliable, there is no guarantee that the packet(s) that solves reordering will ever arrive. Thus, a timeout is needed to avoid infinite blocking.

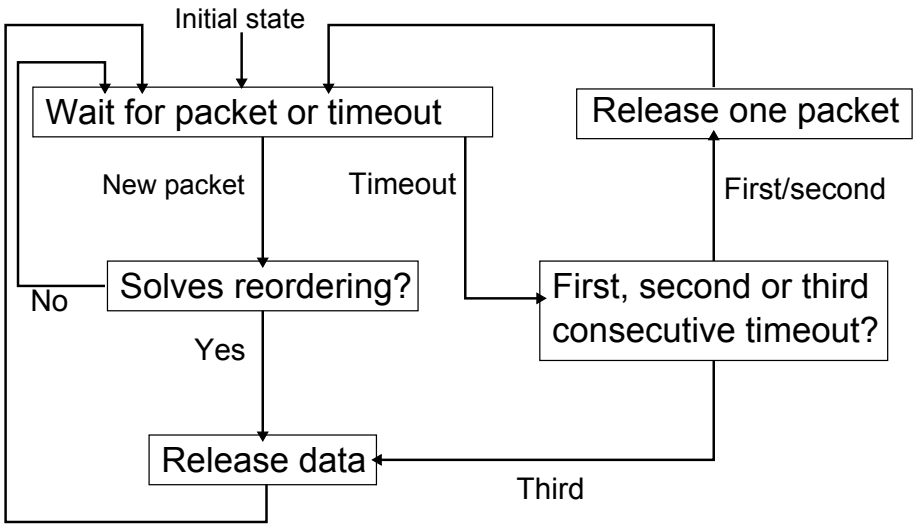


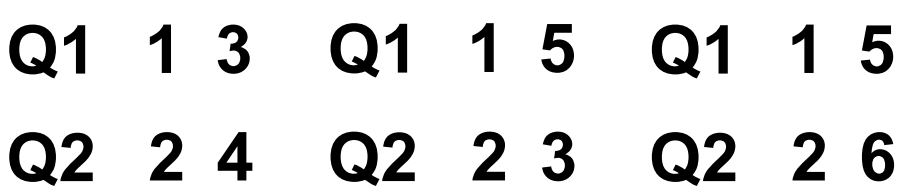
Figure 5.5: A state-diagram showing how the resequencer works

The resequencer is summarised in figure 5.5. When a packet arrives at the client, it is automatically passed to the resequencer. If the packet has arrived in-order, it is released from the buffer. Similarly, if the packet solves reordering, then the packet and other buffered in-order packet are released. How packet reordering is detected, will be explained later in this section. If a timeout is triggered while the resequencer is waiting for packets, the behavior depends on the number of consecutive timeouts.

The timeout used to avoid infinite waiting is calculated in the same way as TCP's Retransmission Timeout (RTO) [58], except that the one way delay (OWD) is used. The OWD is measured from the client and provides a good estimate for the worst-case time between a packet is sent and its reception at the client. When the timeout expires for the first and second time, one packet is released. When it expires for the third consecutive time, all packets held in the buffer are released. Waiting until the third timeout to release all buffered packets is done to reduce the probability of releasing large bursts of data. The timeout and consecutive timeout counter are reset for every in-order packet that arrives.

In order to detect reordering, the global sequence number (the sequence number that is valid across all the tunnels) is used. Packets are sent in-order from the proxy, and we assume that there is no internal reordering on a single path. This is a fair assumption, since the reordering in the backbone network is between 0.01 % and 1.65 % [32]. However, we would like to point out that the technique for detecting reordering also works in the presence of internal reordering.

The resequencer stores the largest global sequence number that has arrived for each tunnel. When a packet arrives, the resequencer selects the lowest of these values. This global sequence number is the largest sequence number the resequencer can assume is covered. That is, because UDP is unreliable, one can assume that no packets with a lower



(a) In-order (b) Out of order (c) Loss

Figure 5.6: The three states of the resequencer. Q1 and Q2 are the resequencing queues for two different interfaces.

global sequencer number will arrive later. Packets with a sequencer number larger than the one selected can not be released, as the sequence number(s) belonging to the gap can not be account for. The packets with the missing sequencer number(s) (i.e., covered by the gap) can either be lost or still be in transit (reordered). We would like to point out that if a path experiences internal reordering, the technique will still work as the largest global sequencer numbers are stored. If a packet with a lower global sequencer number arrives, it will just be released immediately.

An example of how reordering is detected is shown in figure 5.6. The implementation of the resequencer only maintains one packet queue. However, in order to provide a better explanation and illustration, we have used one packet queue per tunnel in the figure. The three different states shown in the figure can be explained as follows:

Figure 5.6(a) Here, every packet has arrived and is in order. Consequently, all packets are released by the resequencer.

Figure 5.6(b) In this state, the resequencer detects potential reordering. The highest sequence number the resequencer can assume is covered is three, meaning that packet 1, 2 and 3 will be released. Packet four cannot be accounted for, it can either be lost or reordered and still in transit, so packet five is held by the resequencer until a decision can be made. This figure also demonstrates the need for a timeout. If the sender for example has stopped sending data and packet four has been lost, then, without a timeout, packet five will never be released.

Figure 5.6(c) shows how the resequencer deals with packet loss. Packet loss has occurred on both paths (packet three and four are missing). However, because more packets have arrived on both paths, and they are in-order, all four packets will be released. In other words, if packet loss is detected, the missing global sequence number(s) is ignored.

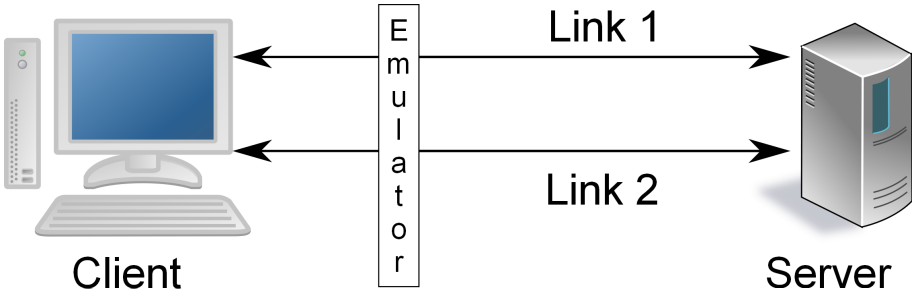


Figure 5.7: The controlled environment-testbed used to evaluate the performance of the transparent bandwidth aggregation technique for UDP.

5.1.2 Evaluation in a controlled network environment

Testbed

Our controlled network environment testbed consisted of three machines, shown in figure 5.7, each running Linux 2.6.31-14. The machines were connected directly to each other using 100 Mbit/s Ethernet, and one machine was used as both proxy and sender, the second emulated path delay (when needed), while the third was the multilink-enabled client and also where we limited the bandwidth. Our own tool was used to generate the constant UDP bitrate stream that was used to measure the performance of the technique. To emulate bandwidth and RTT, the network emulator *netem* [1] was used, together with the hierarchical token bucket.

Bandwidth aggregation

A good bandwidth aggregation depends on an efficient packet scheduler and correct congestion control. The proxy has to accurately estimate the capacity of each path and select the "correct" tunnel. Otherwise, if for example a slow path is prioritized, the full capacity of the faster paths will not be used. To measure how efficiently the proxy aggregates bandwidth, three sets of experiments were performed. For all the results presented in this subsection, the server sent a 10 Mbit/s UDP-stream to the client, and the sum of the available bandwidth at the client was always equal to 10 Mbit/s.

In the first series of tests, the proxy was faced with different levels of bandwidth heterogeneity (the RTT was not changed and was less than one 1 ms). The results are shown in figure 5.8. For each level of bandwidth heterogeneity, the proxy achieved close to the same bandwidth as a single 10Mbit/s link, and, hence, using multiple links gave a significant performance increase over a single link for all levels of heterogeneity. The congestion control accurately estimated the capacity of each path, which enabled the packet scheduler to make the right scheduling decisions.

[1] http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

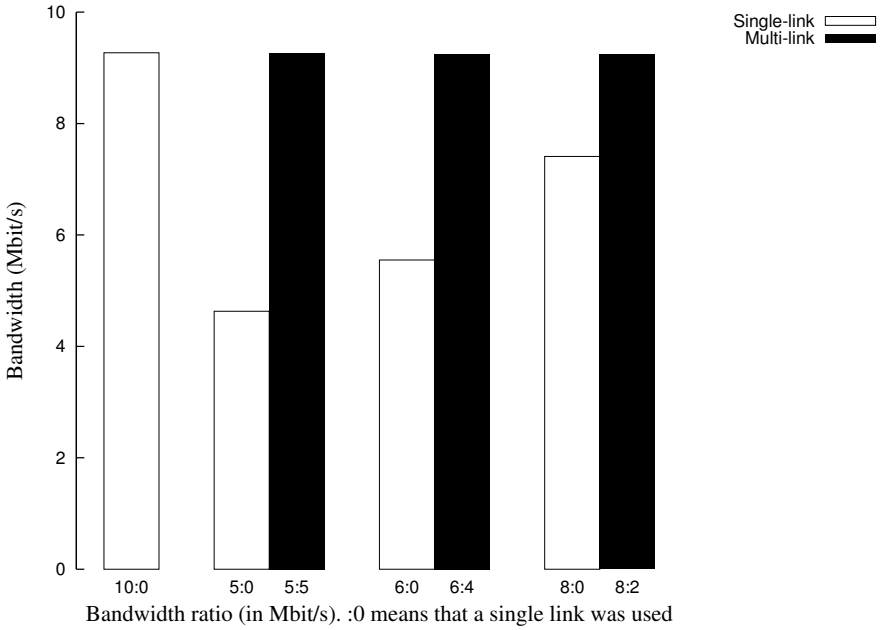


Figure 5.8: Achieved aggregated bandwidth with a constant 10 Mbit/s UDP stream and fixed bandwidth heterogeneity. The X:Y notation means that link 1 was allocated X Mbit/s and link 2 Y Mbit/s. :0 means that a single link was used.

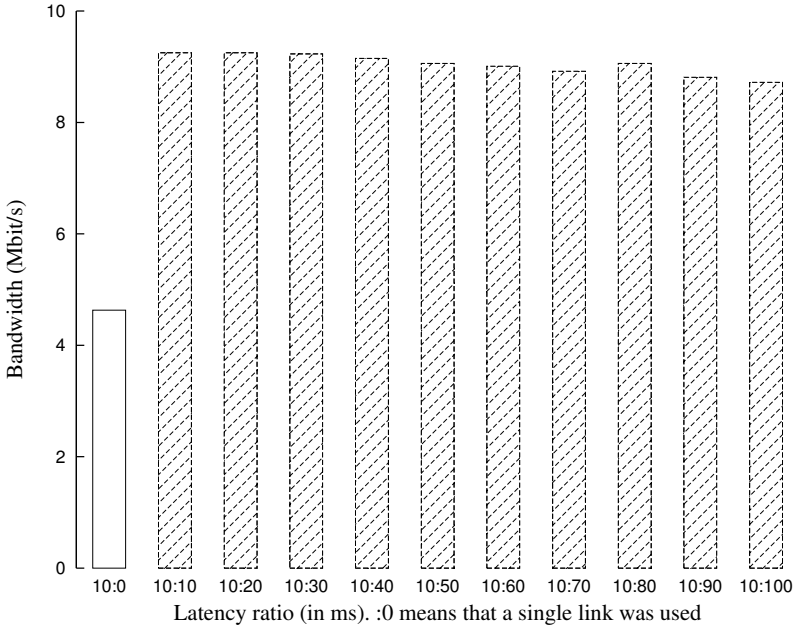


Figure 5.9: Achieved aggregated bandwidth with a constant 10 Mbit/s UDP stream and fixed latency heterogeneity. The X:Y notation means that link 1 had an RTT of X ms and link 2 Y ms. :0 means that a single link was used.

The purpose of the second series of tests was to see how latency heterogeneity affects the effectiveness of the bandwidth aggregation technique. Each link had a bandwidth of 5 Mbit/s, to avoid potential side-effects caused by bandwidth heterogeneity, while the RTT of one path was set to 10 ms, and the other assigned an RTT of r ms, with $r \in \{10, 20, \ldots, 100\}$. The results are shown in figure 5.9. The proxy significantly improved the performance compared to a single 5 Mbit/s link.

A small decrease in the aggregated bandwidth can be observed as the heterogeneity increased, indicating that the latency heterogeneity would at some point have an effect. This is as expected when congestion control is used. As the latency increases, the growth rate of the congestion window decreases due to the increased time it takes for feedback (the ACKs) to arrive. During the initial phase, or when congestion control has been invoked, the congestion window and thereby throughput will grow at a slower rate.

Finally, to get an impression of the effectiveness of the bandwidth aggregation technique when faced with dynamic link behavior, we used a modified version of the script described in section 4.3.1 to emulate dynamics. The bandwidth of the links was updated at a random interval, $t \in \{2, \ldots, 10\}$, and the sum of the available bandwidth was always equal to 10 Mbit/s. The RTT of path 1 was normally distributed between 0 ms and 20 ms, while the RTT of path 2 was uniformly distributed between 20 ms and 80 ms.

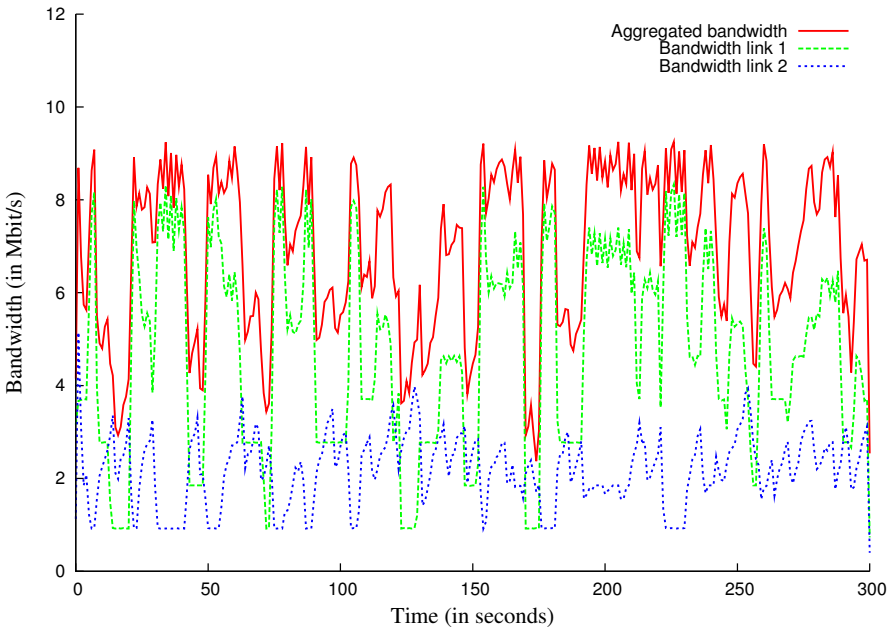


Figure 5.10: Achieved bandwidth aggregation with emulated network dynamics and a constant 10 Mbit/s UDP-stream. The bandwidth was measured every second.

The results are shown in figure 5.10. Using the bandwidth aggregation proxy resulted in a higher achieved bandwidth. The average aggregated bandwidth was 6.92 Mbit/s,

compared to 4.72 Mbit/s for the fastest the single link.However, even though the bandwidth was higher than that of the fastest single link, the difference was not as significant as in the other experiments. This was caused by the combination of bandwidth and latency heterogeneity, as well as the dynamic behavior. Due to frequent changes in available bandwidth, the client often experienced packet loss, and the size of the congestion window was reduced at the proxy, reducing the allowed send rate. In addition, the RTT affected the growth of the congestion window, causing the bandwidth to increase more slowly than with lower RTTs. The size of these drops depends on the RTT. With a high RTT, it will take longer for the proxy to get acknowledgements and, thus, adjust the congestion window for each tunnel.

Throughput gain

Increasing the throughput is important for most applications, as they process data sequentially. Reducing reordering is the responsibility of the resequencer at the client. For measuring the throughput, four sets of tests were run. One for different levels of bandwidth heterogeneity, one for different levels of latency heterogeneity, one where different bandwidth and latency heterogeneities were combined, and one were we emulated link dynamics.

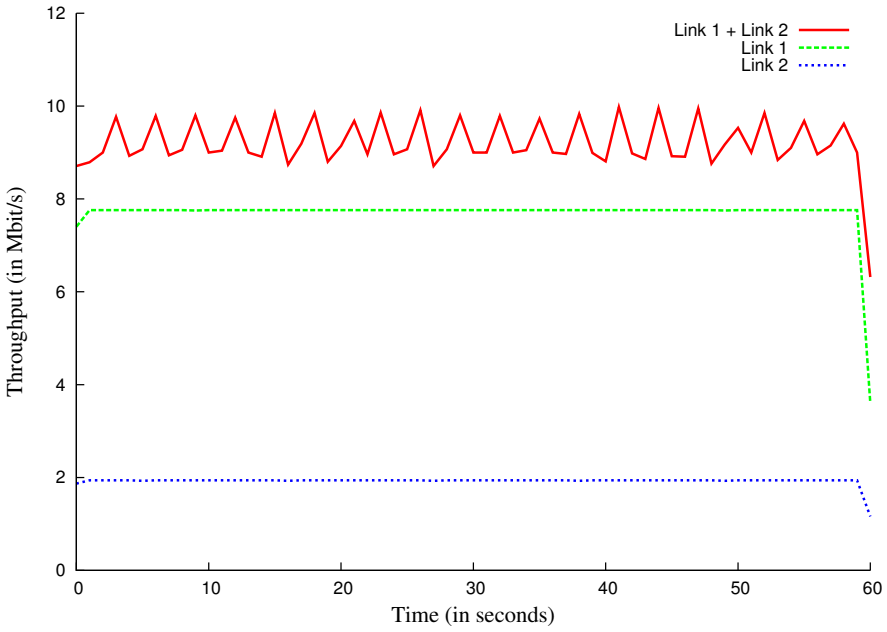


Figure 5.11: Achieved aggregated throughput with a bandwidth ratio of 8 Mbit/s:2 Mbit/s (equal RTT) and a constant 10 Mbit/s UDP stream. The throughput was measured for every second. The spikes are caused by data being released in bursts because of the heterogeneity.

One representative sample of the throughput for the most severe bandwidth hetero-

geneity, 8:2, is shown in figure 5.11. The throughput increased significantly, however, a bursty pattern can be observed. This is caused by the bandwidth heterogeneity. Due to the difference in capacity, the tunnel established over the path with the most capacity was allocated a larger share of the packets. When a packet that solved reordering arrived over the slow path, the buffer often contained several packets waiting to be released to the virtual interface and the application, causing the spikes.

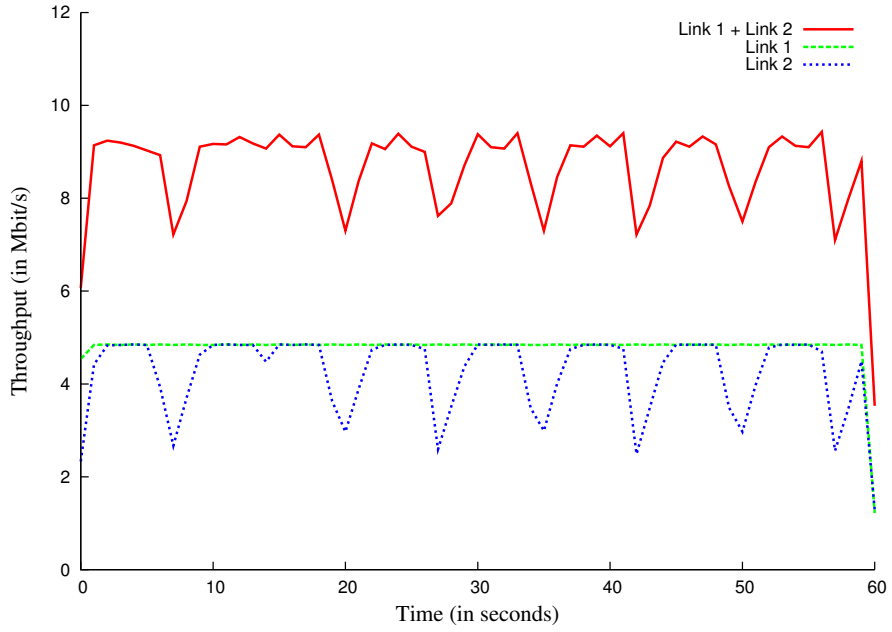


Figure 5.12: Achieved aggregated throughput with a latency ratio of 10 ms:100 ms (equal bandwidth) and a constant 10 Mbit/s UDP stream. The throughput was measured for every second. The drops in throughput are caused by the slower growth rate of the congestion window on the high RTT link

Figure 5.12 displays the measured throughput for a case of worst-case latency heterogeneity (10ms:100ms). As with bandwidth heterogeneity, the throughput was significantly better than that of a single link. However, a distinct pattern can be seen also in this graph. When the path with the highest RTT got congested (due to the congestion window reaching its maximum capacity) and the proxy invoked congestion control, it took a significant amount of time before the aggregated throughput grew back to the previous level. As with the decrease in aggregated bandwidth, this was caused by the increased RTT affecting the growth rate of the congestion window. The reason there were no significant throughput spikes, is that the bandwidth was homogeneous (in order to isolate the effect of latency heterogeneity), so close to the same number of packets were sent through each tunnel. Thus, there were few out-of-order packets in the buffer.

In figure 5.13, we show the results from one experiment with the combination of the worst-case bandwidth and latency heterogeneity. A more bursty traffic pattern can be

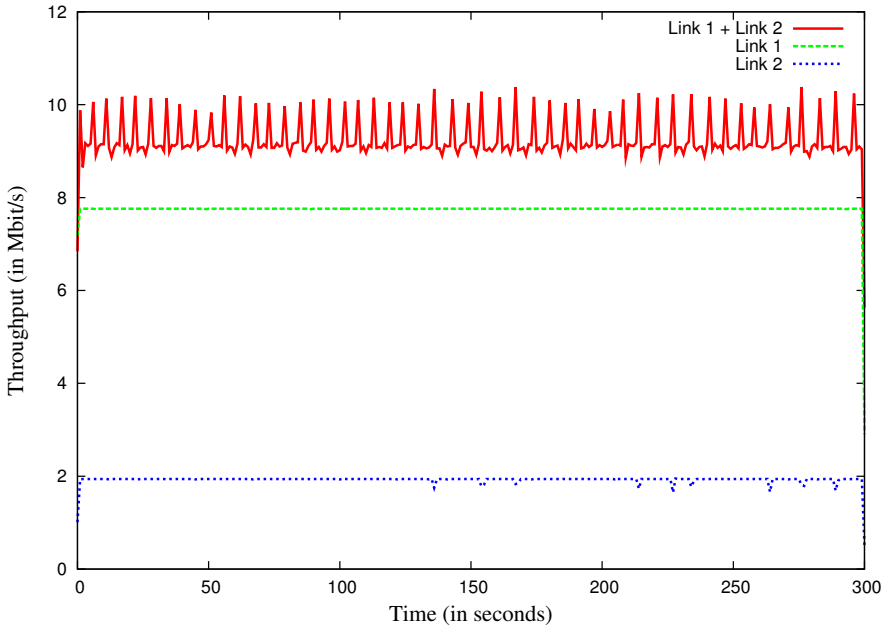


Figure 5.13: Achieved aggregated throughput with a combination of bandwidth and latency heterogeneity (8 Mbit/s, 10 ms RTT and 2 Mbit/s, 100 ms RTT), and a constant 10 Mbit/s UDP stream. The throughput was measured for every second.

observed, which was caused by the low bandwidth path also having the highest RTT. Reordering occurred more frequently, and when a packet that solved reordering arrived over the slow path, more packets were waiting in the out-of-order buffer than in the tests with only bandwidth heterogeneity. The reason for the lack of the throughput drops seen in figure 5.12, was that less traffic was sent through the tunnel established over the high RTT path. When looking at the logs, we see that the drops were present, however, they are less visible in the graph.

Finally, figure 5.14 displays the achieved throughput with emulated network dynamics for one representative sample, using the same script and parameters as in section 5.1.2. As with bandwidth aggregation, adding a second link increased the performance. However, also here, the performance gain was not as significant in the other experiments. This was, again, caused by the combination of bandwidth and latency heterogeneity, as well as the dynamic behavior. The total average aggregated throughput was 6.88 Mbit/s, compared to 4.60 Mbit/s for the single fastest link.

One difference between the bandwidth aggregation and aggregated throughput, was that the throughput was significantly more bursty. This can also be observed by comparing figure 5.10 and 5.14 The burstiness was, in addition to the dynamic link behavior, caused by the resequencer. Packets would frequently be buffered longer in the resequencer, causing the throughput drops. When reordering was resolved, a larger number of packets were released to the higher layer, causing a spike. In order to avoid the spikes and drops

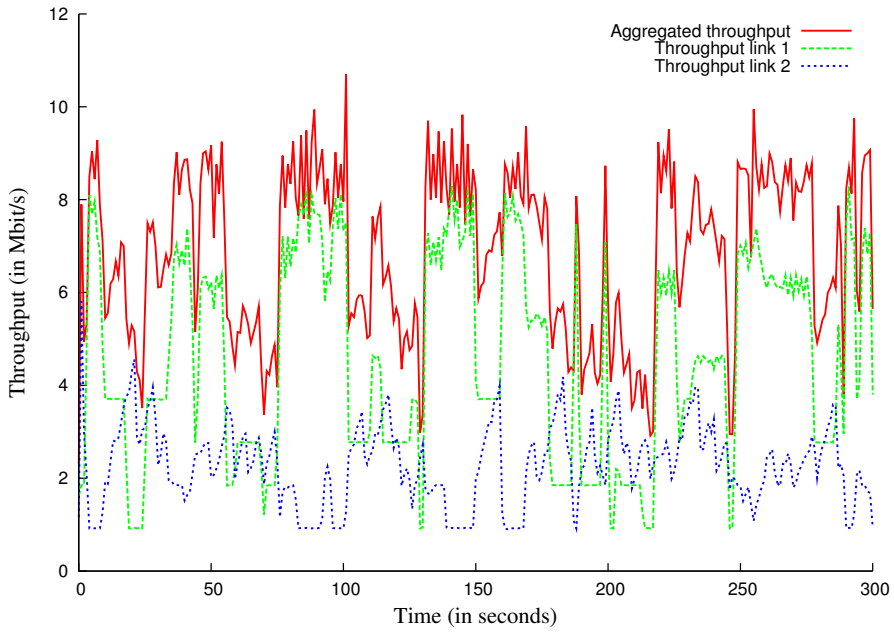


Figure 5.14: Achieved aggregated throughput with emulated network dynamics and a constant 10 Mbit/s UDP stream. The throughput was measured every second.

in throughput caused by the resequencer, a smoothing filter could have been applied. By for example adding a second buffer to ensure a smooth delivery of data, one could compensate for the bursty delivery of data to the higher layer.

5.1.3 Evaluation in real-world networks

Testbed

To get an impression of how our bandwidth aggregation technique performs in real-world networks, and in the presence of dynamic bandwidth and latency heterogeneity, we also measured the performance when the client was connected to one public WLAN (placed behind a NAT) and one HSDPA-network. The specified bandwidth and average measured RTT of the networks were 4 Mbit/s / 25 ms and 2.5 Mbit/s / 60 ms, respectively. The same application was used to generate the network traffic as in the other experiments.

Throughput gain

As in the experiments performed in the controlled network environment, the sender sent a 10 Mbit/s stream to the client. The average aggregated bandwidth when the client was connected to the real-world networks was 5.52 Mbit/s, which is a significant improvement over using only WLAN (which measured an average of 3.34 Mbit/s).

The measured throughput for one experiment is shown in figure 5.15. As can be

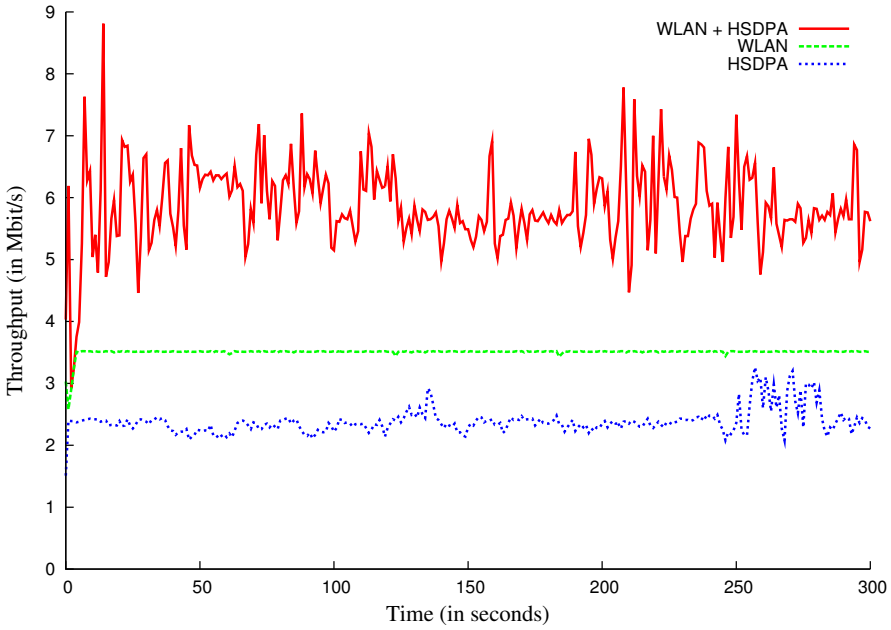


Figure 5.15: The aggregated throughput experienced in real-world networks. The sender sent a constant 10 Mbit/s UDP stream and the throughput was measured for every second.

seen, the aggregated throughput was better than that of the single, fastest link, and this observation is valid for all the experiments. Also, the throughput was significantly more bursty than in the experiments performed in the controlled network environment. This was caused by the dynamic behavior of the links, as well as the combined heterogeneities. A higher number of out-of-order packets will often be buffered at the client, so when reordering is resolved, a larger amount of data will be delivered to the application.

5.1.4 Summary

In this section, we have presented a technique for how to use multiple links to improve the performance of UDP-streams. Unlike the related work we are aware of, our technique can be used in real-world networks and requires no changes to existing transport protocols, applications or infrastructure. The only requirement is that a globally reachable proxy must be present, in order to stripe traffic across the multiple paths to a client. In order to avoid overflowing the paths and get a measurement of the current path capacity, we have implemented the window-based CCID2 congestion control. CCID2 is a TCP-like congestion control and a congestion window is maintained for each tunnel. Packets are sent over the path with the most available capacity, i.e., the tunnel with the most free space in its congestion window.

Most applications process data sequentially, thus, efficient bandwidth aggregation is not always sufficient. Packet reordering, caused by for example latency heterogeneity,

must be considered as well. In order to compensate for the reordering caused by the latency heterogeneity, and improve the throughput, we use a resequencer. The resequencer buffers packets until reordering is resolved or a timeout expires, in order to avoid deadlocks or head of line blocking.

Our technique was implemented together with MULTI running in invisible mode. In a controlled network environment and with real-world networks, our technique provided efficient bandwidth aggregation and increased the throughput significantly compared to that achieved by the single fastest link.

5.2 Transparent bandwidth aggregation of TCP-streams

TCP is, together with UDP, the most common transport protocol in use today. Unlike UDP, TCP is reliable and guarantees in-order delivery of data to the application. It performs congestion control to ensure a fair usage of the network, and flow control to avoid sending packets faster than they can be processed at the receiver. The transparent bandwidth aggregation technique presented in this section is designed based on the same core principles as the UDP-technique. A proxy is used to stripe packets according to path capacity, while a resequencer at the client compensates for reordering by buffering packets. However, in order to support the specific features of TCP, a core change was needed.

Because TCP itself does congestion control, a load balancing scheme similar to what was presented in the previous section will not work. Prior knowledge about each path's capacity is needed, otherwise, it will affect the growth of the TCP connection's congestion window and thereby the send rate (see also figure 1.6 and the discussion on the effects of bandwidth heterogeneity in section 1.2.2). If we assume that the paths have different bandwidth and equal RTT (for simplicity), the packet scheduler used to aggregate bandwidth for UDP-streams will behave like round-robin when faced with TCP traffic. Each corresponding tunnel's congestion window will grow at the same rate, as the sender receives ACKs and the TCP connection's congestion window grows and the send rate increases. However, as soon as the path with the lowest bandwidth becomes congested, TCP will invoke congestion control. This will affect the performance of the whole transfer, and the full capacity of the other paths will not be utilized. The maximum possible size of the congestion window at the sender is equal to the number of links times the maximum congestion window of the lowest bandwidth path. This is similar to the discussion in the introduction about the challenges introduced by bandwidth heterogeneity.

Instead of passive capacity measurements, for example active probing can be used. By actively measuring the available capacity of each path, the proxy can adjust the weight of

the paths on the fly and ensure a correct load balancing of the traffic. Because TCP itself guarantees fairness, there is no need for the proxy to check that this principle is upheld.

In the rest of this section, we will first introduce the active probing technique. Then, we continue with presenting the results from our evaluations. With TCP, it makes no sense to look at the achieved bandwidth, as this does not imply a better performance for the connection. TCP requires packets to be in-order and any lost data to have arrived before data is released to the application. Therefore, the evaluations are only focused on the achieved throughput.

Based on the evaluations, we discovered that the performance of transparent TCP bandwidth aggregation depends on the latency heterogeneity. Because of TCP's default behavior, we have not been able to design a transparent bandwidth aggregation to improve the performance of a TCP stream in the presence of severe latency heterogeneity. Instead, we present the design of a semi-transparent bandwidth aggregation technique that will not be affected by latency heterogeneity. Finally, we summarize and conclude our TCP work.

5.2.1 Architecture

The core architecture for the transparent TCP-based bandwidth aggregation technique is the same as for UDP. The technique is designed around MULTI being used in invisible mode. A proxy is responsible for distributing packets amongst the different tunnels that make up the multilink overlay network, while a resequencer at the client compensates for reordering by buffering packets until the in-order criteria is fulfilled, or packet loss is detected.

The only significant change compared to the UDP-technique is that the packet scheduler has been replaced. TCP is not compatible with passive load balancing, and our technique instead makes use of active probing in order to determine path capacity. The rest of this subsection describes the new packet scheduler.

Send vector-based packet scheduling

In order to achieve efficient transparent bandwidth aggregation of TCP streams, a priori knowledge of the capacity of each path is needed. The send rate of a TCP connection depends on the size of and the free space in the congestion window. Inaccurate scheduling of packets will lead to congestion, which in turn causes TCP to perform congestion control and reduce the send date.

To get an estimate of the capacity of each path, the proxy sends a UDP packet train [42] through each tunnel every second. By sending several equal sized packets back-to-back,

and measuring the inter-arrival time between the first and last packet in the train, the current available bandwidth can be calculated. The reason for using a packet train rather than a packet pair (introduced in [39] as packet dispersion), is that the estimations are more accurate. By sending a single packet pair, i.e., a packet train with a length of two, queueing delays in the network and cross-traffic can have a more significant effect on the estimations.

There exists several different packet scheduling disciplines, however, few of them can be applied to our scenario. In order to achieve efficient path utilisation, the packet scheduler must take the measured bandwidth into consideration and then balance the traffic accordingly. To achieve this, we propose the use of Weighted Round Robin-scheduling (WRR). WRR generates a scheduling sequence, from now on referred to as the *send vector*, and each element in the set used to generate the vector is assigned a weight. In our case, the elements are the different tunnels and the weights the bandwidth measurements of the corresponding paths.

Algorithm 5 createSendVector(n, w_0, w_1)

Input: Vector length $n \in N > 0$ and weights $w_0, w_1 \in R_{\geq 0}$
Output: Send vector V of length n
1: $V = zeros(n)$; {initialize V with the ID of tunnel 1}
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: $\quad V(\lceil i/r \rceil) = ID\ of\ tunnel\ 2$;
6: **end for**

The send vector contains the order of tunnels (identified by their unique IDs) to be picked and is used to decide the forwarding destination of incoming packets at the proxy. A pointer into the send vector is incremented after each lookup and reset when the end is reached. During initialization, the send vector V is set to emulate pure round-robin behavior (equivalent to WRR where the paths have equal weight), i.e., $V = \{0, 1, ..., m\}$ for m client interfaces. Once bandwidth estimations have been made, the send vector is updated accordingly. For the bandwidth estimates of two paths, which correspond to two weights w_0 and w_1, the send vector V is constructed as described in algorithm 5, a fair variant of weighted round-robin scheduling. Send vectors for more than two paths can be created by recursively merging send vectors created for two weights.

5.2.2 Evaluation in a controlled network environment

Testbed

The controlled network environment used to evaluate the performance of the technique for transparent bandwidth aggregation of TCP streams, was the same as for the UDP-based technique. Three machines, each running Linux 2.6.31-14, were connected directly to each other using 100 Mbit/s Ethernet. One machine was used as both proxy and sender, the second emulated latency (when needed), while the third was the multilink-enabled client. In order to measure the achieved throughput, the tool *wget* [2] was used to download the same file from our web-server. The network emulator *netem* [3] was used, together with the hierarchical token bucket, to emulate different levels of RTT and bandwidth heterogeneity.

Bandwidth heterogeneity

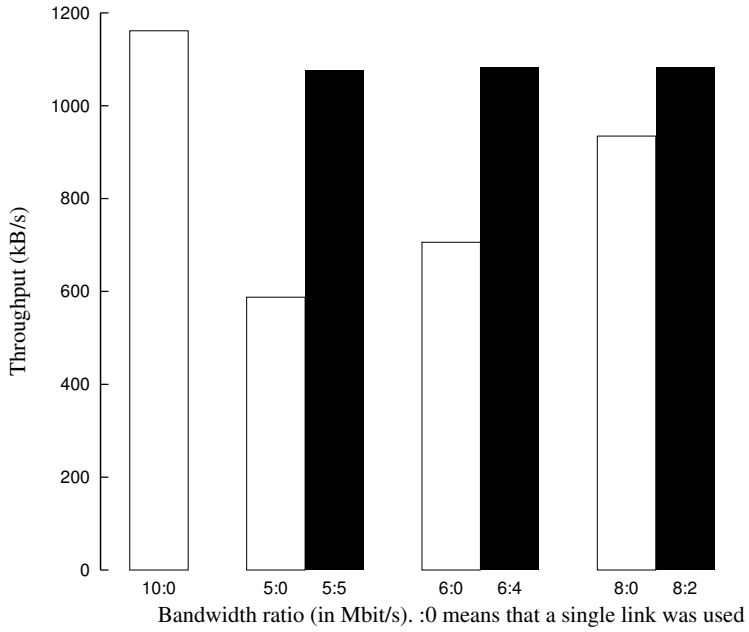


Figure 5.16: Average achieved aggregated throughput with TCP for fixed levels of bandwidth heterogeneity. The X:Y notation means that link 1 had a bandwidth of X Mbit/s, link 2 Y Mbit/s and :0 means that a single link was used.

Figure 5.16 shows the achieved throughput for different levels of bandwidth heterogeneity. As can be seen, the packet scheduler was able to efficiently estimate path capacity and stripe packets. The achieved throughput with aggregation was always significantly better than that of a single link. However, there was a slight performance difference

[2] http://www.gnu.org/software/wget/
[3] http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

between when a single 10 Mbit/s link and when bandwidth aggregation was used. This was caused by the finite length of the send vector, which was sometimes not sufficient to represent the ratio between the two paths with full accuracy. Because of this, paths sometimes got congested and the throughput was reduced for slight periods of time.

Latency heterogeneity

Latency heterogeneity causes packet reordering, which has a significant effect on TCP-performance. Because TCP assumes that packet loss is caused by congestion, reordering will trigger congestion control and cause a reduction in throughput. In order to avoid exposing TCP to the additional reordering caused by the latency heterogeneity, the technique presented in this section makes use of a resequencer. The resequencer was the same as in section 5.1.1 and buffers packets until reordering was resolved, loss was detected or a timeout expired. The timeout was used to prevent excessive waiting, which will trigger a retransmission timeout. As retransmission timeouts causes a connection to enter slow start again, it is more desirable to trigger some potentially redundant fast retransmissions.

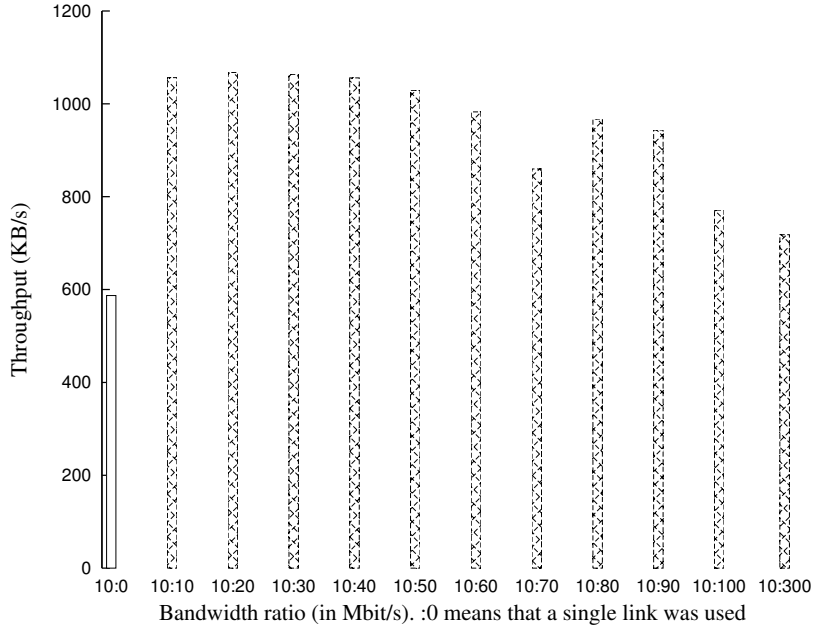


Figure 5.17: Average achieved aggregated throughput with TCP for fixed levels of latency heterogeneity. The X:Y notation means that link 1 had an RTT of X ms, link 2 Y ms and :0 means that a single link was used.

Figure 5.17 depicts the achieved throughput for different levels of latency heterogeneity. The bandwidth was limited to 5 Mbit/s on both links in order to avoid bandwidth heterogeneity affecting the results, and we see that the achieved throughput is higher than that of a single 5 Mbit/s link for all levels of latency heterogeneity. However, even though

there was some instability, the overall trend was that the gain offered by adding another link decreased as the latency heterogeneity increased.

The decrease was caused by the default behavior of TCP, together with the resequencer. When looking at the logs, we saw a large number of redundant retransmissions caused by the packets arriving over the low RTT path. That is, reordered packets that had not arrived yet were retransmitted. The only difference was that as the timeout value increased, so too did the number of RTOs. Also, increasing the time which packets were held by the resequencer, caused the congestion window to grow slower and delayed actual retransmissions because of the increase in time before feedback was sent from the receiver.

Another problem is TCP's behavior when deciding which packet to retransmit. TCP will only retransmit the first packet that has not been acknowledged. Thus, even if the resequencer detects loss, it is not necessarily the lost packet that is retransmitted. If the most recent acknowledged packet had sequence number X, then packet X+1 is the one that will be retransmitted. Even if the resequencer detects that it is X+2 that has been lost, and X+1 cannot be accounted for.

Due to TCP's default behavior, we have not been able to design a latency independent, fully transparent bandwidth aggregation technique for TCP. Out-of-order packets are interpreted as lost and causes TCP to trigger congestion control and reduce the send rate. Hiding reordering from the receiver by using the resequencer either causes redundant retransmissions (by making use of a time and releasing out-of-order data), or increases the number of RTOs. Also, because a TCP connection is end-to-end and not aware of the multiple links, congestion control affects the overall performance. Ideally, it should only affect the path that is congested. In other words, if two links were used, the send rate should only be reduced by one half, instead of one fourth. A semi-transparent technique that would achieve efficient bandwidth aggregation in the presence of severe latency heterogeneity, is one based on TCP connection splitting. This will be discussed in more detail in the next subsection.

5.2.3 Bandwidth aggregation using TCP connection splitting

The concept of TCP connection splitting, in literature also referred to as Split TCP [50], I-TCP [9] or TCP acceleration, was introduced in the early 1990's and is based on the indirect protocol model [8]. The main idea is that a TCP connection is split into multiple independent parts using a proxy/gateway. Connection splitting was originally designed to improve the performance of mobile hosts connected using wireless links. By inserting an intermediate node on the path between a server and the client, the node can buffer packets and implement scenario specific optimizations to improve the throughput at the client.

In order to avoid having to modify the server and make it aware of the connection splitting, the intermediate node establishes a connection to the server on behalf of the client [9]. Packets from the client are forwarded on this connection, and the node fakes an image of the client [8]. By using an intermediate node, the flow and congestion control of the fixed and wireless part of the connection are separated. This is desirable because of the often very different performance characteristics of the different types of links.

The send rate to the node is decided by its available downstream bandwidth and buffer space. For connection splitting to make sense, the node has to at least support the available bandwidth at the client. How the client behaves and communicates with the node depends on the scenario. It can for example use normal TCP and rely on a more aggressive retransmission scheme being implemented at the intermediate node, or use a proprietary protocol designed to maximize performance over a specific type of link.

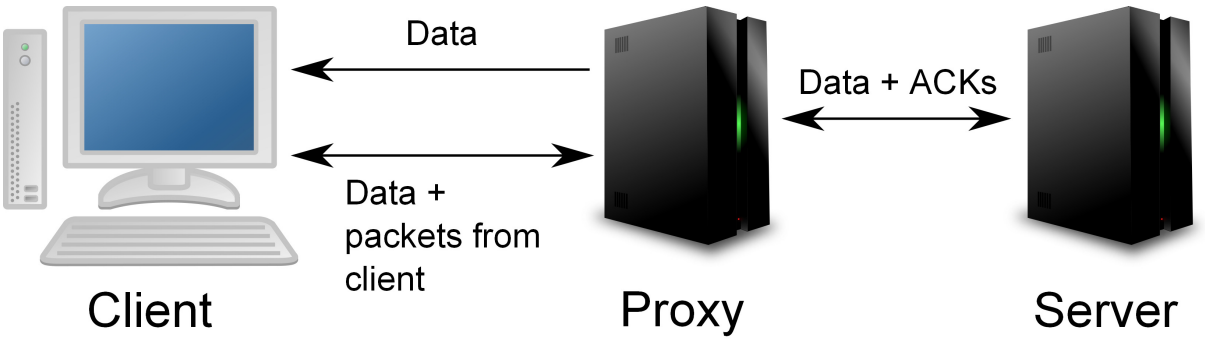


Figure 5.18: Example of the design of a bandwidth aggregation solution based on connection splitting.

An overview of how TCP-based bandwidth aggregation can be achieved using connection splitting, is shown in figure 5.18. The proxy (the intermediate node) fakes the behavior of the multihomed client. Any packet containing data sent from the client is forwarded, and data from the server is acknowledged and buffered at the proxy. The data will then be sent to the multihomed client according to any fit scheme. For example, the packet scheduler and congestion control discussed in section 5.1 can be extended with reliability and used to aggregate bandwidth.

At the client, packets can be buffered until reordering is resolved. Because the send rate at the server does not depend on feedback from the client, resequencing packets will not affect the throughput. Latency heterogeneity will still increase the time a packet is buffered by the resequencer, and thereby cause a more bursty release of data. However, it will not affect the send rate from the server, and, thus, the aggregated throughput is more robust against latency heterogeneity. A high RTT will still limit the throughput if

congestion control is used, as was discussed in section 5.1.2.

Even though connection splitting would allow for designing a transparent technique for TCP bandwidth aggregation, it is outside of the main focus of this thesis. By doing connection splitting, the end-to-end principle of TCP is violated and the behavior of the protocol modified. One of our goals was to design techniques for transparent bandwidth aggregation that was compatible with the default behavior of the protocols. Therefore, we have not performed any thorough analysis of bandwidth aggregation for TCP-streams based on connection splitting. However, in order to give an impression of the potential performance gain, we created a client and server application which emulates connection splitting.

Assuming that the proxy and server is able to support the aggregated bandwidth of the client, bandwidth aggregation using connection splitting is the equivalent of sending as much data over the paths as fast as possible. Our emulated proxy makes use of the send vector for packet scheduling, while the client establishes one TCP connection to the proxy for each active interface and acts as a sink for the received data. This is similar to what MPTCP [36] and SCTP CMT [38] does, except for the use of the send vector. The experiments were performed in the same testbed and for the same bandwidth and latency heterogeneities as in section 5.2.2.

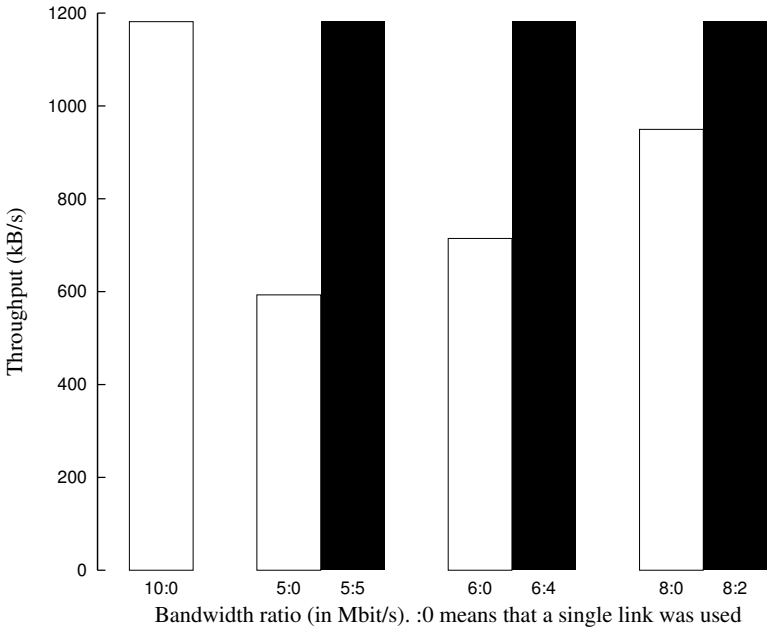


Figure 5.19: Average achieved aggregated TCP-throughput using emulated connection splitting for different levels of bandwidth heterogeneity.

Figure 5.19 shows the achieved, aggregated in-order throughput for different levels of bandwidth heterogeneity. As with the fully transparent TCP-based bandwidth aggregation, the send vector provided an accurate representation of available path capacity.

The aggregated throughput was close to the ideal value (10 Mbit/s) for every level of bandwidth heterogeneity.

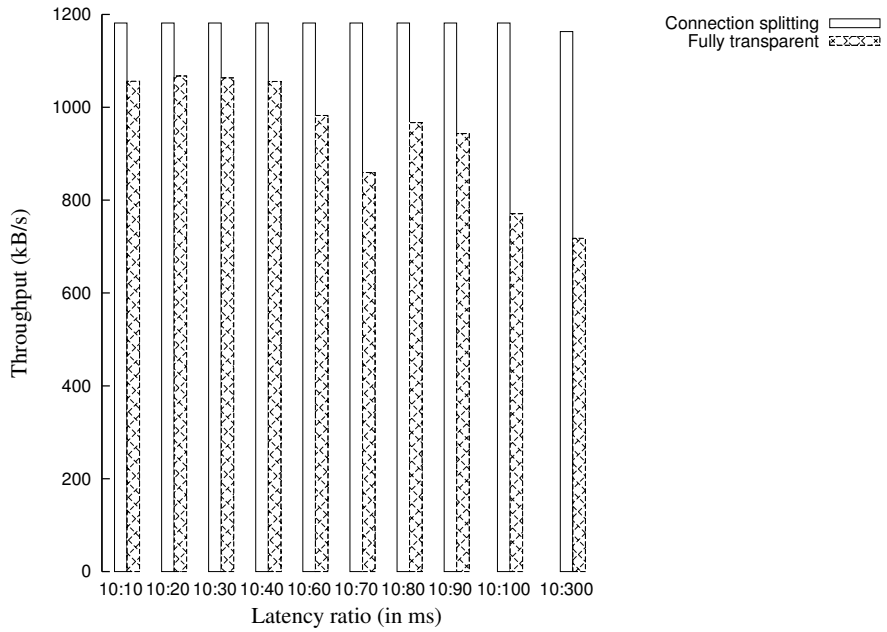


Figure 5.20: Average achieved aggregated TCP-throughput using connection splitting for different levels of latency heterogeneity, compared to the throughput achieved by the fully transparent technique.

The achieved throughput for different levels of latency heterogeneity is shown in figure 5.20, together with the results from the experiments done using the fully transparent aggregation technique. The throughput improved significantly and was much more stable than when fully transparent bandwidth aggregation was used. This was because the send rate from the server (the packet generation rate when the prototype was used) was no longer affected by the feedback generated by the client. Also, as the congestion control for the paths to the client are decoupled, a congestion event on one path no longer reduces the send rate of the overall connection. Finally, the effect of packet reordering has been removed. Because TCP is reliable, the packet(s) that solve reordering will arrive at some point in time. Thus, the resequencer never has to release packets early, and there are no redundant retransmissions. However, a slight decrease in the aggregated throughput can be seen as the heterogeneity increased. This is expected when TCP is used over a high RTT path, as it takes longer for a connection (the congestion window) to recover after packet loss has occurred.

5.2.4 Summary

The initial bandwidth aggregation technique for increasing the performance of TCP streams was based on the same core concepts as the UDP technique. The only difference was that active probing and a send vector were used to stripe packets, instead of a congestion window. This was done because TCP requires prior knowledge about the bandwidth of each path, and correct load balancing, in order for the congestion window to grow properly.

Unlike the UDP technique, the performance of the TCP-based bandwidth aggregation technique depended on the level of latency heterogeneity. TCP interprets out-of-order packets as packet loss and invokes congestion control, and hiding reordering completely caused a significant increase in the number of RTOs. Because of how TCP is designed, we have not been able to design a fully transparent, latency independent technique for bandwidth aggregation. When faced with severe heterogeneity, the throughput decreases and is in some cases worse than that of a single link.

Instead, we introduce a semi-transparent technique. By making use of connection splitting, the congestion controls are decoupled. The send rate from the server depends only on the feedback generated at the proxy. The TCP connection at the client acts as a sink, and any fit approach can be used to distribute the packets across the multiple links. Because connection splitting violates the end-to-end principle, it is outside the main focus of this thesis and did not perform any thorough evaluations of such a bandwidth aggregation technique. However, in order to give an impression of the potential performance gain, we emulated connection splitting and showed the achieved aggregated throughput for different levels of bandwidth and latency heterogeneity. As expected, a significant improvement over the fully transparent technique was seen in the latency heterogeneity experiments. The bandwidth heterogeneity experiments gave the same results as for the fully transparent technique.

5.3 Conclusion

Transparent bandwidth aggregation is desirable because bandwidth aggregation can be achieved without changing applications or transport protocols. In this chapter, we have presented techniques for how to increase the performance of UDP and TCP using multiple links. Both techniques were focused on downstream traffic and designed based on the same core concepts. A proxy was used to stripe packets and to avoid requiring changes to the server. To reduce reordering, a resequencer at the client buffered packets until reordering was resolved or a timeout was triggered. The timeout was used to avoid deadlock, head of line blocking and excessive waiting.

In order to balance the traffic correctly when used together with UDP-traffic, the TCP-like CCID2 congestion control was used. The bandwidth aggregation technique for UDP was able to utilize almost the full capacity of the paths independent of bandwidth and latency heterogeneity. The throughput also increased when a second link was added.

Because of the way TCP is designed, it requires prior knowledge about the capacity of each path in order for the congestion window to grow properly. Therefore, the feedback-based CCID2 was replaced with active probing and dynamic weighting of the paths, in order to enable the proxy to efficiently aggregate the bandwidth. The technique increased the performance when faced with bandwidth heterogeneity, but the performance depended on the level of latency heterogeneity.

Based on the results from our experiments and knowledge about TCP and its default behavior, we have not been able to efficiently aggregate the bandwidth without changing TCP in the presence of severe latency heterogeneity. A decoupling of the connection's congestion control from each path's congestion control is needed. TCP connection splitting supports this decoupling, and would allow for the design of a bandwidth aggregation technique that is more resilient to latency heterogeneity. However, such a technique is outside of the main focus of this thesis, as it violates the end-to-end principle of TCP. Still, in order to show the potential of a technique based on connection splitting, we emulated bandwidth aggregation using connection splitting. The achieved aggregated throughput was independent of bandwidth heterogeneity, and the performance for the different levels of latency heterogeneity was significantly better then for the fully transparent technique.

In the next chapter, we summarize and conclude the work presented in this thesis, as well as present our ideas for future work.

Chapter 6

Conclusion

Today, several different types of devices are multihomed. For example, smart phones can connect to 3G-networks and WLANs, while most laptops at least come equipped with LAN and WLAN interfaces. In addition, the coverage area of different wireless networks are ever-expanding. Thus, clients are often within the coverage area of multiple networks. Using multiple links, or network interfaces, simultaneously allows for desirable properties like increased network performance or reliability. In this thesis, we have investigated the potential of aggregating the bandwidth of different, heterogeneous links. Efficient bandwidth aggregation relies on solving a set of challenges. In addition to deployment challenges, bandwidth aggregation techniques must consider the link heterogeneity. For example, latency heterogeneity causes packet reordering, while bandwidth heterogeneity requires the traffic to be balanced accordingly. Also, especially wireless links typically displays fluctuating behavior, which must be compensated for. In this chapter, we summarise the thesis, as well as propose some ideas for future work.

6.1 Summary and contributions

The research presented in this thesis has resulted in three major contributions, as outlined in section 1.6: 1) We have developed a framework, called MULTI, that eases the development, deployment and evaluation of multilink solutions and techniques, and 2) designed and evaluated an application-layer technique for bandwidth aggregation, optimised for quality-adaptive video streaming, as well as 3) techniques for transparently improving the performance of UDP and TCP streams.

MULTI was motivated by our observation that no existing solution was able to meet our deployment requirements. For example, current solutions do not traverse NAT properly, rely on static configuration or add a significant overhead. MULTI is platform independent and solves the different challenges related to deployment. It works by monitoring

and automatically configuring the network subsystem of the operating system. It then exposes a continuously updated list of available interfaces to the applications making use of the components in framework (known as managers). The content of this list is depends on if MULTI is used to extend an application with multilink support directly, or used together with a transparent multilink solution. In the first case, the available interfaces are exposed to the manager directly. Otherwise, MULTI operates on the network layer and makes use of a proxy (to avoid changing the remote machine), and creates a multilink overlay network consisting of IP tunnels for each active interface. The available tunnels are then exposed and used by the managers.

The application-specific bandwidth aggregation technique was optimised for quality adaptive video streaming, and implemented on top of the common HTTP-protocol. When adding a second link, a significant increase in the visual video quality compared to using a single link was observed, both in the presence of bandwidth and latency heterogeneity. Also, the number of data delivery deadline misses was reduced, reducing the number of buffer underruns and meaning that the playback was smoother. However, the technique is not limited to HTTP or streaming. As long as the client is able to divide a file into smaller segments, so that the logical parts that can be requested over different links, the technique can be used to increase the performance of any bandwidth intensive application.

The two transparent bandwidth aggregation techniques presented in this thesis were targeted at improving the downstream performance of UDP and TCP-streams, the two most common transport protocols in use today. The techniques were based on the same core concepts and both operate on the network layer, were designed around the features offered by MULTI's invisible mode, and make use of a proxy to stripe traffic and avoid changing the server. In addition, both techniques rely on a resequencer to buffer out-of-order packets until reordering is resolved.

Where the two techniques differ, is in how the proxy distributes the traffic across the paths from the proxy to the client, as this depends on the behavior of the targeted transport protocol. Because UDP has no congestion control, the bandwidth aggregation technique makes use of the TCP-friendly CCID2 congestion control [28]. Applying congestion control is important because it ensures that the sender does not consume a too large share of the available resources. The amount of free space in each tunnel's congestion window is used by the packet scheduler at the proxy to select which tunnel to use. The congestion window represents the current capacity of the corresponding path, and the free window space represent the current available capacity. Adding a second link increased the performance of the UDP streams compared to using a single link. Almost the full capacity of each path was utilised, and the throughput was close to the maximum possible value (sum of all aggregated capacities), both in the presence of bandwidth and

latency heterogeneity.

Because of how TCP's own congestion window grows, prior knowledge about the available capacity of a path is needed. Otherwise, packets will be dropped when the lowest bandwidth path gets congested. This will cause the TCP connection's congestion control to be triggered and the throughput to be reduced. In order to increase the performance of a TCP stream using transparent bandwidth aggregation, we made use of active probing and weighted round-robin striping. UDP packet trains were used to measure the available bandwidth, and this information was used by the weighted round-robin scheduler to create a send vector. The send vector is a finite length vector containing the tunnels making up the multilink overlay network, sorted by the order in which they will be used. Using these techniques, we achieved an increase in performance when the technique was faced with bandwidth heterogeneity. However, the performance depended on the latency heterogeneity. The packet reordering caused TCP to trigger congestion control, due to packets being interpreted as lost, and reduce the send rate.

Because of the way TCP is designed, we have not been able to design a technique for transparent bandwidth aggregation that will work in the presence of latency heterogeneity. The reordering caused by latency heterogeneity will limit the growth of the congestion window, and thereby the throughput. The only way to efficiently aggregate bandwidth with TCP, is to decouple the connection's congestion control from the path's congestion control. One possible way to do this is to use connection splitting, and we present the design of a semi-transparent bandwidth aggregation technique based on this concept. The send rate from the server is decided based on the feedback generated by a node (the bandwidth aggregation proxy) inserted on the path between the client and server. This node "fakes" the behavior of a TCP client, and, in our case, any scheduler can be used to distribute the packets destined for the multihomed client. Connection splitting violates TCP's end-to-end principle and is outside the scope of this thesis. However, in order to get an impression of the possible performance gain, we emulated bandwidth aggregation based on connection splitting. The throughput was significantly more robust against latency heterogeneity, compared to the fully transparent bandwidth aggregation technique.

6.2 Concluding remarks

Users' appetite for bandwidth intensive services, like video streaming and moving content to and from cloud storage, shows no sign of slowing down. In addition, multihomed devices are becoming even more powerful and wireless networks capable of supporting higher bandwidths. For these reasons, we believe bandwidth aggregation both is and will

be an even more desirable property.

In this thesis, we have shown the potential of bandwidth aggregation. Bandwidth aggregation has been an active research field for several years. However, the related work has failed to consider challenges present in the real world networks. As we discovered through performing experiments and doing analysis, bandwidth and latency heterogeneity must be considered in order to do efficient bandwidth aggregation. Also, much related work ignore connectivity challenges. For example, NAT causes clients to be unreachable outside their own network.

There are several different approaches that can be used to enable bandwidth aggregation. One can for example modify or create new transport protocols, or design techniques optimised for one specific application or application type. During the work with this thesis, we built a framework which supports and eases the development of application specific and transparent, network layer bandwidth aggregation techniques. On top of this framework, we have designed one bandwidth aggregation technique optimized for quality-adaptive video streaming, as well as fully transparent techniques optimised for UDP and TCP. A network layer technique must support the behavior of the targeted transport protocols.

Each of the bandwidth aggregation techniques increased the throughput compared to using a single link. The application-specific technique, as well as the transparent technique targeted at UDP, utilised close to 100 % of the link capacity when faced with both bandwidth and latency heterogeneity, as well as real-world networks. The performance of the initial TCP technique depended on the latency heterogeneity. However, by splitting the connection, close to 100 % utilisation can be achieved also with TCP. Connection splitting violates the TCP end-to-end principle and has therefore not been a main focus of this thesis. A technique built on this concept requires no changes to either transport protocol or OS, and is therefore transparent to both sender and receiver.

After working on this thesis, we believe that the application and network layer are the most suited layers for implementing bandwidth aggregation. The application layer is controlled by developers and allows for unlimited flexibility. The network layer, on the other hand, allows for the design of fully transparent techniques that requires no changes to transport protocol or OSes. Transport layer/protocol changes take a long time to reach standardisation and deployment, while the link layer is limited by connectivity issues. Link layer techniques require interfaces to be connected to the same end point and be based on the same technology. This is not possible in our scenario, where a client's interfaces often use different technology and are connected to different networks.

6.3 Future work

We have presented several different techniques for how to efficiently aggregate bandwidth when a device has several links available. However, the field of bandwidth aggregation and use of multihoming in general is far from explored, and our ideas for future work includes:

- Several typical multihomed devices are mainly powered by battery, for example smart phones. Using multiple network interfaces simultaneously will consume more battery, reducing the time between when the device has to be charged. Evaluating the impact of bandwidth aggregation on battery life, as well as optimise techniques for devices with limited battery capacity, would increase the popularity and the probability of bandwidth aggregation being deployed. One possible idea is to limit the use of bandwidth aggregation to those cases where the client knows it will improve performance.

- The transparent bandwidth aggregation techniques presented in this thesis increases both bandwidth and throughput. However, we have not evaluated the effect bandwidth aggregation has on different types of applications. For example, a game might react completely different to an application streaming video. An interesting research task would be to look into these differences, and see if extending the aggregation techniques with a form for profile support (or similar) would improve performance.

- The bandwidth aggregation techniques presented in this thesis have been designed to be technology agnostic. That is, no assumptions have been made about the underlying technology. However, wireless technologies behave differently at the link layer and in the physical domain. For example, HSDPA has a more aggressive link layer retransmission scheme than for example WLAN, and is therefore more reliable at the expense of an increased latency. Also, wireless technologies can affect each other. For example, using two WLAN interfaces can cause interference, even if the interfaces are connected to different networks. Making a trade-off between technology independence and technology-specific optimisations might lead to even better performance.

- Another desirable property of multilink is the increased reliability offered by network handover. This has not been the focus of this thesis, however, it was used as an example of MULTI's invisible mode. Different handover approaches already exists, for example Mobile IPv6 [44] and Mobile SIP [76], but they, to the best of our knowledge, all rely on new protocols or protocol changes. We believe that transparent, application-layer handover has a large potential, and exploring this

field further would allow for the design of innovative, flexible and easily deployable handover techniques.

Appendix A

Publications

The work presented in this thesis has resulted in 10 peer-reviewed conference publications, one patent application and one journal article. This appendix contains summaries of each publication, as well as the contribution of each author.

A.1 Conference publications

Title: An Analysis of the Heterogeneity and IP Packet Reordering over Multiple Wireless Networks [47]

Authors: D. Kaspar, K. R. Evensen, A. F. Hansen, P. E. Engelstad, P. Halvorsen, and C. Griwodz.

Published: IEEE Symposium on Computers and Communications (ISCC), 2009.

Summary: This paper explores how link heterogeneity affects IP packet reordering. It also introduced the concept of the send vector (described in section 5.2.1).

Contributions: Dominik Kaspar was responsible for the performance evaluation and the writing. The other authors contributed with input and feedback on the text and the implementation/experiments.

Title: A Network-Layer Proxy for Bandwidth Aggregation and Reduction of IP Packet Reordering [18]

Authors: K. R. Evensen, D. Kaspar, P. E. Engelstad, A. F. Hansen, C. Griwodz, and P. Halvorsen.

Published: The 34rd Annual IEEE Conference on Local Computer Networks (LCN), 2009.

Summary: The work done for this publication was our first attempt at transparent bandwidth aggregation. We focused on improving the performance of downstream UDP-streams, and made use of a proxy to avoid changing the server. The proxy

striped packets according to the send vector, and a delay equalizer at the proxy delayed packets and thereby reduce reordering. The performance was evaluated inside a controlled network environment, and we achieved efficient bandwidth aggregation and saw a significant throughput increase. The work done in this paper served as the foundation for the work presented in chapter 3 and 5.

Contributions: Kristian Evensen was responsible for writing the code, while Dominik Kaspar was in charge of the writing the paper and creating the figures. The two were also both responsible for designing the technique and algorithms, as well as conducting the experiments. Every author contributed with feedback on the design and implementation, as well as contributing to the writing of the paper.

Title: Enhancing Video-on-Demand Playout over Multiple Heterogeneous Access Networks [46]

Authors: D. Kaspar, K. R. Evensen, P. E. Engelstad, A. F. Hansen, P. Halvorsen, and C. Griwodz.

Published: Consumer Communications and Networking Conference (CCNC), 2010.

Summary: In this paper, we explore how bandwidth aggregation can be achieved using the range-request feature of the common HTTP-protocol. The findings was the start of the work that is presented in chapter 4.

Contributions: The idea for using range-requests to request different parts of a file over multiple links came from Kristian Evensen, who was also responsible for implementing a prototype tool. Dominik Kaspar conducted all the experiments and was also the main author of the text. Every author contributed with feedback to the text, implementation and evaluation.

Title: Using HTTP Pipelining to Improve Progressive Download over Multiple Heterogeneous Interfaces [45]

Authors: D. Kaspar, K. R. Evensen, P. E. Engelstad, and A. F. Hansen.

Published: International Conference on Communications (ICC), 2010.

Summary: This paper was a continuation of the CCNC-paper and we analyzed how HTTP pipelining can be used to improve the performance of HTTP-based bandwidth aggregation.

Contributions: Dominik Kaspar and Kristian Evensen both contributed to the improvement of the prototype used for the CCNC-paper. Dominik Kaspar was also responsible for performing the evaluations, while each author participated in discussions about the content of the paper, as well as giving feedback.

Title: Quality-Adaptive Scheduling for Live Streaming over Multiple Access Networks [21]

Authors: K. R. Evensen, T. Kupka, D. Kaspar, P. Halvorsen, and C. Griwodz.

Published: The 20th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV), 2010.

Summary: In this paper, we applied the previous HTTP-based bandwidth aggregation techniques to quality-adaptive video streaming. This paper introduced the first version of the request scheduler and quality adaption mechanism (section 4.2.4) and the static subsegment approach (section 4.4). Evaluations were performed both in a controlled environment and with real-world networks, and a significant improvement in video quality was seen.

Contributions: Kristian Evensen and Tomas Kupka were responsible for designing the request scheduler, quality adaption mechanism and static subsegment approach. Kristian Evensen was also responsible for the implementation and evaluation. Dominik Kaspar, Kristian Evensen and Pål Halvorsen wrote the text, and every author participated in discussions about the content as well as provided feedback.

Title: Improving the Performance of Quality-Adaptive Video Streaming over Multiple Heterogeneous Access Networks [19]

Authors: K. R. Evensen, D. Kaspar, C. Griwodz, P. Halvorsen, A. F. Hansen, and P. E. Engelstad.

Published: Proceedings of the second annual ACM conference on Multimedia systems (MMSYS), 2011.

Summary: As discussed in section 4.4, the performance when the static subsegment approach is used depends on the size of the receive buffer at the client. In this paper, we introduced the dynamic subsegment approach (section 4.5) and compared the performance of the two approaches with three different types of streaming. The dynamic subsegment was able to utilise the links better and improved the video quality compared to the static approach.

Contributions: Kristian Evensen was responsible for the design of the technique, implementation, evaluation and most of the writing. Dominik Kaspar and Pål Halvorsen also contributed to the writing, as well as creating figures used for illustration. Every author provided feedback on the content.

Title: Using Multiple Links to Increase the Performance of Bandwidth-Intensive UDP-Based Applications [20]

Authors: K. R. Evensen, D. Kaspar, A. F. Hansen, C. Griwodz, and P. Halvorsen.

Published: Proceedings of IEEE Symposium on Computers and Communications (ISCC), 2011.

Summary: This paper was an improvement of our LCN-paper. The technique for bandwidth aggregation of UDP-streams presented here copes with the deployment challenges (especially NAT), in addition to compensating for reordering and scheduling

packets in ways which work with real-world networks. Most of the content in section 5.1 comes from this paper.

Contributions: Kristian Evensen was responsible for the design, implementation, evaluation and writing. Each author contributed with feedback on the text, as well as participating in different technical discussions.

Title: Mobile video streaming using location-based network prediction and transparent handover [23]

Authors: K. R. Evensen, A. Petlund, H. Riiser, P. Vigmostad, D. Kaspar, C. Griwodz, and P. Halvorsen.

Published: The 21th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV), 2011.

Summary: A desirable feature of having access to multiple links is transparent connection handover. For this paper, MULTI was used to implement this feature. In addition, we made use of a database containing bandwidth measurements of the different networks available at given co-ordinates. These two components were combined with an unmodified quality-adaptive video streaming system using path prediction to schedule which video quality to request when. Evaluations were performed along a tram-line in Oslo, which had a 100 % 3G coverage and a fast WLAN was available at two stops. In addition to sustaining the connection for the whole journey, a higher video quality was achieved when the client was able to use the WLAN when available. The system presented in this paper was used as the example of MULTI's invisible mode (section 3.4).

Contributions: Kristian Evensen was responsible for implementing transparent handover, as well as some of the writing. Andreas Petlund designed and configured the location-based database, while Haakon Riiser and Paul Vigmostad was responsible for performing the evaluations and analysing the findings. Each author contributed to the text.

Title: Demo: Quality-Adaptive Video Streaming With Dynamic Bandwidth Aggregation on Roaming, Multi-Homed Clients [22]

Authors: K. R. Evensen, A. Petlund, D. Kaspar, C. Griwodz, P. Halvorsen, H. Riiser, and P. Vigmostad.

Published: Proceedings of the 9th international conference on Mobile systems, applications, and services (MOBISYS), 2011.

Summary: For this demo, the video streaming client used in the MMSYS-paper [19] was extended with support for roaming clients, i.e., support for changes in link state. Participants could plug and unplug network cables, and the application aggregated the performance of the available links dynamically, increasing video quality when

possible. The implementation example of MULTI's visible mode served as the foundation for this demo (section 3.4).

Contributions: Kristian Evensen was responsible for the design, implementation, as well as most of the writing. Haakon Riiser presented the demo, while every author contributed to the text.

A.2 Journal articles

Title: Improving MPEG DASH-like systems using Multiple Access Networks

Authors: K. R. Evensen, D. Kaspar, C. Griwodz, P. Halvorsen, A. F. Hansen, and P. E. Engelstad.

Published: Accepted for publication in Signal Processing: Image Communication.

Summary: This paper is an extension of the MMSYS-paper [19], describing how the data retrieval technique presented in that paper can be implemented by clients supporting the coming MPEG DASH standard [1].

Contributions: Same as for MMSYS.

A.3 Patent applications

Title: "Data Segmentation, Request and Transfer Method", US Patent application (number 12/713,939), filed February 2010

Authors: D. Kaspar, K. Evensen, P. Engelstad, A. Hansen, C. Griwodz, and P. Halvorsen.

Summary: This patent application is based on the CCNC [46] and ICC [45] papers.

Contributions: Dominik Kaspar was responsible for most of writing, as well as communication with the patent experts Adam Piorowicz and Tom Ekeberg. The two pattern experts converted the content of the papers into proper "patent language". Each co-inventor participated in meeting and discussions about the content of the patent.

Bibliography

[1] MPEG DASH, ISO/IEC 23001-6 CD (N11578), 2011.

[2] H. Adiseshu, G. Parulkar, and G. Varghese. A reliable and scalable striping protocol. *SIGCOMM Comput. Commun. Rev.*, 26:131–141, August 1996.

[3] A. A. E. Al, T. Saadawi, and M. Lee. Ls-sctp: a bandwidth aggregation technique for stream control transmission protocol. *Computer Communications*, 27(10):1012 – 1024, 2004. Protocol Engineering for Wired and Wireless Networks.

[4] M. Allman, H. Kruse, and S. Ostermann. An application-level solution to tcp's satellite inefficiencies, 1997.

[5] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581 (Proposed Standard), Apr. 1999. Obsoleted by RFC 5681, updated by RFC 3390.

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

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

[8] B. Badrinath, A. Bakre, T. Imielinski, and R. Marantz. Handling mobile clients: A case for indirect interaction. In *Workstation Operating Systems, 1993. Proceedings., Fourth Workshop on*, pages 91–97. IEEE, 1993.

[9] A. Bakre and B. Badrinath. I-tcp: Indirect tcp for mobile hosts. In *Distributed Computing Systems, 1995., Proceedings of the 15th International Conference on*, pages 136–143. IEEE, 1994.

[10] R. Braden. Requirements for Internet Hosts - Application and Support. RFC 1123 (Standard), Oct. 1989. Updated by RFCs 1349, 2181, 5321, 5966.

[11] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), Oct. 1989. Updated by RFCs 1349, 4379, 5884, 6093.

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

[13] Cisco. Cisco EtherChannel Technology, 2003.

[14] D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young. Computing as a discipline. *Commun. ACM*, 32:9–23, January 1989.

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

[16] A. H. Eden. Three paradigms of computer science, 2007.

[17] K. Egevang and P. Francis. The IP Network Address Translator (NAT). RFC 1631 (Informational), May 1994. Obsoleted by RFC 3022.

[18] 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 N/A, editor, *The 34rd Annual IEEE Conference on Local Computer Networks (LCN)*, pages 585 – 592. IEEE, 2009.

[19] 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 K. M.-P. Ali C. Begen, editor, *Proceedings of the second annual ACM conference on Multimedia systems (MMSYS)*, pages 57–68. ACM, ACM, 2011.

[20] K. R. Evensen, D. Kaspar, A. F. Hansen, C. Griwodz, and P. Halvorsen. Using multiple links to increase the performance of bandwidth-intensive udp-based applications. In A. S. Elmaghraby and C. Douligeris, editors, *Proceedings of IEEE ISCC 2011*, volume 16, pages 1117–1122. IEEE, IEEE, 2011.

[21] K. R. Evensen, T. Kupka, D. Kaspar, P. Halvorsen, and C. Griwodz. Quality-adaptive scheduling for live streaming over multiple access networks. In D. C. A. Bulterman, editor, *The 20th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2010)*, pages 21–26. ACM, 2010.

[22] K. R. Evensen, A. Petlund, D. Kaspar, C. Griwodz, P. Halvorsen, H. Riiser, and P. Vigmostad. Demo: Quality-adaptive video streaming with dynamic bandwidth aggregation on roaming, multi-homed clients. In A. Agrawala, editor, *Proceedings of the 9th international conference on Mobile systems, applications, and services (MOBISYS)*, pages 355–356, New York, 2011. ACM, ACM.

[23] K. R. Evensen, A. Petlund, H. Riiser, P. Vigmostad, D. Kaspar, C. Griwodz, and P. Halvorsen. Mobile video streaming using location-based network prediction and transparent handover. In C. B. Krasic and K. Li, editors, *NOSSDAV*, pages 21–26, New York, 2011. ACM, ACM.

[24] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785.

[25] M. Fiore, C. Casetti, and G. Galante. Concurrent multipath communication for real-time traffic. *Comput. Commun.*, 30:3307–3320, November 2007.

[26] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582 (Experimental), Apr. 1999. Obsoleted by RFC 3782.

[27] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782 (Proposed Standard), Apr. 2004.

[28] S. Floyd and E. Kohler. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control. RFC 4341 (Proposed Standard), Mar. 2006.

[29] S. Floyd and E. Kohler. TCP Friendly Rate Control (TFRC): The Small-Packet (SP) Variant. RFC 4828 (Experimental), Apr. 2007.

[30] S. Floyd, E. Kohler, and J. Padhye. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC). RFC 4342 (Proposed Standard), Mar. 2006. Updated by RFC 5348.

[31] J. Funasaka, K. Nagayasu, and K. Ishida. Improvements on block size control method for adaptive parallel downloading. In *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04) - Volume 7*, ICDCSW '04, pages 648–653, Washington, DC, USA, 2004. IEEE Computer Society.

[32] L. Gharai, C. Perkins, and T. Lehman. Packet reordering, high speed networks and transport protocol performance. In *Proc. IEEE ICCCN*, pages 73–78, 2004.

[33] K. Hamzeh, G. Pall, W. Verthein, J. Taarud, W. Little, and G. Zorn. Point-to-Point Tunneling Protocol (PPTP). RFC 2637 (Informational), July 1999.

[34] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992 (Informational), Nov. 2000.

[35] H.-Y. Hsieh and R. Sivakumar. ptcp: An end-to-end transport layer protocol for striped connections, 2002.

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

[37] A. Inc. Mac OS X server – QuickTime streaming and broadcasting administration, 2007.

[38] J. R. Iyengar, P. D. Amer, and R. Stewart. Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths. *IEEE/ACM Trans. Netw.*, 14:951–964, October 2006.

[39] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM Computer Communication Review*, volume 18, pages 314–329. ACM, 1988.

[40] V. Jacobson and R. Braden. TCP extensions for long-delay paths. RFC 1072, Oct. 1988. Obsoleted by RFCs 1323, 2018.

[41] R. Jain. *The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling*. Wiley, 1991.

[42] R. Jain and S. Routhier. Packet trains–measurements and a new model for computer network traffic. *Selected Areas in Communications, IEEE Journal on*, 4(6):986–995, 1986.

[43] D. Johansen, H. Johansen, T. Aarflot, J. Hurley, Å. Kvalnes, C. Gurrin, S. Zav, B. Olstad, E. Aaberg, T. Endestad, H. Riiser, C. Griwodz, and P. Halvorsen. DAVVI: A prototype for the next generation multimedia entertainment platform. In *Proc. ACM MM*, pages 989–990, 2009.

[44] D. B. Johnson, C. E. Perkins, and J. Arkko. Mobility support in IPv6. RFC 6275, RFC Editor, Fremont, CA, USA, July 2011.

[45] D. Kaspar, K. R. Evensen, P. E. Engelstad, and A. F. Hansen. Using http pipelining to improve progressive download over multiple heterogeneous interfaces. In W. L. Fambirai Takawira, editor, *International Conference on Communications (ICC)*, pages 1–5. IEEE, 2010.

[46] 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 IEEE, editor, *Consumer Communications and Networking Conference (CCNC)*. IEEE, 2010.

[47] 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 ISCC, editor, *IEEE Symposium on Computers and Communications (ISCC)*, number CFP09SCC, pages 637–642. IEEE, 2009.

[48] K.-H. Kim and K. G. Shin. Prism: Improving the performance of inverse-multiplexed tcp in wireless networks. *IEEE Transactions on Mobile Computing*, 6:1297–1312, December 2007.

[49] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), Mar. 2006. Updated by RFCs 5595, 5596.

[50] S. Kopparty, S. Krishnamurthy, M. Faloutsos, and S. Tripathi. Split tcp for mobile ad hoc networks. In *Global Telecommunications Conference, 2002. GLOBECOM'02. IEEE*, volume 1, pages 138–142. IEEE, 2002.

[51] G. Malkin. Internet Users' Glossary. RFC 1983 (Informational), Aug. 1996.

[52] S. Mascolo, L. A. Grieco, R. Ferorelli, P. Camarda, and G. Piscitelli. Performance evaluation of westwood+ tcp congestion control. *Perform. Eval.*, 55:93–111, January 2004.

[53] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), Oct. 1996.

[54] Merriam-Webster. *Merriam-Webster's Collegiate Dictionary, 11th Edition*. Merriam-Webster, 2003.

[55] F. Miller, A. Vandome, and J. McBrewster. *IEEE 802.11*. VDM Publishing House Ltd., 2010.

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

[57] P. Ni, A. Eichhorn, C. Griwodz, and P. Halvorsen. Fine-grained scalable streaming from coarse-grained videos. In *Proc. ACM NOSSDAV*, pages 103–108, 2009.

[58] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988 (Proposed Standard), Nov. 2000.

[59] D. S. Phatak, T. Goff, and J. Plusquellic. Ip-in-ip tunneling to enable the simultaneous use of multiple ip interfaces for network level connection striping. *Computer Networks*, 43:787–804, 2003.

[60] J. Postel. User Datagram Protocol. RFC 768 (Standard), Aug. 1980.

[61] J. Postel. Transmission Control Protocol. RFC 793 (Standard), Sept. 1981. Updated by RFCs 1122, 3168, 6093.

[62] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Standard), Oct. 1985. Updated by RFCs 2228, 2640, 2773, 3659, 5797.

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

[64] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), Sept. 2001. Updated by RFCs 4301, 6040.

[65] H. Riiser, T. Endestad, P. Vigmostad, C. Griwodz, and P. Halvorsen. Video streaming using a location-based bandwidth-lookup service for bitrate planning (accepted for publication). *ACM Transactions on Multimedia Computing, Communications and Applications*, 2011.

[66] P. Rodriguez and E. W. Biersack. Dynamic parallel access to replicated content in the internet. *IEEE/ACM Trans. Netw.*, 10:455–465, August 2002.

[67] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2:277–288, November 1984.

[68] W. Simpson. The Point-to-Point Protocol (PPP). RFC 1661 (Standard), July 1994. Updated by RFC 2153.

[69] H. Sivakumar, S. Bailey, and R. L. Grossman. Psockets: the case for application-level network striping for data intensive applications using high speed wide area networks. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[70] K. Sklower, B. Lloyd, G. McGregor, D. Carr, and T. Coradetti. The PPP Multilink Protocol (MP). RFC 1990 (Draft Standard), Aug. 1996.

[71] A. C. Snoeren. Adaptive inverse multiplexing for wide-area wireless networks. In *GLOBECOM*, pages 1665–1672, 1999.

[72] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), Sept. 2007. Updated by RFC 6096.

[73] W. Townsley, A. Valencia, A. Rubens, G. Pall, G. Zorn, and B. Palter. Layer Two Tunneling Protocol "L2TP". RFC 2661 (Proposed Standard), Aug. 1999.

[74] B. Wang, W. Wei, Z. Guo, and D. Towsley. Multipath live streaming via tcp: Sche performance and benefits. *ACM Trans. Multimedia Comput. Commun. Appl.*, 5:25:1–25:23, August 2009.

[75] B. Wang, W. Wei, J. Kurose, D. Towsley, K. R. Pattipati, Z. Guo, and Z. Peng. Application-layer multipath data transfer via tcp: Schemes and performance tradeoffs. *Perform. Eval.*, 64:965–977, October 2007.

[76] E. Wedlund and H. Schulzrinne. Mobility support using sip. In *Proceedings of the 2nd ACM international workshop on Wireless mobile multimedia*, pages 76–82. ACM, 1999.

[77] Wikipedia. Comparison of Download Managers, June 2009.

[78] A. Zambelli. IIS Smooth Streaming technical overview. Technical report, Microsoft Corporation, 2009.

[79] M. Zhang, J. Lai, A. Krishnamurthy, L. Peterson, and R. Wang. A transport layer approach for improving end-to-end performance and robustness using redundant paths. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 8–8, Berkeley, CA, USA, 2004. USENIX Association.

[80] H. Zimmermann. OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.