Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Lehrstuhl für Intelligente Netze
und Management Verteilter Systeme

Multi-Path Aware Internet Transport Selection

vorgelegt von
Dipl.-Inform.
Philipp S. Tiesel
geb. in Berlin-Steglitz

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:
Vorsitzender: Prof. Dr. Rolf Niedermeyer, TU-Berlin
Gutachterin: Prof. Anja Feldmann, Ph. D., TU Berlin
Gutachter: Prof. Steve Uhlig, Ph. D, Queen Mary University of London
Gutachter: Prof. Olivier Bonaventure, Ph. D, Université catholique de Louvain

Tag der wissenschaftlichen Aussprache: 29. März 2018

Berlin 2019

DOI: https://doi.org/10.14279/depositonce-7830

This work was supported in part by the EU project CHANGE (FP7-ICT-257422)
and Leibniz Prize project funds of DFG (Leibniz-Preis 2011 – FKZ FE 570/4-1).

For my lovely monsters.

Abstract

When the Internet experiment started more than 30 years ago, no one could foresee its expansion — What started as an experiment to interconnect a few research computers has become an essential global infrastructure of humanity. Originally, there was usually only one way to transport data between two computers; this has changed dramatically — today's Internet offers us way more diverse transport options: Most end hosts are connected via multiple paths to the Internet, most content on the Internet is available on multiple servers, and there is a variety of transport protocols available to meet the needs of different applications.

In this thesis, we tackle the problem of *how to exploit Internet transport diversity to improve applications' performance*. More specifically, we discuss how to choose among transport options, how to realize multi-path aware transport option selection at the clients' operating system, and take a glimpse at the performance benefits we can achieve using transport option selection.

To reason about transport options and how to choose among them, we characterize three dimensions of *transport diversity* on the Internet: paths, endpoints, and protocol alternatives. We analyze a representative set of Internet Protocols with regards to the functionality and the granularity of control they provide and how they can be combined.

To realize multi-path aware transport option selection at the clients' operating system (OS), the OS needs to know what to optimize for. Therefore, we introduce the concept of *Socket Intents*; a means for applications to share their knowledge about their communication pattern and express performance preferences in a generic and portable way. We sketch a generic *policy framework* that enables the OS to *choose suitable transport options* while taking the interests of stakeholders like users, vendors and network operators into account.

To estimate the performance benefits we can achieve using path- selection, we analyze the benefits of using our *Earliest Arrival First (EAF) path selection strategy* for Web browsing. The EAF schedules the transfer over the path or path combination that minimizes the expected transfer time. We estimate the possible *performance benefits* in a custom simulator with a full factorial experimental design covering the Alexa Top 100 and Top 1000 Web sites and a small testbed study using our prototype and demonstrate significant performance benefits.

We demonstrate the implementability of path selection and endpoint selection, we implement our *Multi-Access Prototype* as an extension to the BSD Socket API. Our prototype enables connection reuse via implicit connection pooling and can control the path-management of MPTCP, but also reveals limitations originating by the BSD Socket API. Finally, we give an outlook of the challenges for deploying automated transport option selection within commodity OSes and underline the need for a replacement of the BSD Socket API.

Zusammenfassung

Zu Beginn des Internets konnte niemand dessen Erfolg erahnen. Was als Verbund weniger Großrechner in ausgewählten Forschungsinstituten begann, ist heute, über 30 Jahre später, zu einer für die ganze Menschheit wichtigen Infrastruktur geworden. Gab es zu Beginn des Internets meist nur eine Möglichkeit, bestimmte Inhalte zu beziehen, sehen wir uns heute mit einer Fülle verschiedener Optionen konfrontiert. So haben die meisten Endgeräte heutzutage mehrere Zugangswege zum Internet, z.B. Mobilfunk und WLAN, Inhalte werden von verschiedenen Quellen angeboten, und es gibt spezialisierte Protokolle für so ziemlich jedes Anforderungsprofil.

Diese Arbeit beschäftigt sich mit der Frage, wie man diese Vielfalt an Kommunikationsoptionen sinnvoll nutzen kann. Dabei konzentrieren wir uns auf drei Kernfragen: Die Auswahl aus den vorhandenen Kommunikationsoptionen, die Realisierung eines Auswahlmechanismus als Betriebssystemkomponente und die erzielbaren Performancegewinne.

Um die Vor- und Nachteile der einzelnen Kommunikationsoptionen im Internet gegeneinander abwägen zu können, charakterisieren wir ihre drei Dimensionen: Zugangsnetze, Gegenstellen und Protokollkombinationen. Wir analysieren die einzelnen Kommunikationsoptionen innerhalb ihrer Kategorie und analysieren verschiedene Protokolle auf Basis ihrer Funktionalität, ihrer Abstraktionstiefe und ihrer Kombinierbarkeit.

Um eine sinnvolle Auswahl zwischen den verfügbaren Kommunikationsoptionen im Betriebssystem zu treffen, muss das Betriebssystem wissen, woraufhin es optimieren soll. Dazu führen wir mit Socket Intents eine Abstraktion ein, die es den Anwendungen ermöglicht, ihr Wissen über ihre Kommunikationsmuster und ihre Präferenzen dem Betriebssystem auf entwicklerfreundliche Weise verfügbar zu machen. Darüber hinaus entwerfen wir ein Framework, das es ermöglicht, basierend auf diesem Wissen und den Anforderungen verschiedener Beteiligter, wie z.B. Nutzer, Anwendungsentwickler und Kommunikationsanbieter, geeignete Kombination auszuwählen.

Als Beispiel für eine Zugangsnetzauswahlstrategie stellen wir unsere EAF-Strategie vor, die für jede zu übertragenden Datei die Kombination an Zugangsnetzen auswählt, die die kürzeste Übertragungszeit verspricht. Wir evaluieren die aus dieser Strategie resultierenden Performancegewinne beim Websiteaufruf unter Verwendung eines Simulators für die Alexa Top 100 und Top 1000 Websites und unter Verwendung unseres Prototypen für eine kleine Websiteauswahl. Die Ergebnisse zeigen signifikante Performancesteigerungen in der Mehrzahl der betrachteten Szenarien.

Wir zeigen die Realisierbarkeit von automatischer Zugangsnetzwahl mit Hilfe einer prototypischen Implementierung auf Basis der BSD-Socketschnittstelle. Unser Prototyp erlaubt es, Verbindungen innerhalb von impliziten Verbindungspools wiederzuverwenden und die Pfadwahl von zu MPTCP beeinflussen, zeigt aber auch die von der BSD-Socketschnittstelle induzierten Limitierungen auf. Als Abschluss geben wir einen Ausblick darauf, wie diese Features in verbreitete Betriebssysteme Einzug halten können und erklären, warum ein Ersatz für die BSD-Socketschnittstelle dafür zwingend notwendig ist.

Acknowledgements

I first have to thank my spouse-monster Anja Tiesel for enabling me to finish this thesis by taking over most of the daily chores and having my back wherever she could. I thank her for staying with me through troubled times, illness and ever-changing life. I also have to apologize to my kid-monster Finn for having to bear me uselessly typing onto the computer keyboard instead of doing useful things like building wooden railway tracks. You are awesome and the most important thing in my life.

Next, I want to acknowledge my colleague Theresa Enghardt. Theresa joined my Multi-Access project early on for her master thesis about Socket Intents. Today, she is the major driver of the Multi-Access Prototype, our queen of policies and metrics, and a good friend. Thank you, Theresa, for sticking with the project and me despite all the complications.

I thank my advisor Anja Feldmann for giving me a chance to start from scratch with my Ph.D. after two years that nearly broke me. I thank her for keeping trust in me through difficult times, discussing and helping to refine all these strange ideas that lead to the Multi-Access projects and this thesis, and for all the support and feedback she gave me.

Special gratitude to Steve Uhlig, who helped me to get started at TU Berlin and was there to give me advice when I needed it most. Thanks to Ruben Merz for his inspiration leading me away from SDN for cellular backbones and towards the topic of this thesis. I also want to thank me Mirja Kühlewind and Brian Trammell for inviting me to the IAB SEMI workshop which got me into the IETF and therefore resulted in a new perspective on how to do network research.

Thanks to my collaborators Mirko Palmer and Ramin Kahili as well as to the students Patrick Kutter, Tobias Kaiser, and Bernd May — you all helped to make this happen. Also, I want to thank my colleagues Franziska Lichtblau, Doris Schiöberg, Florian Streibelt, Tobias Fiebig, Rainer May and our admin team for supporting and sometimes just bearing with me.

Finally, I have to thank my long and faithful friends, in particular, Robert S. Plaul, Leonie Kücholl, Jennifer Gabriel, and Cordelia Sommhammer for backing and supporting me whenever needed during the last strenuous years.

Publications & Collaborations

Parts of this thesis are collaborative work or are based on peer-reviewed papers that have already been published. All collaborators to this thesis are listed here — either as Authors of joined publications or with their kind of collaboration. Some publications were made under my name of birth *Philipp S. Schmidt*.

International Conferences

Philipp S. Schmidt, Theresa Enghardt, Ramin Khalili, and Anja Feldmann. "Socket Intents: Leveraging Application Awareness for Multi-access Connectivity". In: *ACM CoNEXT*. Santa Barbara, California, USA: ACM, 2013, pp. 295–300. ISBN: 978-1-4503-2101-3. DOI: 10.1145/2535372.2535405

Workshops

Philipp S. Schmidt, Ruben Merz, and Anja Feldmann. "A first look at multi-access connectivity for mobile networking". In: *Proceedings of the 2012 ACM workshop on Capacity sharing*. CSWS '12. Nice, France: ACM, 2012, pp. 9–14. ISBN: 978-1-4503-1780-1. DOI: 10.1145/2413219.2413224

Philipp S. Tiesel, Bernd May, and Anja Feldmann. "Multi-Homed on a Single Link: Using Multiple IPv6 Access Networks". In: *Proceedings of the 2016 Applied Networking Research Workshop*. ANRW '16. Berlin, Germany: ACM, 2016, pp. 16–18. ISBN: 978-1-4503-4443-2. DOI: 10.1145/2959424.2959434

Pre-Prints

Philipp S. Tiesel, Theresa Enghardt, Mirko Palmer, and Anja Feldmann. *Socket Intents: OS Support for Using Multiple Access Networks and its Benefits for Web Browsing*. Submitted to ACM/IEEE Transactions on Networking, initial version (June 2017) accepted with major revision, revised version (Apr. 2018) rejected. Apr. 2018. arXiv: 1804.08484

Internet Drafts

Philipp Tiesel, Theresa Enghardt, and Anja Feldmann. *Communication Units Granularity Considerations for Multi-Path Aware Transport Selection*. Internet-Draft draft-tiesel-taps-communitgrany-01. IETF Secretariat, Oct. 2017. URL: `http://www.ietf.org/internet-drafts/draft-tiesel-taps-communitgrany-01.txt`

Philipp Tiesel, Theresa Enghardt, and Anja Feldmann. *Socket Intents*. Internet-Draft draft-tiesel-taps-socketintents-01. IETF Secretariat, Oct. 2017. URL: `http://www.ietf.org/internet-drafts/draft-tiesel-taps-socketintents-01.txt`

Philipp Tiesel and Theresa Enghardt. *A Socket Intents Prototype for the BSD Socket API - Experiences, Lessons Learned and Considerations*. Internet-Draft draft-tiesel-taps-socketintents-bsdsockets-01. IETF Secretariat, Mar. 2018. URL: `https://www.ietf.org/archive/id/draft-tiesel-taps-socketintents-bsdsockets-01.txt`

Collaborations

The design and implementation of the Multi-Access Prototype described in Chapter 6 as well the formulation of the individual Socket Intents in Chapter 3 was done in tight collaboration with *Theresa Enghardt*. The Path Characteristics Data Collectors framework in Section 6.3.3 was done by *Theresa Enghardt*. The Multipath-TCP integration of the Multi-Access Prototype (Section 6.3.4) was done by *Mirko Palmer* [8]. The augmented name resolution API variant (Section 6.3.1.2) was part of the Bachelor Thesis of *Tobias Kaiser* [9] under my supervision.

The Web transfer Simulator used in Chapter 5 was joint work with *Mirko Palmer*.

Some ideas on how to structure policies in Chapter 4 are based on discussions with *Brian Trammell*, *Tommy Pauly*, *Mirja Kühlewind*, *Anna Brunstrom* and *Gorry Fairhurst* in context of the *IETF TAPS Working Group*.

Post-Published Work

The following Internet Drafts have been published after the initial version of this thesis was presented to the committee, but are reflected in several chapters of the final version as part of future work, outlook and conclusion.

Tommy Pauly, Brian Trammell, Anna Brunstrom, Gorry Fairhurst, Colin Perkins, Philipp Tiesel, and Christopher Wood. *An Architecture for Transport Services*. Internet-Draft draft-ietf-taps-arch-02. IETF Secretariat, Oct. 2018. URL: `https://www.ietf.org/archive/id/draft-ietf-taps-arch-02.txt`

Brian Trammell, Michael Welzl, Theresa Enghardt, Gorry Fairhurst, Mirja Kuehlewind, Colin Perkins, Philipp Tiesel, and Christopher Wood. *An Abstract Application Layer Interface to Transport Services*. Internet-Draft draft-ietf-taps-interface-02. IETF Secretariat, Oct. 2018. URL: `https://www.ietf.org/archive/id/draft-ietf-taps-interface-02.txt`

Anna Brunstrom, Tommy Pauly, Theresa Enghardt, Karl-Johan Grinnemo, Tom Jones, Philipp Tiesel, Colin Perkins, and Michael Welzl. *Implementing Interfaces to Transport Services*. Internet-Draft draft-ietf-taps-impl-02. IETF Secretariat, Oct. 2018. URL: `https://www.ietf.org/archive/id/draft-ietf-taps-impl-02.txt`

Contents

List of Figures

List of Tables

1 Introduction

Thirty years ago, most computers connected to the Internet where servers located at universities or research institutes. These hosts were shared by many concurrent users and usually had only a single connection to the Internet. Since then, the Internet underwent substantial changes. Today, almost everyone uses the Internet in some way. The players involved in the Internet and its structure changed: Instead of a few leased lines between research institutes, the Internet has become a complex ecosystem that includes thousands of (commercial) players, networks, and protocols.

In today's Internet, the predominant number of hosts are mobile devices with a single user: Smartphones, tablets, and laptops. These devices often have built-in interfaces for WiFi and cellular, whereby each of these interfaces typically provides at least one path to access the Internet.

In principle, applications could take advantage of the different characteristics of these access networks, e.g., delay, bandwidth, and expected availability, by choosing the path that meets the communication needs best:

- For video streaming applications like Youtube, bandwidth is most crucial.
- For voice calls packet loss and latency are important.
- Push notifications channels should be resilient and energy efficient.
- Software updates should afflict the lowest cost possible.

By using multiple interfaces at the same time, it is also possible to aggregate the bandwidth of multiple access networks or use the interfaces to gain redundant communication channels, e.g., to compensate the loss of connectivity when moving out of reach of one of the access networks.

However, when communicating over the Internet, there are more choices than just choosing among access networks. Let's review other options we have when initiating a connection: Each communication takes place between at least two endpoints, i.e., a host or an application. As a single host does not have the resources to serve popular content to a large user base, there are often several copies of that content distributed across mirror servers forming a content delivery networks (CDNs). Thus, if content is available from multiple servers, applications can choose among these when fetching the content.

To establish the connection, we also need a transport service: A set of protocols that implements the functionality the endpoints need to communicate through the Internet. There exist hundreds of diverse protocols providing transport services of different kinds, e.g., HTTP/1.1 over TLS over TCP over IPv4 or HTTP/3 over QUIC over UDP over IPv6. Depending on the applications' requirements, there may be multiple suitable combinations of protocols that can provide the transport services needed. Choosing a stack from these protocols completes the set of choices available.

These choices form the three dimensions of what we, throughout this thesis, call *transport diversity*: paths, endpoints, and protocol alternatives. Each path, endpoint, or element of a protocol stack available adds an option the endpoint can choose from. Thus, we call each of them a transport option. For each communication or transfer, an endpoint can choose a set of transport options which we then refer to as a *transport configuration*.

Endpoints that want to take advantage of transport diversity usually use their own heuristics for selecting a suitable transport configuration. This can be an extremely complex task, as each transport option comes with a bewildering set of trait-offs, e.g., regarding performance and guarantees. Determining these may require system privileges, historical knowledge, or active probing.

Traditionally, operating systems (OSes) usually use only one network interface at a time or allow fixing them on a per-application basis. Modern advances allow automatic and adaptive switching between interfaces, e.g., in case of weak WiFi reception[13] or add support for splitting TCP flows with MPTCP [14–16].

However, there is no way for the OS to precisely match the communication needs of an application to the most suitable set of transport options. One reason is that from the perspective the default programming interface for communication on the Internet, the BSD Socket API, all transfers look the same. For most applications, selecting the most suitable transport options is infeasible because of the complexity and the elevated privileges needed to access crucial performance information. Consequently, the available transport diversity is usually not exploited.

1.1 Problem Statement

The overarching question of this thesis is **how to exploit Internet transport diversity to improve applications' performance**. As this question is fairly general, we break it down into three questions that tackle individual aspects:

- **How to choose among transport options?**
- **How to realize transport option selection at the clients' OS?**
- **What are the performance benefits we can achieve using transport option selection?**

1.2 Contributions

To tackle the first aspect, we need to understand what transport options are available, determine their properties, and systematize them. We need to understand how we can combine transport options, i.e., destinations, paths, transport protocols and protocol options available, into transport configurations and find a way to assess and compare these transport configurations.

- By introducing the three dimensions of transport diversity and analyzing a representative set of Internet protocols, this thesis provides a basis to compose protocol stacks automatically.

In order to enable almost all applications to take advantage of transport diversity, exploiting it should be as automated and easy to use as possible. As we do not want to require applications to deal with each detail of the available transport options and do not want to introduce additional complexity, the network subsystem of the OS is the natural place to handle transport diversity. In the network subsystem, all communication, crucial performance information and, the complete interface configuration is visible. This unique position enables joint optimizations across application boundaries, including coordinated bandwidth management and sharing of protocol state. However, in order to choose the most suitable transport configuration for the application, we need to know what the applications intents to do — What kind of transfer is expected and what guarantees the application wants or needs — and find a representation that allows taking these intents as an input to transport option selection. Therefore, our contributions towards the second aspect of the problem statement are:

- With **Socket Intents**, we provide means for applications to share their knowledge about their communication pattern with the OS and express performance preferences in a generic and portable way.
- We sketch a **generic policy framework** that allows users, vendors and network operators to express their interests towards transport option selection. Within this framework, we provide a strategy to select suitable transport configurations based on the system state, the Socket Intents, and the different interests expressed by the different stakeholders. The best-ranked transport configurations chosen by the policy framework then compete in a connection-establishment race — Happy Eyeballs on Steroids — to choose the best transport configuration.
- We **demonstrate the feasibility of transport option selection within the OS** by implementing a working prototype on top of the BSD Socket API that realizes path selection and endpoint selection.

Finally, we need a strategy to choose the most appropriate transport options, combine them into transport configurations. This strategy needs to take the applications' intents, the available transport configurations, their properties and the requirements of other stakeholders into account. Therefore, to approach the third

aspect of the problem statement, we want to evaluate the potential of a concrete application aware strategy for a single use-case: Improving Web performance by using multiple paths.

- We provide a first application and path aware policy — the $\mathcal{EAF}$ **policy** — that improves Web performance.
- We **estimate the possible performance benefits in a custom simulator** using a full factorial experimental design covering the Alexa Top 100 and Top 1000 Web sites for a wide range of network characteristics.
- We **evaluate the $\mathcal{EAF}$ policy and our prototype** in a small testbed study using a few selected Web sites.

1.3 Structure of this Thesis

The initial part of the thesis is concerned with analyzing the transport diversity provided by the Internet and sketching the building blocks needed to realize transport option selection within the OS. In Chapter 2, we analyze the three dimensions of *transport diversity — path selection*, *endpoint selection*, and *protocol stack composition*. We introduce the notion of *communication units*, i.e., slices of a communication that have a semantic and can be distinguished at the respective layer, to reason about the granularity of communication at which transport option selection can be performed. Moreover, we analyze a set of mechanisms, i.e., functionalities that are provided by Internet protocols as part of their transports service. We use this terminology to analyze a representative set of Internet Protocols with regards to how they interact with transport option selection and on what granularity of communication units they can be used. In Chapter 3, we introduce the concept of *Socket Intents*. Socket Intents allow applications to express what they know about their communication patterns and preferences in a generic and portable way. The information collected about the possible paths, endpoints, and protocol stack compositions combined with the intents of the application provided using Socket Intents completes the basis needed for transport option selection within the OS. We present our generic policy framework that uses this information for choosing and ranking transport configurations in Chapter 4.

The second part of the thesis evaluates two aspects of transport option selection: The performance benefits that can be achieved and the feasibility of performing transport option selection within the OS In Chapter 5, we demonstrate the performance benefits of one prominent use-case for transport option selection: Improving Web browsing performance by combining two paths, distributing requests and by using different transport protocol compositions for the Web transfers. Chapter 6 presents our Multi-Access Prototype we use to evaluate the implementability of transport option selection as an extension to the BSD Socket API and do a preliminary performance evaluation using an *HTTP proxy* that uses our Multi-Access Prototype to path selection on a per HTTP request basis.

Finally, in Chapter 7, we conclude and discuss directions for future research.

2

Transport Options

Transport diversity — being able to choose transport options, including different paths, endpoints, and network protocols — is no advantage per se. Applications can benefit from performance improvements by choosing the "right" transport options, e.g., by aggregating the bandwidth of two paths for a large transfer. But also the opposite is true — choosing a "wrong" set of transport options can considerably hurt applications' performance, e.g., by choosing a high-bandwidth, high-latency link for a latency sensitive transfer.

In this chapter, we explore the overall design space for transport option selection, assuming that we do not want to change the Internet's protocols, but enhance the operating system (OS) on the end host. Therefore, we introduce a terminology to systemize *transport diversity* and analyze a set of protocols used in the Internet leveraging this terminology.

We start by revisiting some background about the Internet protocol stack (Section 2.1). Despite their layered design, we show how intertwined and interdependent protocol stacks are and how the same functionality, such as reliable transmission or confidentiality protection, is provided on different layers and by different protocols. Next, we recapitulate the *End-to-End Argument* (Section 2.2). We explain why, on the one hand, layering enables protocol stack composition, but, on the other hand, why focusing on layering only is not sufficient to build good transport configurations.

In the second part of the chapter, we analyze all three dimensions of *transport diversity*: multiple paths, multiple endpoints and different network protocols stacks. We consider each individual path, endpoint and transport configuration as individual transport option. To analyze transport options, we first introduce the concept of *Communication Units* Section 2.3 and compare the communication units to the Protocol Data Units (PDUs) of protocols used in the Internet and their layering in Section 2.8. Then, we explore each dimension of *transport diversity*:

- In Section 2.5, we characterize the different aspects of *path selection* and analyze protocols that incorporate path selection Section 2.6.
- The aspects of *endpoint selection* are laid out in Section 2.7.
- To approach *protocol stack composition*, we first define the problem space in Section 2.8. We identify the Transport Mechanisms, i.e., the functionality these protocol stack compositions can provide, in Section 2.9 and show how this decomposition applies to protocols used in the Internet in Section 2.10.

For each dimension of transport diversity, we look at *what transport options are available* and *how to select the most suitable transport options* with the overall goal of combining a set of suitable transport options into a transport configuration. Finally, we take a look at the tradeoffs transport option selection in Section 2.11, and conclude in Section 2.12.

2.1 The Internet Protocol Stack

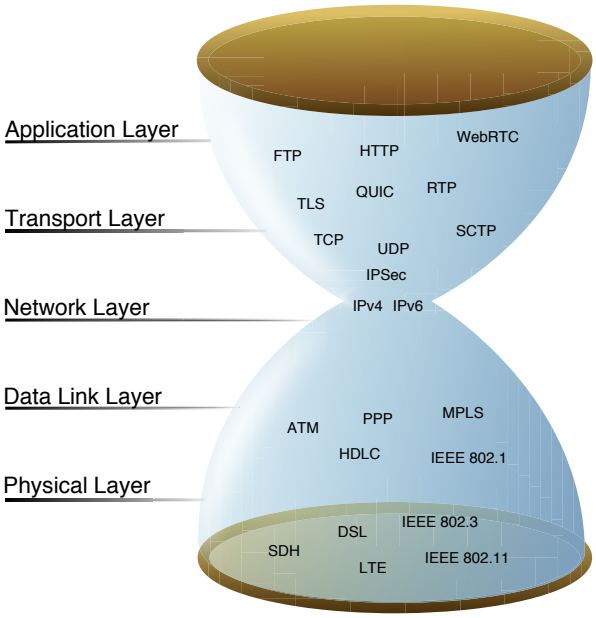


Figure 2.1: The narrow waist of the Internet.

The "Internet Protocol Stack" is a large family of protocols. Many protocols are stacked on top of each other: Each layer provides a well-defined *transport service* to the layer directly above and uses the transport service of the layer(s) below. The layering model of the Internet protocols consists of five layers and is often described by the "Hourglass Model", see Figure 2.1.

The top of the stack is consists of many *application layer* protocols, as there are diverse applications needs. For these protocols, a smaller number of *transport layer* protocols provide basic transport services, ranging from unreliable datagram delivery (UDP) over reliable stream transfer (TCP) to reliable in-order delivery of multiple message stream (SCTP).

In the middle of the stack, the *network layer* is providing network-wide unreliable packet delivery. This layer is also called "narrow waist of the Internet", as it only consists of the IP protocol in versions IPv4 and IPv6. These relatively simple protocols are the "common ground" of the Internet and allow forwarding of packets.

The bottom of the stack is made up by the *data link layer* and the *physical layer*. Protocols at the data link layer, e.g., Ethernet, enable a host or router to reach the next host or router and, thus, provide the transport service needed by the IP protocols. Finally, the physical layer is concerned with the physical medium itself and enables communication on the data link layer. The protocols at the data link layer and the physical layer are more diverse and specialized to the underlying infrastructure and properties of the physical medium.

Given a specific communication need of an application, e.g., fetching a Debian package file from a set of mirror servers, the best protocol stack to be used is not necessarily determined a priori: One could use HTTP or FTP. For HTTP, there is a choice between TCP, MPTCP, TLS/TCP, TLS/MPTCP or QUIC [17–19]/UDP as transport protocol. For FTP, depending on whether using active or passive mode, there is either choice between TCP and MPTCP, or between TCP, MPTCP, TLS/TCP, and TLS/MPTCP. As a network protocol, IPv4 or IPv6 can be used, optionally with IPSec. We ignore the data link layer and the physical layer and consider them as a property of the path. Even for this simple example, we end up with 44 feasible combinations of protocols that provide the functionality our example application needs. We call these feasible combinations of protocols *protocol stack compositions*.

Note that protocols used on the Internet do not strictly match the layers of the Internet model — therefore, their positioning in Figure 2.1 is a little fuzzy:

- Multiple protocols can reside within the same layer of the Internet model, e.g., QUIC and UDP.
- Some protocols span multiple layers of the Internet model or sit somewhere between the layers anticipated by the Internet model, e.g., TLS.
- The same functionality is implemented by many protocols at different layers, e.g., confidentiality and integrity protection can be provided by TLS as well as by IPSec.
- Functionality needed to manage or support one protocol is realized using transport service by the same layer, e.g., management for IPv6 is done by ICMPv6, which is layered on top of IPv6.
- Protocols may violate the expected layering by using identifiers of other protocols in weird ways, e.g., DHCP for IPv4 using invalid IP-addresses.

In the next section (2.2), we revisit the "End-to-End Argument in System Design" and check how it can be used as a guide to choose among protocols.

2.2 Revisiting the End-to-End Argument

One of this most famous publications about placing functionality in a communication system is the paper "End-to-end Arguments in System Design" by Saltzer, Reed and Clark [20]. Its core argument is to place functionality as close as possible to the endpoints.

> The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the endpoints of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.) [20, page 278]

This argument has proven useful as a rule of thumb and still serves as a guiding principle for many people in the IETF [21–23]. Nevertheless, it is made for designing a clean slate system and does not consider how to integrate an application in an existing layered system, e.g., the Internet. Despite this different perspective, it discusses the principle of layering in its conclusion:

> It is fashionable these days to talk about layered communication protocols, but without clearly defined criteria for assigning functions to layers. Such layerings are desirable to enhance modularity. End-to-end arguments may be viewed as part of a set of rational principles for organizing such layered systems. We hope that our discussion will help to add substance to arguments about the "proper" layering. [Conclusion of 20, page 287]

In today's Internet, the same functionality is provided at multiple layers. This is a contradiction to reference models like the ISO/OSI Reference Model [24], that tried to provide a "proper" layering with a fixed mapping between functionality and layers. Therefore, today, most people in the networking community agree that there is no single "proper" layering that fits all communication needs in the Internet. The Internet protocol stack only anchors the functionality of the network layer, which all have to agree on to enable end-to-end connectivity. On the other layers, the Internet's layering provides modularity which allows to place functionality where it serves any specific communication need.

Nearly 20 years later "Tussle in cyberspace: defining tomorrow's Internet" [25] describes this as a more generalized principle for all "tussels", i.e., cases where conflicting objectives allow no "proper" solution[1]:

> Design for tussle — for variation in outcome — so that the outcome can be different in different places, and the tussle takes place within the design, not by distorting or violating it. Do not design so as to dictate the outcome. Rigid deigns will be broken; designs that permit variation will flex under pressure and survive.
> [...]
> Modularize the design along tussle boundaries, so that one tussle does not spill over and distort unrelated issues. [25, page 466]

In case of the Internet protocol stack, this modularization takes place along two different kinds of boundaries: At **the applications' communication abstraction** and, at **network layer boundaries**.

[1] RFC3724 [23] contains similar arguments.

When using the BSD Socket Interface, applications can choose a communication unit abstraction that suits their communication pattern, e.g., messages, byte streams, or message streams. The BSD Socket Interface then translates applications' communication units into the Protocol Data Units (PDUs) of a transport protocol. This transport protocol itself adapts its PDUs to the PDUs of the layer below. This can result in dissension between semantic communication units, e.g., requests or messages of an application, and the abstraction provided by the transport service, e.g., a byte stream provided by TCP. In the next section, we discuss how to deal with this dissension when considering transport configurations.

2.3 Communication Units

When considering transport configurations, just comparing the paths, endpoints, and protocols at each layer is not sufficient. Protocols can operate on different granularities of communication, i.e., the semantic units that can be distinguished by the protocol implementation differ. To make things worse, these communication units often do not match the PDUs used by the protocol, e.g., TCP segments do not necessarily align with messages at the application layer.

2.3.1 Problem Statement

The main question of this Section is how to systematically approach the optimization problem of choosing paths, endpoints, and protocols at each layer and combining them.

Let us consider the following example: If we want to aggregate the bandwidth of two access networks to load a web page, we might need to choose between two strategies: Strategy one issues the HTTP requests over different TCP connections using different access networks. The other strategy uses a single MPTCP connection and lets MPTCP distribute the traffic. Just comparing the protocols at each layer is not useful, as the same functionality — bandwidth aggregation — is provided at different layers. Also, the distribution scheme of strategy one could be layered on top of MPTCP. Thus, care has to be taken to avoid conflicting optimizations when mixing both traffic distribution mechanisms.

2.3.2 Communication Units: A Semantic Perspective

To build, rank, and choose among transport configurations, we need to look at the functionality the individual transport options provide. To achieve the desired outcome, e.g., aggregating bandwidth or performing reliable transmission using an unreliable transport service, each of the protocols at each layer can apply suitable mechanisms to implement the functionality desired. The same mechanisms can be applied at multiple layers which apply them to different communication units. For

example, reliable transmission can be achieved by retransmission of lost packets. This can be done at the application layer for full control, which comes at the cost of complexity in the application logic. Retransmissions can be done at the transport layer for application programmer's convenience, but, if combined with in-order delivery, this comes at the cost of causing head-of-line blocking while waiting for a retransmission to arrive. Finally, retransmissions can be done at the physical or data link layer. As this cannot guarantee end-to-end reliability, it is no replacement to retransmissions at the application or transport layer. Nevertheless, applying the mechanism at the physical or data link layer can be useful to cut retransmission delays or compensate for a high loss rate of a physical media the upper layers cannot tolerate.

To approach this optimization problem, we need to analyze the mechanism providing the functionality offered by the transport option and the granularity of communication units the mechanism operates on. To do so, we not stick to the perspective of the PDUs used by the protocols, as it is often not well aligned with the messages layers on top. Instead, we choose the perspective of *Communication Units*.

Definition 2.1 (Communication Unit)
A Communication Unit is the smallest object that can be distinguished by a protocol and has a semantic meaning for the application.

That means a message split across serval PDUs of a lower layer protocols is still considered one communication according to this perspective, as the individual lower level PDUs have no meaning on their own. So, when going down the protocol stack, communication units of the application and upper layers may get split in finer chunks by lower-level protocols (see also Section 2.9.8), but become indistinguishable and. Therefore, the granularity of communication units observable at the lower layers become corse. As a result of this, we can exhibit less control about what happens, e.g., in order to optimize for a specific kind of messages.

As an example, Figure 2.2 shows two logical message streams sent by an application. The messages are transported using PDUs of a transport protocol, e.g., SCTP in this example. While logically separated from the applications' perspective, messages of different message streams can be packaged into the same PDU of the underling

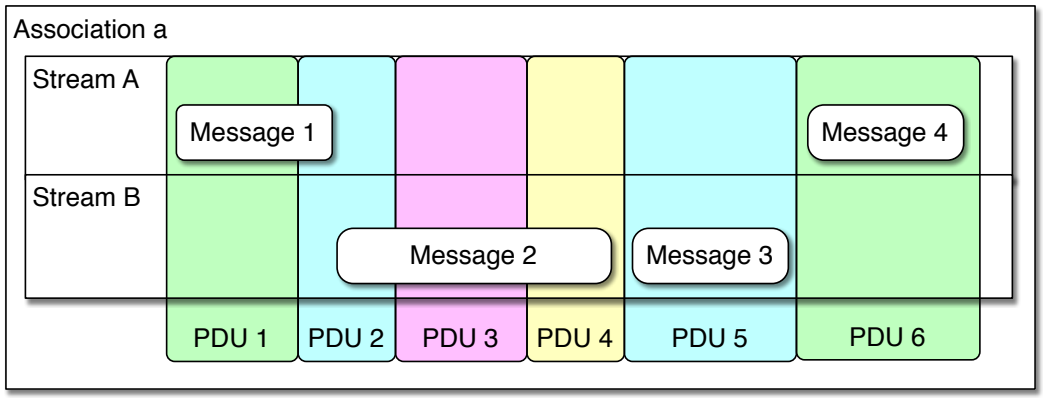


Figure 2.2: Communication Units vs PDUs.

transport. Also, messages may be split across several PDUs. If the messages get packetized arbitrarily, e.g., PDUs 1-4 in Figure 2.2, the layers providing transport services to the transport protocol cannot differentiate the individual messages or streams, e.g., to prioritize them. They can, at most, tell different associations (see next section) apart. In contrast, when PDUs and messages/streams are aligned, e.g., PDUs 5 and 6 in Figure 2.2, the PDUs can be tagged and treated differently by the lower layers.

By approaching transport configurations from a communication unit perspective, i.e., by following sets of PDUs that have a meaning for the application, we can gain the following advantages over just looking at layering of PDUs:

- We can abstract from the protocols used in the transport configuration and look at the functionality provided.
- We can reason about what communication unit a transport option can treat differently and which, by design, it has to treat the same. This allows us to reason about tradeoffs, like the ones discussed in Section 2.2, and helps us to make informed choices a given communication.
- We can distinguish between different mechanisms providing the same functionality, e.g., retransmissions and forward error connection providing reliable transmission and vet whether they can be optimized for the specific application.
- In case the same mechanism is applied multiple times within the same transport configuration, we can identify candidates for conflicting optimizations.

2.3.3 Communication Unit Granularities

To classify the different communication units in the thesis, we define the following granularities: **message**, **stream**, **association**, or **association set**. These granularities of communication units pretty much match the usual abstractions used at the socket layer or group multiple instances of these abstractions. Therefore, we use this perspective for most reasoning throughout this thesis.

Decisions should always be made on the finest communication unit granularity feasible, that usually means at message granularity and at the client. Afterwards, in case identical transport services are requested, these communication units can be aggregated into a corse granularity. For example, a web-based e-mail client that renders a message may be delay-sensitive for UI/backend communication, but requires a high bandwidth for downloading attachments like photos. Using HTTP pipelining and forcing them into the same transport configuration will most likely hurt user experience.

When a communication unit is passed down the protocol stack towards the physical layer, it is usually treated as an opaque value and, thus, does not have a semantic meaning. Therefore, at a lower layer protocol, multiple of these messages become indistinguishable and, based on this, form an equivalence class. This equivalence class becomes the communication unit from perspective of this protocol.

Note that, in practice, communication units often pass through protocols without changing granularity, e.g., a *stream* passing through TLS and TCP is still a *stream*. Communication units can also fall in multiple categories. For example, a trivial *stream* might just carry one *message*; an *association* and *association set* can carry a single stream. When a transport service involves middle-boxes, we assume these either to be transparent or to be endpoints themselves and, thus, function as proxy for some kind of communication unit — For a discussion how ho reason about middle-boxes, see Section 2.5.5.

In the reminder of this section, we will define the individual granularities used throughout this thesis and provide real-world examples for these: For some protocol examples, we use the communication units the protocol provides transfer services for, for other examples, we use the communication units of the transport service used by the protocol.

Message

Definition 2.2 (Message)
A message *is a structured piece of data that, on its own, has a meaning for the application.*

This is the smallest kind of communication unit we consider. This does not mean that this is the smallest datagram used by any protocol, as protocols may apply the *chunking* mechanism (see Section 2.9.8), but the smallest communication unit that has a meaning for the endpoint. Examples of communication units at message granularity used by well-known protocols include:

HTTP:	HTTP-Requests and Responses
HTTP/2:	all frames, e.g. DATA, HEADERS, or GO_AWAY frames [26]
XMPP:	XML messages
SCTP:	a message sent over an SCTP stream

Stream

Definition 2.3 (Stream)
A Stream is an ordered sequence of bytes or messages.

Usually, messages or bytes belonging to the same stream are indistinguishable by the stream transport and therefore are treated the same by the transport system. Examples of communication units at stream granularity used by well-known protocols include:

HTTP/2, XMPP:	underlaying TCP connection used
QUIC:	QUIC stream
TCP:	TCP connection

Association (Flow)

Definition 2.4 (Association)
An association is set of messages or streams with common endpoints.

In most cases, an association multiplexes streams or messages. As a consequence, the individual streams or messages within the association become indistinguishable for protocols in the stack below the protocol doing the multiplexing. Association and flow describe the same concept, the former from the perspective of the application, the latter from the perspective of the network. We prefer to use the term association as the term flow is overused and specified contradictory in many contexts. Examples of communication units at association granularity used by well-known protocols include:

HTTP/2: underlaying TCP connection used
SCTP, QUIC: the SCTP or QUIC connection between two endpoints
TCP, UDP: the set of IP packets that carry TCP or UDP segments and share the same 5-tuple of src-address, dst-address, protocol, src-port, dest-port

Association Set

Definition 2.5 (Association Set)
An association set is a set of semantically related associations or flows.

That means that the individual associations are distinguishable by the underlying transport, but, as the application needs to process them together, special care needs to be taken when treated differently. Examples of communication units at association set granularity used by well-known protocols include:

RTP: session consisting of multiple RTP associations containing payloads and one used for RTCP association for control messages
SIP: SIP session with all related RTP sessions

2.4 Analysis: Communication Units and PDUs

To begin our analysis of selected Internet protocols, we compare the PDU and communication unit perspective as outlined in Section 2.3. This comparison of visibility and control guides us then exploring the functionality of the protocols in Section 2.10 and path selection in Section 2.6.

Table 2.1 shows a selection of protocols and systems used in the Internet. The table is roughly sorted by the layers of the Internet model as presented in Section 2.1. We ignore all protocols below the network layer as we consider their functionality as path property. Protocols listed towards the upper part of the table are often stacked atop of protocols further below, but not all protocols listed can be stacked on-top of each

other. Table 2.1 list the communication units and PDUs the respective protocols use at their interfaces to the layers they provide transport service for (upper) and to the layers they use transport service from (lower). The corresponding comparison of the functionality provided by the protocols is shown in Section 2.10, Table 2.3.

For the remainder of this section, we revisit how these protocols compose, whether they can maintain communication units that allow to use application-aware mechanisms at lower layers and highlight some trade-offs originating from granularity issues.

2.4.1 Application Layer

Most application layer protocols such as **HTTP** and **XMPP** are designed to be layered over TCP, the only widely available protocol that supports reliable transport. Therefore, these protocols use a byte stream as PDU towards the lower layers, as expected by TCP. Despite that HTTP requests and XMPP messages have clear message boundaries, these messages are indistinguishable for TCP and all layers below and, thus, inaccessible for optimizations at the layers below. This is the result of designing these protocols into an existing ecosystem instead of designing them as complete end-to-end systems described in Section 2.2.

Table 2.1: Internet Protocols' Granularity and Interfaces

Layer	**Protocol**	**Granularity**		**PDUs**	
		Upper	**Lower**	**Upper**	**Lower**
Application	HTTP	message	stream	messages	bytes
	XMPP	message	stream	messages	bytes
	SIP	message	message	messages	varying[sip]
	DTLS	message	message	messages	messages
	TLS	stream	stream	bytes	bytes
Transport	RTP/SRTP	varying[prf]	message	messages[◎]	messages
	QUIC	stream	assoc.	bytes[⊛]	PDUs
	UDP	message	message	messages	IPT PDUs
	DCCP	message	message	messages	IPT PDUs
	TCP	stream	assoc.	bytes	IPT PDUs
	MPTCP	stream	assoc.	bytes	IPT PDUs
	SCTP	message	assoc.	messages[⊛]	IPT PDUs
Network	IPsec[t]	assoc.	assoc. set	IPT PDUs	IPT PDUs
	IP	assoc.	assoc. set	IPT PDUs	IP PDUs
	NEMO/IFOM	assoc.	assoc. set	IP PDUs	IP PDUs

[⊛] Multiple parallel streams are supported.
[◎] Messages are extracted from content by content-specific profiles.
[IPT] IP Transport PDUs — Protocols assume being layered on top of IP.
[IP] IP Packet — Regular IP packets
[sip] SIP transport can adapt to stream or message.
[prf] Determined by content specific profiles - usually message or stream of messages.
[t] IPsec used in Transport Mode.

The signaling protocol **SIP** is internally strictly message based. SIP comes with its own small transport layer that adapts SIP to the transport services of UDP, TCP, and SCTP. It maintains the message granularity as long it is run over message based transports.

Security add-ons such as **TLS** and **DTLS** are designed to slide in between application and transport. They use the PDUs of the transport's upper layer interface also towards the application layer and, thus, can be used as add-ons for new and existing applications without requiring to changes to the communication units or pattern.

2.4.2 Transport layer

Next, consider the classic and most widely used transport protocols: TCP and UDP. **UDP** is a message based multiplexing protocol that allows wrapping application messages. This usually results in sending one IP PDU per application message. Thus, it maintains the messages granularity, but forces the application to make sure its message size does not exceed the MTU of the path or rely on IP fragmentation. The Datagram Congestion Control Protocol (**DCCP**) is a drop-in replacement for UDP that adds congestion control.

TCP provides a byte stream abstraction that resembles a bi-directional Unixpipe. To maintain the properties of a pipe, it provides a rich set of transport services, including reliable transmission, in-order delivery and congestion control. **MPTCP** extends TCP with multi-path capabilities.

Note that the bundling reliable stream and unreliable message is rather a historic artifact than a conceptional one. The popularity of TCP and UDP and the lack of easily available alternatives forced protocols to adapt to either of these two options. As a consequence, protocols that need transport services like reliable transport message transport (e.g., for HTTP requests) either use TCP or implement all required transport services on top of UDP. This comes with some trade-offs. For example, multiplexing independent messages into a reliable stream can cause unnecessary head-of-line blocking: The transport has no way to extract the messages and, in the presence of packet loss, can only deliver a prefix of the stream data without violating the in-order property. Thus, it is forcing the application to wait for completely unrelated messages. See Section 2.10.2 for more discussion.

SCTP is one attempt to provide a much more flexible abstraction: It transports multiple streams of messages within a single association with all transport services TCP provides. The different streams prevent head-of-line blocking of independent messages while providing in-order delivery for dependent messages. SCTP maintains the messages for the upper layer, but multiplexes and chunks these messages within a single association and, therefore, does not maintain the message granularity to the lower layers. It is very versatile, but cannot be used as a drop-in replacement for other transport services, since it uses a different abstraction and has a relatively complex programming interface. Moreover, it is blocked by many firewalls and only available on a few OSes. This limits its deployment.

QUIC is the newest addition to the transport protocol family. QUIC was originally designed by Google as an application-aware drop-in replacement for TCP that prevents head-of-line blocking in HTTP. At the moment, it is in heavy flux and in the process of becoming a generic, state-of-the-art transport protocol. QUIC provides multiple independent byte streams towards the upper layers, which makes it a feasible drop-in replacement for TCP while avoiding head-of-line blocking between different streams. As the creation of streams within an existing association is cheap, the messages for the upper layer can be maintained by sending each message over a new stream. The messages granularity is intentionally hidden from the lower layers by encrypting the whole protocol.

A very special case with regards to communication units takes **RTP** as it tries to maintain communication units of the upper layers. RTP uses application specific *profiles* which take the applications' byte streams and chunk them into semantic messages. These messages can then be policed in application-aware ways, e.g., to implement congestion control, and multiplexed in a way that respects timing constrains. The protocol messages are assumed to be transported via UDP, but no UDP specific data is included into the RTP PDUs.

Some words about the lower layer PDU interface of the transport protocols discussed so far: TCP, MPTCP, UDP, DCCP and SCTP include an *IP pseudo header* in their checksum calculations and, thus, are required to be run over IP or an IP-like adaption layer. QUIC does not take parts of the lower layers into account and can be run on any message based transport.

2.4.3 Network Layer

With regards to network layer protocols, we only consider IP, IPSec in transport mode and, the Mobile-IP variants NEMO and IFOM. We chose the later three because these provide interesting examples with regards to their communication units and functionality. We ignore other VPNs as they, for our purpose, can be considered additional paths.

If strictly looking at the layering, **IP**[2] can only distinguish hosts and upper layer protocols. Therefore, application awareness at this granularity is pretty limited. As the OS on the endpoint still has state to match the IP PDUs to the transport protocols and their addressing/multiplexing, we still assume a per-association granularity for any functionality at the network layer despite that strict layering would only give us corse per-association-set granularity.

A good example how this per-association granularity is used is the transport mode of IPsec as well as Flow Bindings in Mobile IPv6 and Network Mobility [27] (NEMO). **Internet Protocol Security [28] (IPSec)** slides in between IP and other transport protocols to provide integrity and confidentiality protection. The decision whether to use IPsec is often done on a per-association level by OS level policies.

[2]IPv4 and IPv6 are identical with regards to their communication unit granularity.

Flow Bindings in Mobile IPv6 and Network Mobility [27] (NEMO) and **IP flow mobility for Proxy Mobile IPv6 [29] (IFOM)** are extensions of Mobile-IP. They can be used to redirect associations to certain paths, which require IP address rewriting, the addition of routing headers, and application of IPsec. NEMO assumes this is taking place within the endpoint's OS, while IP flow mobility for Proxy Mobile IPv6 [29] (IFOM) assumes this is done within an on-path element, usually a virtual network interface. While communication unit granularity of the former is analogous to IPSec, the later has to analyze the headers of upper layer protocols.

2.5 Path Selection

The availability of multiple paths is the first dimension of transport diversity we look at. To take advantage of this dimension of transport diversity, we have to identify the characteristics of the available paths. Then, for each communication unit or chunk given, we choose the best path or the most suitable set of paths. While path selection chooses a path, and, therefore, usually also an access network, it does not change IP routing or requires non-local routing.

In the following sections, we will take a closer look at different aspects of path selection. First, in Section 2.5.1, we look into the relation between path selection and scheduling. Then we discuss basic path characteristics (Section 2.5.2) and the abstraction of Provisioning Domain [30]s (PvDs) (Section 2.5.3). Finally, we discuss a few special cases including how to model middle-box behavior (Section 2.5.4 and Section 2.5.5) and path selection mechanisms integrated within cellular networks (Section 2.5.6).

2.5.1 Path Selection vs. Scheduling

When doing path selection on small communication units or chunks of communication units, like TCP segments, the term "path selection" is most often replaced by the term "scheduling". This shift in perspective is necessary, as the overhead of doing complex path selection becomes prohibitive for small communication units. Therefore, for small communication units or chunks, path selection is usually split into two subproblems:

Definition 2.6 (Candidate Path Selection)
Candidate Path Selection determines feasible paths and chooses a set of preferred paths that can be used for an larger set of communication units.

Definition 2.7 (Path Scheduling)
Path Scheduling selects one or more paths from the Path Candidates for each chunk or small communication unit.

For example, in case of MPTCP, candidate path selection decides which subflows to establish while path scheduling assigns bytes from the send buffer to the subflows.

Thus, while the candidate path selection can afford a more expensive decision process scheduling has to be cheap, e.g., only based on local state. Examples of scheduling strategies include:

- Schedule all chunks on a single preferred path, as long as this path is available, otherwise, use a less-suitable backup path.
- Distribute chunks based on path capacity, whereby capacity can be pre-determined or information available from other mechanisms, e.g., derived from the congestion window of MPTCP.

A similar effect as using the two-step approach described above can be achieved by caching path selection results. Indeed, using a cached path selection results can also be considered a scheduling strategy.

2.5.2 Path Characteristics

The characteristics of paths are manyfold. Some of them are end-to-end properties, while others are shared by all paths using a local interface or a provisioning domain (see Section 2.5.3). The most obvious characteristics of a path are its **bandwith**, **delay** and **packet loss probability**. All three characteristics are, in principle, end-to-end properties. Determining them for multiple candidates before the actual communication is often infeasible because of the overhead involved. In most access networks, where these characteristics matter, we can assume that the bandwidth bottleneck and the governing delay contributor is the access networks itself [31], and therefore with a small loss of precision treat these characteristics as properties of the local interface representing the path.

Also, when looking at cellular networks, **monetary cost** becomes a relevant property. Without destination based billing or zero-rating[3], cost is a property of the Provisioning Domain [30] (PvD) — usually billed as traffic budget, traffic volume or in the time domain.

Depending on the applications' needs, a path selection mechanism can be used for different objectives, including **minimizing delay**, **maximizing or aggregating bandwidth**, **maximizing availability**, and **minimizing cost**. Also, to achieve these objectives, there are multiple ways how to use path selection, e.g., based on the application's communication units, on streams of them or, based on chunks within a protocol.

2.5.3 Provisioning Domains

Different paths can belong to the same PvD. For example, a Laptop computer can be connected via WiFi and using an Ethernet cable to the same access network. While the two interfaces to this network are different paths from the OSes perspective, these paths share most properties and, thus, can be treated as being the same network for many purposes, e.g., name resolution (see Section 2.7.1).

[3] Zero-rating is the practice to exclude certain endpoints from traffic budgets and has become a major regulatory concern in cellular networks.

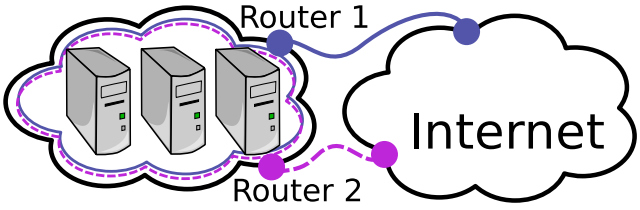


Figure 2.3: Multiple L3 Access Networks on a Single L2 Link.

Also, the opposite case is possible. Figure 2.3 shows an example where a home network is using two access networks each with its own router (Router 1/2)[4]. In this scenario, each router announces at least one IPv6 prefix for its access network. This enables all applications within the home network the choice of access network by choosing the appropriate IP address as source address. With "just" the appropriate routing setup, a single interface can provide multiple paths that are in different provisioning domains.

A comprehensive solution for this problem is provided by the *Multiple Provisioning Domain Architecture* [30] of the concluded IETF MIF working group. To detect which paths are provided by a common provisioning domain, all paths are labeled with their provisioning domain. These labels are announced with the address auto-configuration by protocols like IPv6 Stateless Address Autoconfiguration [32] (SLAAC) or DHCP as discussed in [33].

2.5.4 On-Path Network Functions

Some paths traverse middle-boxes or proxies that provide network functions on certain protocols. Examples for such network functions include [34]:

Firewalls limit the possible associations or their parameters.

WAN optimizers change transport protocol options or terminate transport protocol sessions and connect them end-to-end to improve performance on links with, from endpoint perspective, unusual properties, e.g., unexpected high latency.

Caching-Proxies cache application layer objects and thus allow to save roundtrip times and backbone bandwidth.

Multipath-Proxies split the chunks of a flow to perform transparent bandwidth aggregation over multiple paths.

Transcoders change the format or stream rate of media streams.

While network functions can benefit applications' performance, they also can prevent certain protocols variants [35, 36]. Therefore, the existence of network functions can influence the path choice. If a provisioning domain provides optional network functions, like HTTP or SOCKS proxies, they can provide derivate-paths for some protocols.

[4] See "Multi-Homed on a Single Link: Using Multiple IPv6 Access Networks" [3] for a comprehensive discussion of this scenario.

2.5.5 Path Selection through Network Function

Middle-boxes can perform path selection on communication units passing through them. These network functions are usually found as part of **hybrid-access** routers, **WAN bonding** devices or as virtual interfaces in **cellular offloading** solutions. The term cellular offloading [37, 38] refers to moving some traffic from a cellular network to other access networks, e.g., WiFi hotspots. The use-case of hybrid-access is exactly the opposite — it refers to bundling cellular and residential broadband networks [39], to boost the performance of old residential broadband access networks, e.g., old DSL infrastructure, by offloading some of the traffic to a high bandwidth cellular network. WAN-bonding devices bundle multiple commodity residential broadcast access lines to increase bandwidth or availability.

While all these use-cases are more-or-less identical in their functionality at transport- or network layer, their actual implementations largely differ. WAN bonding devices and hybrid-access routers are usually deployed as part of a customer premise equipment (CPE) or as middle-box directly connected to multiple CPEs. Offloading solutions are often realized as virtual interfaces, e.g., as logical interface [40] for IFOM with *Proxy Mobile IPv6* [41], or as configuration agent for the IP stack.

Integrating these middle-boxes into automatic transport option selection is a complex task. If the network function is explicitly requested by the client, e.g., in case of TCP converters [42], paths through the network functions can be modeled by assuming multiple paths matching the network function requested. In case of *IP flow mobility* [29] without virtual interface, the configuration provided by a 3GPP access network discovery and selection function (ANDSF) can be used as part of the local policy. If the strategy used by these devices or the communication unit granularity they operate on is not known, the objectives of the path selection performed by the middle-box may conflict with the objectives at the end host, e.g., a middle-box may optimize for bandwidth in a case that is latency sensitive. In some cases, using the functionality provided by the network function can only be avoided by choosing a different protocol stack composition or path.

2.5.6 Path Selection and Cellular Networks

The lower layers of 3G, 4G, and other wireless networks include support for path selection or multiple paths of different granularity and complexity:

- GSM and other 2G networks support endpoint mobility.
- 3G networks use soft handovers, where during the transition to a new base-station, a handset remains connected to the old base-station. A soft handover mixes path selection and forward error correction in a clever way that can provide more bandwidth and less packet loss than each of the cells involved could provide under the given circumstances.

- For LTE, there exists a transparent multi-path extension called cooperative multi-point (CoMP). With CoMP, multiple paths from an endpoint using different eNBs to the network core are used to distribute traffic, provide more bandwidth, and improve network coverage.

As these technologies are designed to be transparent to the endpoint, the endpoint cannot choose whether to use any of these when using the path. In heterogeneous access scenarios, solutions below the network layer cannot support transitions between different access technologies and are not exchangeable with other protocols that, in theory, provide the same mechanisms. Therefore, we do not consider such functionality as mechanisms, but as a property of the path.

Nevertheless, cellular networks also support multiple *Access Point Names (APNs)*. Each APN provides a virtual network attachment including its own addressing, reachability, traffic policing and on-path network functions. Therefore, from a transport option selection point of view, these APNs provide different paths in different provisioning domains that most probably share a common access media. The ANDSF of the cellular networks provides hints on which of these paths to use for certain destinations.

2.6 Analysis: Path Selection Opportunities

Now, after introducing the different aspects of path selection, we analyze Internet protocols that support path selection. Hereby, we focus on the *granularity* on which path selection is done

Definition 2.8 (Path Selection Granularity)
We define three types of path selection granularities:

Type 1 All communication units *are assigned to* the same path.
Type 2 Each communication units *can be assigned to* one path.
Type 3 Communication units can be split *and assigned to* multiple paths.

Type 1 path selection can be used to implement mobility and to increase availability, Type 2 and 3 allow bandwidth aggregation by using multiple paths, but differ in their distribution granularity.

In addition, we explore whether a protocol enables path selection on the endpoint or whether it can also be used as part of a network function, e.g., a hybrid access middle-box, as discussed in Section 2.5.5.

Table 2.2 shows an selection of protocols that support path selection. In the remainder of this section, we go up the stack, from network layer to application layer and take a closer look at the protocols and the advantages and drawbacks of using them for path-selection.

Table 2.2: Internet Protocols Performing Path Selection

Layer	Protocol	Path Selection Granularity Type	Comm. Units	Location end-point	on path
Application	HTTP	Type 2	message	✓	✓[prx]
	SIP	Type 2	stream	✓	✓[prx]
Transport	MP-RTP	Type 3	message[prf]	✓	
	MPTCP	Type 3	stream	✓	✓
	SCTP	Type 2	message	✓	
Network	IP (routing)	— no path selection —			
	IP (policy r.)	Type 1	assoc. set		
	IP (ECMP)	Type 2/3	assoc. set	✓	✓
	Mobile IP	Type 1	assoc. set		
	NEMO	Type 2	assoc. set	✓	
	IFOM	Type 2	assoc. set	✓	✓

[prx] Can be done within a proxy.
[prf] Depends on content specific profiles.

2.6.1 Network Layer

Realizing path selection at the network layer enables an end host to use multiple paths without requiring a multi-path aware communication partner. We do not consider the regular routing in the Internet as path selection, but as a prerequisite to provide the paths we can choose from.

In contrast to regular routing, we identify policy-routing as Type 1 path selection, as it directs certain association sets to a specific path. Based on this differentiation, we also have to consider some Equal Cost Multi-Path Routing [43] (ECMP) routing strategies as performing path selection: If the ECMP path selection is done in a way that forwards all PDUs of an association or association set along the same path[5], e.g., based on the 3-tuple or 5-tuple, we consider this as Type 2 granularity. If the ECMP path selection just uses a round-robin scheme, we consider this as Type 3 granularity. Note that ECMP path selection is not application-aware and therefore only included as a corner-case.

A particularly interesting case is Mobile IP. Mobile IP allows keeping an IP address even when leaving the access network and tunnel the traffic to the original access network. Therefore, it provides an addition path. The Mobile IP extensions NEMO[6] and IFOM[7] allow multiple paths to the same network and can do path selection through policy routing. In case of IFOM, where the mobile IP operation is executed by a virtual interface driver or a middle box, this path selection can even happen on a middle-box. The 3GPP assumes that rules for policy routing with IFOM are provided by the ANDSF.

[5] This is usually done to prevent packet reordering which can cause major TCP performance degradation.
[6] Flow Bindings in Mobile IPv6 and Network Mobility [27]
[7] IP flow mobility for Proxy Mobile IPv6 [29]

2.6.2 Transport Layer

Path selection at the transport layer promises low overhead, but requires the collaboration of the other endpoint. MPTCP allows adding additional endpoint addresses to an existing TCP connection. To enable a path candidate, the MPTCP path manager selects an appropriate source and destination addresses, establishes a new TCP connection over this paths, and links this TCP connection to the MPTCP connection as a subflow. Each chunk of the MPTCP stream can then be assigned to one of these subflows; therefore, MPTCP provides Type 3 granularity. Retransmissions are be tried over the same subflow first, but can be performed using other subflows if needed. This per-chunk path selection can be tailored to enable diverse strategies, e.g., to do bandwidth aggregation or to allow fast fallback if one path becomes dysfunctional. As TCP is not protected against on-path modification, middle-boxes can convert regular TCP streams into MPTCP streams [42].

A proposed multipath extension for RTP [44] works in a similar fashion as MPTCP, but allows out-of-band address exchange using, e.g., using SDP in SIP. It works on the message granularity provided by the RTP profiles. Due to the complex interplay of external signaling and rate control, RTP cannot be converted to multipath RTP by middle-boxes. But similar functionality can be achieved using application layer proxies.

SCTP also allows the exchange of additional endpoint addresses. But by default, it uses only one primary path and uses additional addresses as a fallback. In theory, it is possible to assign each message or stream to one of the available paths by using an appropriate source and destination address when sending the object, which results in Type 3 granularity. To use additional paths, SCTP does not require an additional handshake, but it is advisable to check path functionality before using it. In practice, the BSD Socket API and the abstract SCTP API [45] do not expose or enable this functionality[8].

2.6.3 Application Layer

Path selection at the application layer enables path selection at message level. But this comes at the cost of complexity and often requires additional expensive communication handshakes. We choose HTTP as representative of a message based application layer protocol that uses stream transport service and SIP as representative of a signaling protocol that can be used in complex architectures like the IP Multimedia Subsystem (IMS).

[8] The abstract SCTP API defined in RFC6458 [45] only allows overriding the destination address for the packet, but not setting the source address or annotating additional attributes for path selection. The semitic of the `bind()` for SCTP sockets is slightly different form the semantic of TCP/UDP sockets to support fallback addresses, which prevents its use for path selection as described in Section 2.6.3. This limits path selection for SCTP to solely destination address and routing based fallback schemes.

The basic operation of **HTTP** uses a stateless request/replay scheme[9]. Distributing the requests/messages is fairly straightforward by issuing the requests over a transport service bound to a certain path. As we can choose a path per request, we end up with Type 2 flow granularity. The underlying transport services, TCP or QUIC in case of HTTP, can be bound to a path, e.g., by appropriately choosing source and destination addresses for the IP PDUs. Adding a new path may induce significant overhead with regards to latency due to the handshakes. Usually, HTTP+TLS handshakes take at least 2-3 round-trip time (RTT)s, a QUIC handshake takes 1-3 RTTs (0 RTT is only possible for existing connections). In addition to the end host, an HTTP proxy can do the path selection as on-path network function.

SIP is a signaling protocol to negotiate media sessions (typically phone calls and streaming sessions, e.g., as part of the IP Multimedia Subsystem (IMS) used for Voice over LTE). It supports the concurrent use of multiple contact IP addresses for the (concurrent) registration of one endpoint, allowing the creation of multiple signaling paths to a single endpoint. In addition, this can be combined with the path selection features of SCTP or MPTCP.

On these signaling paths, signaling messages carry session description protocol (SDP) messaging to negotiate media streams (i.e., calls or streaming). SDP allows for the (re-)negotiation of the streams of one media session over multiple paths. In turn, this can be used to manage application layer path selection as described for HTTP. From this point of view, SIP can offer application layer multipath support with a Type 2 granularity. Other signaling protocols as XMPP/Jingle use or resemble SDP and, therefore, can be used in the same way to support application layer path selection.

2.7 Endpoint Selection

An endpoint, i.e., an application or host, that initiates a communication, has to know which counterpart to communicate with. Starting with some representation, e.g., a hostname, and depending on the kind of communication, this information can be derived from the original representation by many different means:

- The other endpoints can be given by configuration, e.g., a list of recursive name servers for DNS resolution is usually configured automatically by DHCP or SLAAC, or are configured manually by an administrator.
- The other endpoints can be derived through name resolution, e.g., by using DNS to resolve a hostname to an IP address.
- The other endpoints can be cached within the application, e.g., used by many peer-to-peer networking applications.

In all three cases, more than one derived endpoint my be available, e.g., several addresses of a multi-homed server or a list of CDN servers hosting the same content.

[9] In HTTP/2, the header compression algorithm and *HTTP push* can introduce protocol-level inter-request dependencies.

Endpoint selection is often tied to path selection as endpoints might only be reachable over certain paths. For example, recursive DNS servers are often reachable within a single provisioning domain, e.g., the network of an access provider, to prevent traffic amplification attacks. Depending on the source the endpoint choice originates, there are different ways how endpoint selection can be done. Configuration based endpoint selection often comes with a priority or order (primary/backup server). Cached lists of endpoints can also keep track of historical data like latency of recent communications, available bandwidth or connection failures. In the next section, we will take a closer look at the case where endpoint selection is done on the results of name resolution.

2.7.1 Name Resolution

Name resolution on the Internet relies on both, local decisions at the endpoint as well as decisions within the DNS infrastructure. This results in spreading the endpoint selection process across different administrative domains, where each domain independently contributes to the final selection result.

DNS based traffic engineering is used by major CDNs to influence endpoint selection. The major objectives for this are mapping users to appropriate servers and load-balancing between severs. It involves elaborate network measurements and load-based heuristics. Therefore, authoritative DNS servers often return different responses based on the perceived origin of the request. If an endpoint uses different paths for name resolution and initiating a communication, the chosen server may be suboptimal or not reachable at all using the given path. These problems can lead to a significant performance degradation.

Thus, the DNS configuration strategy outlined in RFC 6106 [46], i.e., having a single DNS configuration repository that merges all DNS servers received, is insufficient for automated transport options. Issuing all DNS queries using a single access network/path will most likely interfere badly with DNS based traffic engineering on other paths. Therefore we need to do endpoint selection in conjunction with access network/path selection. This requires maintaining separate resolver configurations per provisioning domain (see Section 2.5.3) as specified by [30, Section 5.2.1.]. If this is infeasible, a client can take advantage of the DNS Client Subnet EDNS0 extension [47] to request individual name resolutions per path / local endpoint. In addition to that, there may be additional requirements that link name resolution and path selection. E.g., RFC6731 [48] specifies a mechanism to provide special name server configurations for certain endpoints that should prefer a certain PvD.

2.8 Protocol Stack Composition

Protocol Stack Composition is the last and most complex dimension of transport diversity we take a closer look at. It refers to the process of choosing a set of protocols for a given communication unit. In this section, we present a simple algorithm for building all feasible protocol stack composition to give an intuition for the Transport

Mechanisms for Protocol Stack Composition (see Section 2.9). It also demonstrates why it is useful to decompose Internet Protocols into Transport mechanisms (see Section 2.10) in order to derive protocol stack compositions.

Definition 2.9 (Feasible Protocol Stack Composition)
A protocol stack composition is feasible *for an application, iff its end-to-end transport service provides all functionality required by the application.*

Besides being feasible, an optimal protocol stack composition should match the preferences of the application as close as possible and minimize overhead. For sake of simplicity, we ignore all protocols below the network layer and consider all of their properties as path properties.

The following algorithm builds all feasible protocol stack compositions.

1. Build a directed graph of all available protocols (from application to network layer). In this graph, there is an edge between two protocols p_1 and p_2, iff p_1 can layer on top of and p_2.

2. Build all paths in the graph between the local endpoint, i.e., the application layer protocol, and one of the IP protocols. These paths represent all stack compositions. We can, w.l.o.g., ignore cycles as these cycles would represent tunneling configurations which we do not consider as protocol stack compositions, but as additional paths.

3. For each stack composition, determine the set of functionality it provides.

4. Eliminate stack compositions for which do not provide the required functionality. The remaining stack compositions are all feasible.

To build all possible transport configurations, one needs to build all combinations of feasible protocol stack compositions, the available paths and endpoints. If one of the protocols within a protocol stack composition is multi-path aware, the respective transport configurations have to be built with all combinations of endpoints and the power set of the available paths.

As the process above shows, the list of protocol stack compositions can become quite extensive. Therefore, to use protocol stack composition within a real system, a policy component is needed that ranks and filters the transport configurations before trying them or applying connection racing techniques like Happy Eyeballs [49]. We discuss how to build such a policy component in Chapter 4. Next, in Section 2.9, we present how to systematize protocols based on their functionality and analyze our selection of Internet Protocols in Section 2.10.

2.9 Transport Mechanisms for Protocol Stack Composition

Transport protocols on the Internet provide a large variety of functionality. While the functionality of simple protocols like UDP is easy to describe (multiplexing streams of messages), describing the functionality of complex protocols such as QUIC, MPTCP or SCTP is hard and can easily fill multiple pages of text. Also, as we have seen in Section 2.1, the same functionality can be provided at many places throughout the whole stack.

Definition 2.10 (Transport Mechanism)
A transport mechanism is a functionality a protocol offers as part of its transport service.

In the following, we explore the mechanisms transport protocols offer, to, later in Section 2.10, decompose protocols protocols used in the Internet along these mechanisms. This decomposed representation of the transport options provided by different protocols can be used to build transport configurations that provide all functionality required by the endpoints.

2.9.1 Reliability

A reliability mechanism compensates packet loss and packet corruption. The transport service guarantees the endpoint to receive all messages or streams they send, but it may deliver them in a different order than the endpoint handed them over to the transport service. There are two basic categories of mechanisms providing reliability: **Retransmission Mechanisms** and **Forward Error Correction (FEC)**.

2.9.1.1 Retransmissions

Retransmission based mechanisms minimize the overhead for reliable transmission by only retransmitting lost data, but can only do so after detecting the loss. Loss detection is often based on a combination of acknowledgments and timeouts, and therefore, messages that are lost and retransmitted are delayed by at least one RTT. Retransmission based reliability mechanisms perform best in environments with low packet loss and bit error rates and are the predominant end-to-end mechanism on the Internet. For example, TCP, SCTP, and QUIC use retransmission based reliability mechanisms.

Note that reordering sensitivity is no general property of retransmission based reliability mechanisms, but a problem specific to TCP's retransmission mechanism[10]. QUIC is reordering tolerant by using a reordering-timeout and a more flexible acknowledgment mechanism.

[10] The reordering sensitivity of TCP is only solved in part by SACK [50]

2.9.1.2 Forward Error Correction

Forward error correction based mechanisms scarify bandwidth for the ability to recover from bit errors or packet loss. They use additional bandwidth to add redundancy to the PDUs handed to the lower layers. Network coding refers to a special class of forward error correction schemes. These mechanisms can be tuned or adapted to different network conditions or be used to exploit multiple paths.

These mechanisms are useful if packet loss or bit error rates are high. They are predominantly used to counter the effects of lossy physical media, e.g., in DSL. Forward error correction based mechanisms are rarely used end-to-end on the Internet, as the usual end-to-end packet loss and bit error rates in the Internet core are low. The only deployed end-to-end protocol known to the authors that used FEC was an old version of Google's QUIC. It did not perform well compared to retransmission schemes [51].

2.9.2 Ordering

Protocols that provide an ordering mechanism guarantee some kind of ordering across messages or streams to be preserved. The exact kind of order is defined by the transport service definition. An ordering mechanism does not necessarily preserve total chronologic ordering of all messages, but it can also preserve a partial order of messages. In case of QUIC, the bytes of individual streams are delivered in order, but the order of writes across different streams is not preserved or available on the receiver side.

2.9.3 Integrity Protection

A transport service providing integrity protection guarantees that communication units arrive at the receiver end without modification. For this thesis, we ignore simple checksum based mechanisms that can be circumvented by middle-boxes or attacker and only refer to cryptographic mechanisms as integrity protection. Integrity Protection may interfere with on-path network functions (see Section 2.5.4).

2.9.4 Confidentiality Protection

Confidentiality protection refers to hiding the content of the communication units form eavesdroppers. With mass surveillance, as documented by Edwards Snowden, end-to-end confidentiality protection has become a key mechanism on the Internet [52].

Besides the obvious use of privacy protection, confidentiality protection has gained popularity within protocol to prevent *ossification* [53] — In ITEF QUIC, this use

is called “greasing”. Applied to almost all parts of the protocol, it prevents middleboxes (see Section 2.5.4) to relay on protocol details and causing incompatibilities that used to be a major obstruction for protocol evolution [54].

2.9.5 Authenticity Protection

Authenticity protection is usually needed to enable integrity protection and confidentiality protection. Both only work iff we can make sure that we are speaking to the right endpoint. In most use cases, integrity protection and confidentiality protection are realized though symmetric cryptography, with keys derived by a record protocol [55] implementing authenticity protection using asymmetric cryptography.

2.9.6 Congestion Control

Congestion control mechanisms prevent endpoints from overwhelming the network by sending more data than the available bandwidth permits. Without congestion control, many links on the Internet would be overwhelmed resulting in routers dropping packets towards the congested link. When the ratio of the dropped packets gets too high, the network becomes unusable, and we call the state congestion collapse. Packet loss caused by congestion can and must not be countered by reliability mechanisms, as retransmitting or adding redundancy increases the data rate and therefore congestion.

TCP style fairness is de-facto standard for congestion control algorithms. Its basic idea is that packet loss is most likely caused by congestion and thus reduces the sending rate in case of packet loss to a fraction of the last data rate. Then TCP tries to slowly increase the data rate linearly to determinate the feasible data rate. In effect, TCP’s average data rate will oscillate slightly below the feasible data rate. Multiple TCP shares will converge to a fair share. Other congestion control mechanisms should try to show a similar behavior and converge to a fair share of the available bandwidth.

Based on this basic principle, there exist many variants of loss based and delay-based congestion control algorithms and commodity TCP implementations usually implement more than one algorithm. Therefore, congestion control can also be subject to transport option selection.

In addition to loss based congestion control, Explicit Congestion Notification [56] (ECN) allows notifying the receiver of an IP packet that a link on the path is fully loaded and packet drops due to congestion are expected unless the data rate is decreased. The receiver is then obligated to communicate this back to the sender on a higher protocol layer, e.g., in TCP, so that the sender can lower the data rate accordingly. Used together with Active Queue Management [57] (AQM), ECN based congestion control algorithms can gain significant performance benefits [58]. Therefore, if a path provides end-to-end AQM based ECN, transport option selection should select an ECN aware congestion control mechanism.

2.9.7 Multiplexing

Multiplexing refers to merging fine-grained communication units into a stream or association of a corse grain communication unit. A protocol that provides a multiplexing mechanism uses different communication units at its upper and lower layer boundaries, e.g., in case of SCPT, it multiplexes multiple message streams from the application layer into one association at the IP layer. Our definition of multiplexing does not require joining multiple PDUs from the upper PDUs into a single PDU of the lower layer boundary, it is sufficient, that the lower layer cannot distinguish the communication units of the upper layer without violating layering. Therefore, port based addressing in UDP is also considered multiplexing.

2.9.8 Chunking

When a communication unit is handed over to another layer, it might be necessary to split an object, a stream or a set of associations into one or more parts. Typically, chunking splits only large objects or streams into multiple ones while keeping smaller entities untouched. Associations or Flows are typically not split, but sets of Associations or Flows might be partitioned. Once split into chunks, each chunk can be transferred individually over different transfer options.

Chunking can occur at different layers within a system:

- A Web site consists of multiple objects or files. Therefore, the files can be seen as the natural chunks of a Web site.
- RTP profiles organize how to transfer, e.g., video, over RTP, and split a data stream at semantic boundaries.
- TCP takes as input a byte stream and chunks it into segments. TCP chunking (segmentation) occurs at arbitrary byte ranges; thus, it will most likely not align with boundaries of objects that are multiplexed within an application layer Association on top of a TCP connection.
- IP fragmentation splits an IP PDU into smaller PDUs; all but the first chunk does not even have a header to dispatch them to an endpoint.

As chunks often have no meaning on their own, the communication unit granularity does not change. For example, a protocol that applies chunking to a message-stream cannot distinguish between messages within the stream. The protocol can transparently chunk messages, but needs to reassemble the stream before handling the message-stream back to the application. This introduces complex trade-offs when combined with other mechanisms, e.g., by increasing loss probability or introducing head-of-line blocking.

In practice, chunking is often constrained to maintain certain properties that are desirable for the overall system. Examples of such restrictions include:

- Segmentation in TCP restricts the chunk size, i.e., TCP segment size, to the IP MTU or IP Path MTU to avoid fragmentation at the IP layer.
- Equal cost multipath routing does not distribute packets, but flows to avoid reordering that would hurt, e.g., TCP.

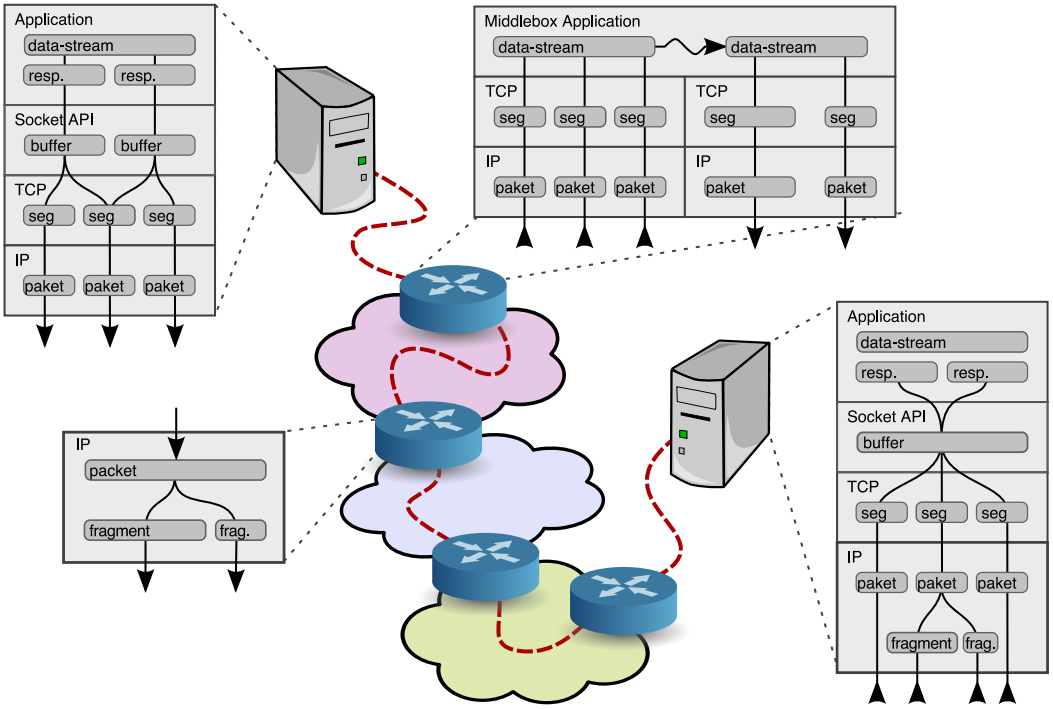


Figure 2.4: Example of different kinds of chunking in the Internet that a TCP flow may experience.

Figure 2.4 shows an example of how data from an application can be chunked while it traverses the network. Typically, the layers below the application layer have to adjust to the MTU of the layer below and split PDUs, e.g., to enable striping or forward error correction[11]. Hereby, each layer has a tradeoff between overhead per chunk, increased delays, increased chunk reordering, overhead for reliability. Moreover, memory constraints for buffer management have to be considered.

When building a transport configuration, the chunking chosen at each layer boundary is a tradeoff between complexity, knowledge, and performance. On the one hand, smarter choices at the application layer or transport layer can reduce the load on the network layer, but using too small or oddly sized chunks can hurt performance. On the other hand, choosing a slightly too big chunk size triggers chunking at the next layer and often results in doubling the numbers of packets, one at real MTU size, the other only carting a few bytes. This leads to a higher loss probability for the double-chunked packets, increases the load on routers and wastes capacity on certain media that require a minimum packet size. A good example for this trade-off is IP fragmentation, as discussed in Section 2.10.4. Given these tradeoffs, transport option selection should minimize re-chunking if possible. However, in the Internet re-chunking is frequently done, as highlighted in Figure 2.4, even though most protocols already try to incorporate the chunk size of the lower layers if feasible.

[11] Nowadays, middle boxes might also effect the chunking of packets either intentionally or as a result of their purpose, e.g., by changing the application layer data stream.

2.9.9 Path Selection

While we discuss path diversity as a dimension of transport diversity in Section 2.5, in this section, we look at the mechanism for selecting which of the available paths to use for each communication units. As stated earlier, this selection can be done in different parts of the protocol stack and, thus, for different kinds of communication units or chunks of them:

- Path Selection can be done within an application on a per message or stream granularity. Choosing one of the local network interfaces can be achieved by setting the source IP address for the communication to the IP address of the interface.
- Path Selection can be done within the transport layer on a per message, stream, or association granularity. For example, MPTCP allows splitting a stream over multiple paths. Each chunk of the stream, each TCP segment, is assigned to one or more subflows for transmission. The subflows are visible to the lower layers as streams or associations.
- Path Selection can also be made on a per-association granularity within a middle-box or network function like a MPTCP-Proxy or a BANANA-box. Path selection is usually realized by choosing the outer source and destination of a tunnel.
- In case of an ECMP router, path selection is made at the network layer on a per association or association set basis.

There is no single optimal place within a protocol stack to perform path selection. As always, the protocol should operate at a granularity, which allows distinguishing between communication units that require different objectives.

Using path selection on chunks of corse communication units can enable better distribution than using path selection on the original communication units, e.g., by splitting messages into smaller chunks, but can also cause head-of-line blocking or increase the message loss probability. This again is a trade-off that depends on the applications needs.

For communication units that can be treated the same, functionalities like bandwidth aggregation can be realized at different protocol levels. Indeed, in Chapter 5, we show using path selection to distribute downloads of multiple objects over multiple paths achieves the same performance, as using a single stream to download these objects while using path selection on the chunks of that single stream.

2.9.10 Mobility

Mobility refers to migrating a stream or association to another path. Therefore, mobility can often be considered as a use-case of path selection as most protocols that do path selection also provide mobility. Note that mobility support of physical layer and data link layer protocols, such as WiFi and 2G networks, that provide simple switch-overs between base stations, are considered features of the path and

therefore not subject to transport option selection. Still, there are also protocols like Mobile IP [59, 60] (without IFOM extension) that only allow migration of all communication units they provide transport services for.

2.10 Analysis: Transport Mechanisms

Next, we take a look at Internet protocols and systems from a functionality perspective. Therefore, we decompose these protocols into *transport mechanisms*, see Section 2.9. Hereby, we explain the tradeoffs incurred by using the respective protocols to gain the desired functionality. By looking at the PUDs, we can see whether two protocols can (in principle) stack on top of each other. By looking at the communication units, we see at which granularity mechanisms of a protocol can operate and, therefore, see whether they can be used in an application-aware manner.

Table 2.3: Internet Protocols' Transport Services

Protocol	Congestion Control	Ordering	Reliability	Integrity Protection	Confidentiality Protection	Authenticity Protection	Chunking	Multiplexing
HTTP	⊛	⊛	⊛				bytes	requests
HTTPS	⊛	⊛	⊛	⊛	⊛	⊛	bytes	requests
XMPP	⊛	⊛	⊛	(⊛)	(⊛)	(⊛)		messages
SIP			✓	(⊛)	(⊛)	(⊛)		messages
DTLS				✓	✓	✓		services,name
TLS		⊛	⊛	✓	✓	✓		services,name
RTP	✓[prf]	✓[prf]					messages[prf]	messages
SRTP	✓[prf]	✓[prf]		✓	✓	⊛[sig]	messages[prf]	messages
QUIC	✓	✓	✓	✓	✓	✓[tls]	bytes	connection-id,+[tls]
UDP								ports
DCCP	✓							ports
TCP	✓	✓	✓				bytes	ports
MPTCP	✓	✓	✓				bytes	ports
SCTP	✓	✓	✓				bytes	ports,streams
IPsec (ESP)				✓	✓	⊛[ike]		spi,next-header
IPsec (AH)				✓		⊛[ike]		spi,next-header
IP		(✓[fr])					(fragments)	address,next-header
NEMO/IFOM						⊛	assoc.	

⊛ Protocol requires transport service.
✓ Protocol provides transport service.
[prf] Realized by content specific profiles.
[tls] Uses TLSv1.3 as sub-protocol; imports authenticity protection and multiplexing from TLS.
[ike] Realized externally by external protocol IKE/IKEv2.
[sig] Realized externally by external signaling protocol (e.g., SIP, XMPP, WebRTC).
[fr] Only when fragmentation is used and only to re-assemble IP PUDs

Table 2.3 shows the same selection of protocols and systems as Table 2.1. The table is again roughly sorted by the layers of the Internet model as presented in Section 2.1. We ignore all protocols below the network layer as we consider their functionality as property of the path. Table 2.3 lists which of mechanisms introduced in Section 2.9 these protocols implement as part of their transport service and which transport services of lower layers they require. For the mechanisms *Chunking* and *Multiplexing*, Table 2.3 also lists for each protocol on which abstractions these mechanisms operate. This time, we have separate entries for some protocol variants, e.g., HTTP/HTTPS, as they require or provide different transport services.

For the remainder of this section, we will revisit the individual mechanisms. This time, we structure the analysis along the mechanisms, rather than along the layering. This allows us to better compare the abstraction these mechanisms work on and the side-effects arising.

2.10.1 Congestion Control

All application protocols that can consume a significant amount of bandwidth either require congestion control as a transport service or implement it. Low-bandwidth protocols such as **DNS** or **SIP** often ignore congestion control. Traditionally, originating from TCP, congestion control is provided by transport layer protocols. With the exception of UDP, all transport layer protocols listed in Table 2.3 provide congestion control in some way.

Congestion control has to work regardless of what data the application is about to send. Therefore, in presence of congestion, application awareness may only determine what to drop first. It should not exceed a, from congestion avoidance standpoint, admittable data rate. Protocols that support multiple streams, such as **SCTP** and **QUIC**, solve this by allowing to prioritize individual streams. **RTP/RSTP** uses a clever layer violation: It performs congestion control in an application data specific way. If RTCP reports packet loss, the RTP profile can selectively drop messages from the application's data streams to reduce the bandwidth and guarantee in-time delivery of the remaining data.

2.10.2 Ordering and Reliability

Due to the heritage of TCP, ordering and reliability are implemented at the transport layer in most cases. As already explained in Section 2.4.2, applying ordering and reliability to an oblivious stream of application messages can cause head-of-line blocking. While this is desirable for dependent messages, i.e., messages that have to be processed in order, it is undesirable for independent messages. To avoid this issue, independent messages should not be multiplexed into the same communication unit of a protocol that implements ordering and reliability. **SCTP** and **QUIC** provide multiple streams to separate independent messages into individual streams.

RTP does not suffer from head-of-line blocking as it only implements (re)ordering and hands off the data to the applications based on deadlines.

Signaling/address resolution protocols such as **SIP** or **DNS** do not need ordering, but reliability, which is usually done by re-transmitting the original messages after a short timeout. As these messages are idempotent, duplicate messages can be tolerated.

IP needs to re-order fragments to re-assemble them but does not expose this mechanism as transport service. See more discussion about IP fragmentation issues in Section 2.10.4.

The data-link layer of some paths might also implement ordering or reliability. Reliability on the data link layer is useful as a performance enhancement for media with a relatively high packet loss or bit error rate. Ordering protection at this layer is sometimes applied to prevent performance degradation when providing transport services to TCP, as TCP misjudges packet reordering for packet loss. QUIC and SCTP are resilient against minor reordering.

2.10.3 Integrity, Confidentiality, and Authenticity Protection

In the beginning of the Internet, all routers and endpoints were considered trusted. Therefore, there was no need for confidentiality and integrity protection and this functionality was added to the Internet protocol suite later using extensions.

The most prominent of these extensions is SSL and its successor **TLS**. TLS realizes all three mechanisms: integrity, confidentiality, and authenticity protection within a separate layer that slides between application and transport layer. Besides the stream based **TLS**, there is a message based variant **DTLS** that does not require ordering and reliability as transport service. The main argument for TLS and DTLS are easy deployment, as they are usually realized as a library and do not require OS support. They also give applications full access to the information of the authentication protection and, thus, allows tight integration with application-specific security protocols.

The second prominent security add-on protocol family is **IPSec**, which slides between the transport protocol and IP[12]. IPSec comes in two variants, Authentication Header (AH), which only provides integrity protection, and Encapsulating Security Payload (ESP), which provides both, integrity and confidentiality protection. IPsec offloads the authenticity protection into a separate protocol — Internet Key Exchange (IKE). IKE performs a handshake and verifies the authenticity of the endpoint(s), e.g., using cryptographic certificates, shared keys or an external directory and, then, negotiates cryptographic keys to be used for confidentiality and integrity protection of the session. Separating the authenticity protection protocol makes the implementation of integrity and confidentiality protection much simpler. Indeed,

[12] We only look at IPsec in transport mode — We model IPsec in tunnel mode as an additional path.

also TLS separates the authenticity protection protocol internally. In case of IPsec, confidentiality and integrity protection is usually implemented in kernel space and can benefit from hardware acceleration.

Other transport protocols that integrate integrity and confidentiality protection are **QUIC** and **SRTP**. QUIC uses the TLS authenticity protection protocol for authenticity protection. SRTP relies on a signaling protocol like SIP for key exchange. See [55] for an extensive discussion of integrity, confidentiality, and authenticity protection as transport service.

IPsec can secure any transport protocol and allows multiple communications between two endpoints to share a single cryptographic secession. Still, if a separate treatment by on-path elements is required for some communication units, these can be made distinguishable by choosing separate SPIs and using different flow labels in IPv6.

In today's OSes, there is no infrastructure that allows applications to access authenticity information from IPsec, making it infeasible for most use cases.

2.10.4 Chunking

Many transport services limit the maximum size of a PDU that can be transferred (MTU). To transfer larger communication units, chunking, as described in Section 2.9.8, has to be applied to adapt the message size to the MTU of the transports. Therefore, protocols can provide transport services that allow using arbitrary communication units. As this is extremely useful, almost all transport protocols, except UDP, support chunking for this purpose. To choose a sensible chunk size, many protocols perform *path MTU discovery* and adjust the chunk size according to its result. Aligning the messages of upper layer protocols with chunk boundaries can further improve performance: **RTP** chunks media streams at their message boundaries to perform congestion control and rate adaptation in an application-aware manner.

Historically, IP supports chunking of IP datagrams; this process is called **IP fragmentation**. As many transport protocols already support chunking, these protocols try to avoid IP fragmentation to avoid double-chunking problems described in Section 2.9.8. They only rely on IP fragmentation in cases where path MTU discovery fails, e.g., because of firewalls that block the ICMP "Fragmentation Needed" errors. For IPv6, the IETF dropped IP fragmentation form the basic IPv6 transport services and moved its functionality into an optional extension.

Finally, the general concept of splitting a communication unit into smaller pieces can also be applied in other contexts. **HTTP** allows limiting requests to byte-ranges of the requested objects. Applications can use this technique in many ways, e.g., to continue the download of a file, to download multiple chunks if a file from multiple endpoints or download them over multiple paths. The Mobile-IP variants **NEMO** and **IFOM** can chunk the association set, represented by source address and destination address in smaller association sets (based on IP five-tuples, three-tuples or six-tuples) to route tier PDUs over different paths.

2.10.5 Multiplexing

We define multiplexing in Section 2.9.7 as merging fine-grained communication units into a stream or association of a corse communication unit. Despite looking like a corner case, the most prevalent use case for multiplexing in the Internet protocols is addressing. IP uses the next-header to multiplex IP PDUs to the same IP address towards different transport protocols. Most transport protocols use port numbers to multiplex messages using the same transport protocol to the different endpoints (processes) using the same transport protocol. Some protocols do not use port numbers; instead, they multiplex on a session basis, e.g., **IPsec** based on the *Security Parameter Index (SPI)* and **QUIC** on base of the **Connection ID**. **TLS** and **DTLS**, and therefore also QUIC, allow further addressing based on optional "server name" and "service" [13].

The second use case of multiplexing is merging multiple messages or streams into one association, as done by **SCTP**, **RTP**, and **QUIC**. This allows separating independent streams of communication, e.g., to prevent head-of-line blocking, without requiring relatively expensive association setup and allows shared state keeping for the whole association.

2.11 Cost and Granularity Tradeoffs

Cost and complexity of transport option selection depends on the number of transport options, the network state used and, other actions that need to be performed either to determine the transport options available, determine their properties or to compose them into transport configurations. If the information about a transport option is easily available and only use local state, e.g., link availability, the cost may be negligible. An example of such a transport option is using *Equal Cost Multipath (ECMP)*. In other cases, the cost can be non-trivial, e.g., when determining transport options involves queries to remote entities or determining the properties transport options requires active network performance measurements. Examples for these include determining endpoints using DNS or DHT lookups, as used by some file sharing protocols. Such expansive mechanisms should be used rather infrequently because costs may become prohibitive if used too often. Thus, to perform transport option selection for fine grained communication units, it is advisable to cache results of transport option selection or to use a two-phase-process. The latter determines transport option candidates and determines their properties in the first phase, and selects between these candidates in the phase.

[13] Primary use-case is Server Name Indication (SNI) introduced to allow to run multiple HTTPS servers on the same IP address and TCP port.

2.12 Conclusion

In this chapter, we revisit the design of the Internet protocol stack from different perspectives to understand and systematize the transport options it provides. By focusing on communication units that have a semantic for the application, we analyze on which communication granularity optimizations can be applied. By looking at the PDUs, we derive which protocols, in principle, can be layered on to of each other. Using this perspective, we analyze the three dimensions of transport diversity: *path selection*, *endpoint selection*, and *protocol stack composition*. To approach the latter, we identify a set of building blocks that can be used to compose transport services provided by Internet protocols — our transport mechanisms.

The results of this analysis is the basis to tackle transport option selection in the remainder of this thesis. We find a diverse set of protocols that provide transport options at almost all granularities. We also often see that different compositions can provide the same transport service functionality.

What our analysis does not cover is the history of the protocols, their implementations, and the implementations of the paths. In fact, these factors dramatically limit the usability of protocols and constrain the layering of protocols. In practice, the latter results in designing protocols into the most easily usable ecosystem instead of using the best technical solution which may be standardized, but not easily available. Because of this, most applications on today's Internet are layering HTTP over TCP, even in use-cases where alternatives like SCTP would address the applications' needs much better. From a communication unit perspective, this also means forcing message granularity communication into a stream abstraction and, thus, preventing further optimizations.

In the next chapters, we explore building blocks for OS based transport option selection which does not require any change in the protocols used in the internet, but only changes at the client side OS. We show how to enable applications to seamlessly take advantage of the protocol diversity in the Internet and prepare for transport evolution.

3

Socket Intents: Expressing Applications' Intents

In Chapter 2, we look at *what transport option selection can choose from in order to optimize application performance*, by analyzing the design space for client based transport option selection. In this chapter, we approach the problem *what transport option selection should optimize for*. Therefore, we introduce the concept of *Socket Intents*.

> Socket Intents allow applications to share their knowledge about their communication pattern and express performance preferences in a generic and portable way.

We first published the concept of Socket Intents alongside with an early version of the system presented in Chapter 6 at CoNEXT'2013 [1]. In this chapter, we describe the general concept of Socket Intents in the version we tried to standardize in the IETF [6] and may become part of a joint effort to build a new transport API [11].

This chapter starts with explaining the motivation behind Socket Intents in Section 3.1, defining a problem statement in Section 3.2 and describing the resulting concept and what kind of information Socket Intents should provide in Section 3.3. In the remainder of this chapter, we define an extensive, but not exhaustive, set of Socket Intents types applications may provide in Section 3.4, discuss related work in Section 3.6 and the implications of a wide deployment of Socket Intents in Section 3.7. Finally, we conclude this chapter in Section 3.8.

3.1 Motivation

To illustrate the need for Socket Intents, we get back to the examples from the introduction (Chapter 1): Examples of common applications that have different communication needs:

- For video streaming applications, such as Youtube, bandwidth is most crucial.
- For voice calls packet loss and latency are important.
- Push notifications channels should be resilient and energy efficient.
- Software updates should afflict the lowest cost possible.

Yet, using today's socket API, all of these applications look the same to the OS.

There is no way for the OS to support exploiting transport diversity in a way that provides each of these applications the transport configuration that suits their needs.

Moreover, there are also examples beyond the design space for transport option selection we explored in Chapter 2 that benefit form information about ongoing communication:

- The TCP Nagle Algorithm avoids sending small TCP segments by coalescing them into larger ones. While it improves the throughput of many applications, it has a negative impact on the user experience of interactive shell session.
- Knowing the data rates of the videos streamed enabled Youtube to de-peak their traffic and, therefore, prevents packet loss without changing the data rate [61].

For most applications, exploiting transport diversity and tuning transport protocols by using their own heuristics is infeasible: As outlined in Chapter 2 the design-space for transport option selection is huge. Obtaining the necessary information is difficult since special privileges are required on most platforms to access this information. In addition to that, choosing a path or tuning advanced protocol parameters usually requires using non-portable APIs or APIs that are unavailable in most programming languages. Consequently, the available access network diversity is usually not exploited.

Therefore, we need a way to communicate the applications' intents to an entity, that is able to deal with the complexity of transport option selection and where all communication and transport options are visible: the OS. Having information about the applications' needs at OS level — on the finest communication unit granularity possible — is the key enabler to automatically exploit the diversity as part of the OS's transport service. It allows matching communications to the most suitable interfaces and jointly optimizing traffic across applications.

3.2 Problem Statement

Application programmers opening a communication channel typically know how this channel will be used. In addition to the protocol and destination address needed to establish a communication channel, there typically is more information available: An application developer knows or has an intuition of many aspects of an upcoming communication, which may include:

preferences whether to optimize for bandwidth, latency, or cost
characteristics expected packet rates, byte rates or how many bytes will be sent or received.
expectations towards path availability or packet loss
resiliences whether the application can gracefully handle certain error cases

These preferences, expectations, and other information known about the upcoming communication should be expressible in an intuitive, generic way, that is independent of the network- and transport protocol. Its representation should be independent of the actual API used for network communication and should be expressible in whatever API available, e.g., as *Socket Options* for BSD sockets, as *transport properties* for the TAPS API [11].

Socket Intents should enable the OS to adjust the communication channel according to the application's intents in a best-effort fashion. They should provide the information needed for automated transport option selection. The actual implementation is not part of the Socket Intents concept. It is realized as an OS policy, that takes care of choosing the transport options the application can benefit most, e.g., the most suitable protocol stack composition, endpoint and combination of paths.

3.3 Socket Intents Concept

Socket Intents are pieces of information that allow an application to express what it knows about its communication. They indicate what the application wants to achieve, knows, or assumes general, intuitive terms. As shown in Figure 3.1, an application can use them to annotate the characteristics, preferences, and intentions it associates with towards the OS via the Socket API. Since this information captures the intents of an application and passes them along with the communication socket, we call these pieces of information Socket Intents.

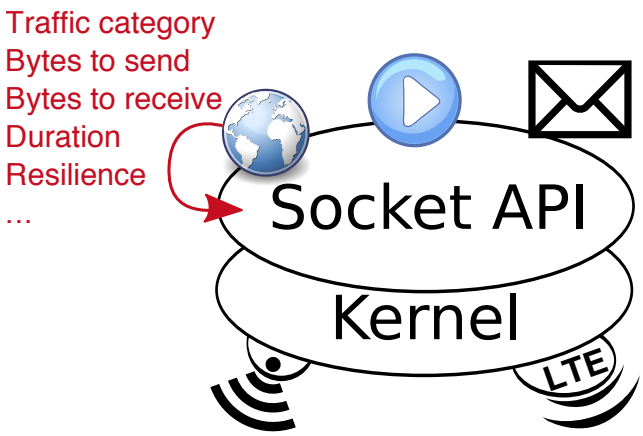


Figure 3.1: Socket Intents passed to OS via the Socket API.

Socket Intents are *optional information* and are considered in a *best-effort* manner. This strictly discerns them from all kinds of requirements, such as mandatory transport functionality as reliable in-order delivery, or bandwidth, delay and jitter requirements expressed through Quality of Service (QoS) style reservations. Therefore, the difference between QoS and our Socket Intents approach, can also be summarized as, "[…] the application tells what it *knows* as opposed to what it *wants*, as in prior work on QoS" (CoNEXT'2013 TPC member).

Examples for types of information include number of bytes to send/receive, the bitrate and duration of a stream, or the preference whether to avoid traffic or time accounted paths. The latter example suggests that specifying Socket Intents may result in connection failure, but other than that, they should not require changes in the application logic.

Applications have an incentive to specify their intents as accurately as possible to take advantage of the most suitable existing resources. We expect applications to selfishly specify their preferences, but since the OS knows about the intents of multiple applications and about the available network resources, it can balance the different requirements. It is up to the OS's policy to prevent commitment of excessive resources suggested by intents, e.g., by checking the accuracy of the intents specified after each communication unit and penalize misbehaving applications.

3.4 Socket Intent Types

Socket Intents are structured as key-value-pairs. The key is a simple string representing the type of a Socket Intent. Values can be represented as *enum*, *int*, *float*, *string* or, a sequence of the aforementioned data types. Implementations determine how these types are represented on the respective platform. Table 3.1 gives an overview of Socket Intent types we specify in our IETF draft [6] and the communication units, as defined in Section 2.3.3, they apply to. In the remainder of this section, we briefly describe these Socket Intent types. The levels of the *enum* Socket Intent types are concluded in Table 3.2.

Traffic Category The Traffic Category describes the dominating traffic pattern of the respective communication unit expected by the application. Most categories suggest the use of other intents to further describe the traffic pattern anticipated, e.g., the *bulk* category suggesting the use of the *Size to be Sent* intent or the *stream* category suggesting the *Stream Bitrate* and *Duration* intents.

Cost Preferences This describes the Intents of an Application towards costs cased by the respective communication unit. It should guide the OS how to handle cost vs. performance and reliability tradeoffs.

Size to be Sent / Received This Intent is used to communicate the expected size of a transfer.

Duration This Intent is used to communicate the expected lifetime of the respective communication unit.

Stream Bitrate Sent / Received This Intent is used to communicate the bitrate of the respective communication unit.

Table 3.1: Socket Intents Types

Intent Type	Data Type	Applicable Granularity			
		Message	Stream	Assoc	Assoc.Set
Traffic Category	Enum		✓	✓	✓
Cost Preferences	Enum	✓	✓	✓	✓
Size to be Sent	Int (bytes)	✓	✓	✓	✓
Size to be Received	Int (bytes)	✓	✓	✓	✓
Duration	Int (msec)		✓	✓	✓
Bitrate Sent	Int (bytes/sec)		✓	✓	✓
Bitrate Received	Int (bytes/sec)		✓	✓	✓
Disruption Resilience	Enum	✓	✓	✓	✓
Timeliness	Enum	✓	✓	✓	✓
Burstiness	Enum		✓	✓	✓

Table 3.2: Socket Intents Types – Enum Values

Intent Type	Enum Values	Description
Traffic Category	query	Single request / response style workload, latency bound
	control	Long lasting, low bandwidth, not bandwidth bound
	stream	Stream of bytes/messages with steady data rate
	bulk	Bulk transfer, huge messages, bandwidth bound
	mixed*	Don't know or none of the above
Cost Preferences	no expense	Avoid expensive transports, consider failing otherwise
	optimize cost	Prefer inexpensive transports, accept service degradation to save cost
	balance cost*	Do not bias policy default when balancing cost
	ignore cost	Ignore cost, choose transport solely based on other criteria
Disruption Resilience	sensitive*	Disruptions result in application failure, disrupting user experience
	recoverable	Disruptions are inconvenient for the application, but can be recovered from
	resilient	Disruptions have minimal impact for the application
Timeliness	stream	Minimize delay and packet delay variation
	interactive	Minimize delay, some variation is tolerable
	transfer*	Delay and packet delay variation should be reasonable, but are not critica
	background	Delay and packet delay variation is no concern
Burstiness	no bursts	Application sends traffic at a constant rate
	regular bursts	Application sends bursts of traffic periodically
	random bursts	Application sends bursts of traffic irregularly
	bulk	Application sends a bulk of traffic
	mixed*	Don't know or none of the above

* default value

Disruption Resilience This Intent describes how an application deals with disruption of its communication, e.g., connection loss. It communicates how well the application can recover from such disturbance and can have implications on how many resources the OS should allocate to failover techniques for this particular communication unit.

Timeliness This Intent describes the desired delay characteristics for this communication unit. It provides hints for the OS whether to optimize for low delay or for other criteria. There are no hard requirements or implied guarantees on whether these requirements can actually be satisfied.

Burstiness This Intent describes the anticipated burst characteristics of the traffic for this communication unit. It expresses how the traffic sent by the application is expected to vary over time, and, consequently, how long sequences of consecutively sent packets will be. Note that the actual burst characteristics of the traffic at the receiver side will depend on the network. This Intent can provide hints to the application on what the resource usage pattern for this communication unit will look like, which can be useful for balancing the requirements of different application.

3.5 Usage Examples

Based on the Socket Intents types we defined in Section 3.4, we describe three use cases in which different Socket Intents can benefit applications in different ways.

3.5.1 OS Upgrade

Consider a cellphone performing an OS upgrade. This process usually implies downloading a large file. This is a bulk transfer for which the application may already know the file size. Timing is typically noncritical and the data can be downloaded as background traffic with minimal cost and power overhead. It does not hurt if the TCP connection is closed during the transfer as the download can be continued.

For this case, the application should set the *Traffic Category* to *bulk*, *Timeliness* to *background*, and *Application Resilience* to *resilient*. In addition, *Size to be Received* can be provided. Finally, the application may set the the *Cost Preferences* to *no expense*.

The OS can use this information and therefore may schedule this transfer on a flaky but not traffic-accounted WiFi link and may reject the connection attempt if no cheap access link is available.

3.5.2 HTTP Streaming

Consider a user watching non-live video content using MPEG-DASH [62]. This usually means fetching a stream of video chunks. The application should know the size of each chunk and may know the bitrate and the duration of each chunk and the whole video. Disconnection of the TCP connection should be avoided because that might have an effect that is visible to the user.

For this case, the application should set the *Traffic Category* to *stream*, the *Timeliness* to *stream*, and *Application Resilience* to *sensitive*. It may also provide the *Stream Bitrate Received* and *Duration* expected. Finally, the application may set the the *Cost Preferences* to *balance cost*.

The OS can use this information and, e.g., use MPTCP if available to schedule the traffic on the cheaper link (e.g., WiFi) while establishing an additional subflow over an expensive link (e.g., LTE). If the desired bandwidth cannot be matched by the cheaper link, the more expensive link can be added to satisfy the desired bandwidth.

If the application sets the *Cost Preferences* to *optimize cost*, the OS would not schedule traffic on the second subflow and the application has to reduce the video quality to adapt to the available data rate.

3.5.3 SSH

Consider a user managing a remote machine via SSH. This usually involves at least one long-lived console session and possibly file transfers using SCP or rsync multiplexed on the same association (e.g., a TCP connection).

For the packets sent for the console session, the application can set the *Traffic Category* to *control*, the *Burstiness* to *random bursts*, the timeliness to *interactive* and the resilience to *sensitive*. For the packets of the file transfers, SSH may set both, the *Traffic Category* and *Burstiness* to *bulk*. It may also know the size of the transfer and therefore sets *Size to be Sent* or *Size to be Received*.

Note that this use-case only works if either the socket interface supports setting the Socket Intents on a per message level or supports multiple streams in a single association (e.g., when using SCTP) and allows to set separate Socket Intents on them. Depending on the protocols available, the OS can use this information to schedule the streams over different links to meet their requirements (latency vs. bandwidth) or optimize the transport, e.g., by setting appropriate Differentiated Services Code Point [63, 64] (DSCP) values or by disabling TCP Nagle Algorithm for console session related transmissions.

3.6 Related Work

There is some previous work on applications specifying their requirements and needs. Most of them focus on QoS, e.g., **QSockets** [65], rather than using the best-effort approach of Socket Intent.

The term *Intents* has its origin in **Intentional Networking** [66], an attempt to explore mobile network diversity by letting applications specify traffic characteristics via an extended Socket API. However, they cannot support major Internet protocols like HTTP because they only support message granularity, which is sufficient for their use-case. Moreover, they imply guarantees while we suggest best-effort use.

The **NEAT Transport API** [67] allows applications to set *neat properties* to communicate their communication needs. The properties used within their prototype are mostly used as requirements, but their *capacity profile* property allows similar optimizations as our *category* intent. The NEAT project was started after our original Socket Intents paper [1] and collaborated with our project.

3.7 Discussion

The introduction of Socket Intents into the transport system may have various effects on the OS and the network. In this section, we discuss these effects – beginning with discussing how the API interaction changes, we will move though various aspects until we look at possible effects on the Internet's traffic pattern.

3.7.1 Socket Intents and API behavior

When called with the same set of parameters, behavior of the Socket API is very consistent, even across different platforms. While its parameters take terminology from the filesystem and inter process communication (IPC) domain, their effect on the the network protocols is fully predictable and does not depend on external parameters. For example, requesting a socket of domain `PF_INET` with type `SOCK_STREAM`, results always in a TCP socket, despite that SCTP could provide the same transport service.

With Socket Intents, this base assumption changes: Intents, by their nature, are interpreted by the transport system and setting them will have different outcome depending on the device, environment and external parameters, such as interface bandwidth. While this is a big shift of the API contract, because of the best-effort nature of the internet, it is only a small change of the expected behavior of the network. Still, it is unclear which developers welcome this shift of paradigm from a consistent, application agnostic towards an intent-aware, application adaptive API behavior.

In addition, it is necessary to clearly separate parameters that are interpreted from parameters that have consistent outcome. Failing to do so could be a slippery slope towards interpreting all parameters and making the API unusable for application that rely on consistent API behavior.

3.7.2 Applicability of Socket Intents to different Communication Units

At the beginning of this chapter, we state that an application should communicate its intents on the finest possible granularity. When considering the design space, as done in Chapter 2, this is indeed true. When considering implementation strategies of the individual intents, the question at which granularity Socket Intents are most useful depends on the available transport options and the implementation strategy. In this section, we will discuss different transport features and how they can incorporate information provided by Socket Intents.

The availability of fine-grained intents is most useful for message based transports like UDP or HTTP (at request level). In these cases, the transport can perform path and endpoint selection on a per-message level unless the application protocol relies on stable endpoints.

In cases like HTTP, the cost of establishing new connections is significant. Therefore, re-using connections whenever possible is crucial and the cost of opening a new connection to choose different transport options often outweighs the benefits that can be achieved by application aware transport option selection. In these cases, strategies that allow path- and destination selection without establishing a new connection for each request are advisable (see also Section 2.5.1). To achieve these goals we anticipate two strategies: The first involves using a connection pool that holds different transport configurations towards the same destination and establish/re-use the most suitable one. The second strategy, which is limited to path selection, involves using a multi-path aware transport protocol and assigns individual messages to paths that are preferable for the given intents. To prevent head-of-line blocking, this protocol should also support multi-streaming. Because of these requirements, no "widely" available multi-path transport today is suitable for this: MPTCP is neither message-aware nor does it not support multi-streaming and SCTP's multi-path support is only intended as a fallback. However, we expect multi-path QUIC to support both requirements soon.

Finally, in cases where a stream-based transport is used to transport messages without using a special connection reuse / scheduling scheme, the intents need to be known when the stream transport is connected and will have no effect on transport option selection when provided at message sending time.

3.7.3 Interactions between Socket Intents and QoS

After presenting the SocketIntents concept at the 99th IETF Meeting in Prague, a widely heard feedback was that Socket Intents were anticipated as a kind of "Best-Effort ATM remake". While some of the Socket Intent types defined in Section 3.4 have a direct corresponding QoS/Integrated Services (IntServ) property (Traffic Category, Stream Bitrate Sent / Received) that was used in ATM, their meaning is quite different: While in QoS/IntServ these properties are used to do mandatory reservations, and therefore as a means for admission control, Socket Intents are purely advisory: The application expresses what it knows about the communication, what the traffic might look like and what the application can tolerate in oder to help the transport system optimizing a best-effort transport on behalf of the application. Therefore, no admission control is performed, but the most suitable transport is provided, which might fail to meet the specified desired parameter. Even for the Socket Intent Types that suggest generating an error in case the requested transport service is not available, e.g., *Cost Preference*, this decision is local and not part of a distributed, network wide admission control.

Therefore, Socket Intents and QoS/IntServ are orthogonal concepts: An application should use QoS/IntServ in cases where it *requires guarantees*. If it wants *best effort service optimized for its needs*, it should use Socket Intents. While the former is not available end-to-end in today's Internet, the latter is always available, but might only perform as well as an un-optimized best effort service.

For the case of QoS/Differentiated Services (DiffServ), the relationship is more complex. Socket Intents like the *Traffic Category* and transport system heuristics based on Socket Intents like the *Size to be Sent/Received* do, indeed, imply DiffServ service classes as described in [68]. Still, this is meant in a best-effort manner.

Overall, the relationship between Socket Intents and both QoS concepts, IntServ and DiffServ, is a little confusing at first sight, but the distinction between both should be clear enough to enable an application programmer to use the concept that fits the application's requirements.

3.7.4 Security Considerations

New communication channels and API paradigm always raise concerns whether they open new attack vectors – Socket Intents are no exception to that. In particular, we anticipate two kinds of attacks that may be enabled by using Socket Intents.

Performance Degradation Attacks As stated at the beginning of Section 3.3, we assume that applications specify their preferences in a selfish, but not malicious way. It is up to the OS to find a compromise between demands. A malicious application could confuse the OS in a way that leads to scheduling traffic with certain Intents on a more expensive interface, penalizing this traffic, or even rejecting it. We consider the additional risk of this attack vector negligible: As the malicious application could

also generate the traffic it claims to intent, it already has a much more powerful attack vector. As a mitigation, the OS could monitor and compare the intents specified with the traffic actually generated and notify the user if the usage of Socket Intents is unusual or defective.

Information Leakage An implementation may expose different protocol parameters on the PDUs to request special network treatment that matches the specified intents. This may allow to gain information about streams or messages multiplexed in the same encrypted association. These distinct parameters can enable an attacker to gain some ground truth about the shares and timings of different kinds of traffic. Therefore, application developers and policy implementors have to weight the small additional information disclosure against the possible performance gains. Using Socket Intents on Association level can be considered safe.

In addition, if used in conjunction with connection pools or multi-streaming protocols as described in Section 3.7.2, the scope in which connection pools or multi-streaming connections are shared matches the protection domain. For example, in a Web browser, this scope needs to be limited to a window/tab and and origin to prevent other web-sites to gain insights on the user's recent browsing activity through a timing channel.

3.7.5 Interactions between Socket Intents and Traffic Pattern

While Socket Intents are only communicated to the local OS and therefore only influence local decisions on the client, these decisions can have global effects. By enabling the OS to optimize application performance using transport option selection, the diversity of the transport options used is likely to increase.

For the protocol dimension, that means that the strong dominance of TCP in the internet traffic mix may become weaker and protocols like SCTP and QUIC may be used more often. While these protocols are designed to be fair against TCP flows, there might be scale effects that have not been observed so far.

In case of the path and destination dimension, the outcome of a wide-spread Socket Intents deployment is mostly dependent on the actual policies that implement path and destination selection. Still, this will most-likely shift a reasonable amount of traffic from the default path and destinations used today towards paths and destinations the policy assumes more appropriate for the specific communication unit. This may put additional load on cellular networks (in cases where WiFi is used as default today) and may interfere with server load balancing strategies used by CDNs today. Depending on the quality of the policy, this will either distribute traffic more evenly and allow to move traffic from congested links more quickly or cause congestion on low-delay paths if the policies feedback mechanism is too slow or non-functional. In the end, the OS's transport option selection policy may become a component of the transport system that is as crucial to work correctly as congestion control is today.

3.8 Conclusion

In this chapter, we introduce the concept of Socket Intents, their design rationals, their usage, and discuss their effects on the OS and network. Socket Intents are a great building block to make the transport system / OS smarter but need adoption from application developers and OS vendors to prove useful.

From all aspect of this thesis, the concept of Socket Intents has the highest impact so far. The concept of Socket Intents already influenced other work, including NEAT [67] and Post Sockets [69]. The concept has become a contribution towards the ITEF TAPS working group and has a good chance of becoming a component of future Socket APIs.

4

Policy: Choosing Transport Options

Transport diversity is no advantage per se — it is an opportunity to optimize the transport service for the individual applications' needs. To approach this problem, we first need to understand the design space, which is described in the previous chapters: Chapter 2 explores the dimension of transport diversity and characterizes what transport options this diversity provides and outlines how transport diversity manifests in Internet's protocols. In Chapter 3, we demonstrate a way to formulate applications' communication needs and introduce a mechanism to describe them. In this chapter, we finally discuss *how to choose among transport options* and present a generalized *policy framework* to realize this.

As a general requirement, our policy framework should enable all relevant stakeholders in the system to express their preferences or constraints towards certain transport configurations. These stakeholders include *the system administrator* of an administrative domain, *the user* and *system vendor* of a specific device, the *provisioning domain* of a path, as well as *application vendors*. Examples for those interests include:

- A user prohibits the game "Fluffy Pufferfish" to use paths using cellular.
- A system vendor wants software updates larger than 256 KB to prefer paths that are "cheap".
- An LTE network prefers IMS voice traffic using their platform to be routed via their LTE network.
- An application vendor wants the control traffic of its application to prefer SCTP or MPTCP over TCP.
- A company's system administrator prohibits its CRM system to use any paths except the company's VPN.

In order to explain the general structure of our policy framework, we start with a naïve approach that describes the basic four steps that are needed in order to perform transport option selection:

1. Determine the available transport options and their properties.
2. Eliminate transport options that are prohibited by policy entries.
3. Rank the remaining transport options based on the stakeholders' interests and application's needs.
4. Choose the best transport option.

In practice, this naïve four-step approach does not suffice. To reliably choose among transport options, we have to respect the dependencies between transport options.

Not all properties, e.g., path characteristics, are readily available and we cannot wait for results without impacting user experience. Therefore, we have to apply heuristic. Moreover, we want to try the best-ranked transport options in parallel to compensate for transport options that do not work and reduce the impact of imprecise heuristics.

The remainder of this chapter is structured as follows: To derive the requirements for the policy framework, we first discuss the dependencies that the transport option selection has to take into account when building and ranking transport configurations in Section 4.1. Most of these dependencies are latency related and result in the need to parallelize the selection process. Then, in Section 4.2, we explain how to represent transport configurations in a tree based structure. Next, to express the stakeholders' interest towards transport option selection, we propose *policy entries* in Section 4.3. Policy entries are considered by our generic policy iff they match a transport option or transport configuration while their actions determine how to resolve trade-offs or purge depreciated transport configurations from the transport configuration tree. In Section 4.5, we use a generalized variant of Happy Eyeballs [49] to probe the most preferred transport configurations from the tree in parallel. Finally, we summarize our findings in Section 4.6.

The policy framework described in this chapter is based on the experiences from designing policies for our Multi-Access Prototype (Section 6.3.5) and for our Web Transfer Simulator (Section 5.2), Moreover, it has been shaped by discussions within the IETF Taps working group [70].

4.1 Policy Dependencies

To realize transport option selection within the OS, our policy framework has to interact with other components of the OS that provide input for the policy. These inputs include characteristics of all available paths — e.g., delay, bandwidth, and packet loss, as described in Section 2.5 — the protocols available locally (see Section 2.8), and name resolution to determine the available endpoints (see Section 2.7).

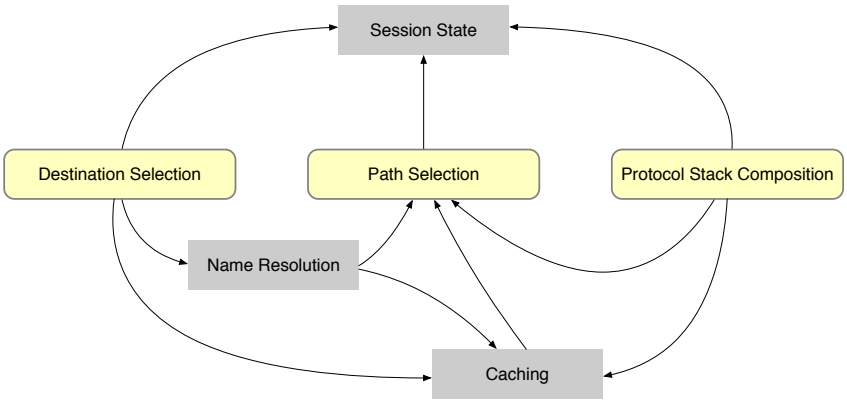


Figure 4.1: Dependencies between Transport options a Policy has to Respect.

Combining these components is non-trivial since there are dependencies between these mechanisms. Figure 4.1 gives an overview of the dependencies within transport option selection a policy has to respect. Most other parts of transport option selection depend on path selection because of the challenges and opportunities arising from on-path network functions — This has been already discussed in detail in Section 2.5.4. In this section, we focus on the dependencies for caching and the re-use of session state. Both are crucial for the user experience to minimize the delay incurred by transport option selection and therefore become a dependency. As we do not want to add delays to perform transport option selection, we have to parallelize the process. Therefore, we describe the dependencies and timing issues between the components that have to be addressed when parallelizing the process.

Name Resolution Name resolution depends on the path (see Section 2.7 for more discussion). Therefore, the endpoints derived through name resolution should only be used on the same path from which they were derived. In some cases, name resolution also requires input from protocol stack composition, e.g., for protocols that use `SRV` type DNS records to determine endpoints[1]. Therefore, name resolution is constrained by the protocols available locally.

In addition, name resolution takes at least one RTT (of the path). Therefore, the delay cost of waiting for name resolution on all paths to finish is the RTT of the slowest path. Thus, if timeliness is essential for the given communication, name resolution should not delay endpoint selection using other paths.

Local Session State Local session state, e.g., the existence of open TCP connections, can influence transport option selection– especially if done at message or stream granularity. The possibility to send a message using an already established connection to save connection setup time and dramatically reduce the latency for the application. This principle can be extended to other protocol layers as well. For example, if confidentiality or integrity protection is needed, it is useful to check if there is an already established IPSEC association or cached TLS state that can be used to perform a 0-RTT handshake. Moving transport option selection from the application to the OS can also enable using these optimizations across application boundaries.

Caching Caching is crucial for name resolution performance and transport configuration probing, especially when transport option selection is done for fine-grained communication units. Using cached results save at least one RTT of connection setup time each. If there are cached results for some transport options, the policy has to handle the trade-off between saving communication setup time and using the best transport option requiring non-cached state.

[1] The DNS query for an `SRV` record already contains the protocol stack to be used — a name resolution failure already allows excluding this protocol stack composition.

4.2 Determining Transport Configurations

In order to choose the most suitable transport configuration, we place the individual transport options in a data structure the policy can operate on. This data structure has to reflect the dependencies explained in Section 4.1: We let the dependencies between the transport options implicitly structure transport options into a tree, whereby each level of the tree corresponds to a dimension of choice [70]. The paths from the root to the leaves represent complete transport configurations.

Figure 4.2 presents a partial example[2] of such a transport configuration tree. The root of this tree is the destination representation provided by the application. This destination representation is annotated with the hard communication requirements (shown in red) as well as Socket Intents (shown in green). As protocol stacks and paths are independent, we choose, w.l.o.g., level 1 to represent protocol stack choices, level 2 to represent the path choices[3]. and level 3, to represent the endpoints derived from name resolution. All the nodes are annotated with the information available for the respective transport option: hard requirements (shown in red), optional external information like Socket Intents (shown in green), heuristically determined information (shown in purple) and protocol state (shown in cyan). In addition to the properties we discuss in Chapter 2, these can include any path, protocol, and endpoint specific information.

To construct the tree representation, we start at the root of the tree: Based on the application's requirement, we determine available protocol stack compositions, see Section 2.8, and place them on level 1 of the tree. Beneath, at level 2, we place the path candidates. Path candidates do not depend on the destination as we treat path characteristics as properties of the local interface representing the path (Section 2.5). Note that the filtering and ranking process, see Section 4.4, operates intertwined with the tree construction; the policy purges paths that are infeasible for the given destination before the next step. Name resolution has to be performed for each of the feasible paths. Once the name resolution has derived endpoints for a path, the corresponding nodes are placed into level 3 the decision tree. Finally, each path from the root to the leaves of the tree represents a feasible transport configuration.

While the policy ranks transport options and filters nodes from the three (see Section 4.4), we also start probing transport configurations (see Section 4.5) once sufficiently ranked transport configurations are determined. We discuss the details of these aspects in the next sections.

[2] Level 2 and 3 beneath the "QIIC/UDP/IPv6" protocol stack composition have been omitted to improve readability as they carry roughly the same annotations as the nodes beneath "TLS/TCP/IPv6".

[3] Since we consider the protocol stack choice, we differ from the decision tree in [70] which roots on the $(destinationrepresentation, service)$ tuple.

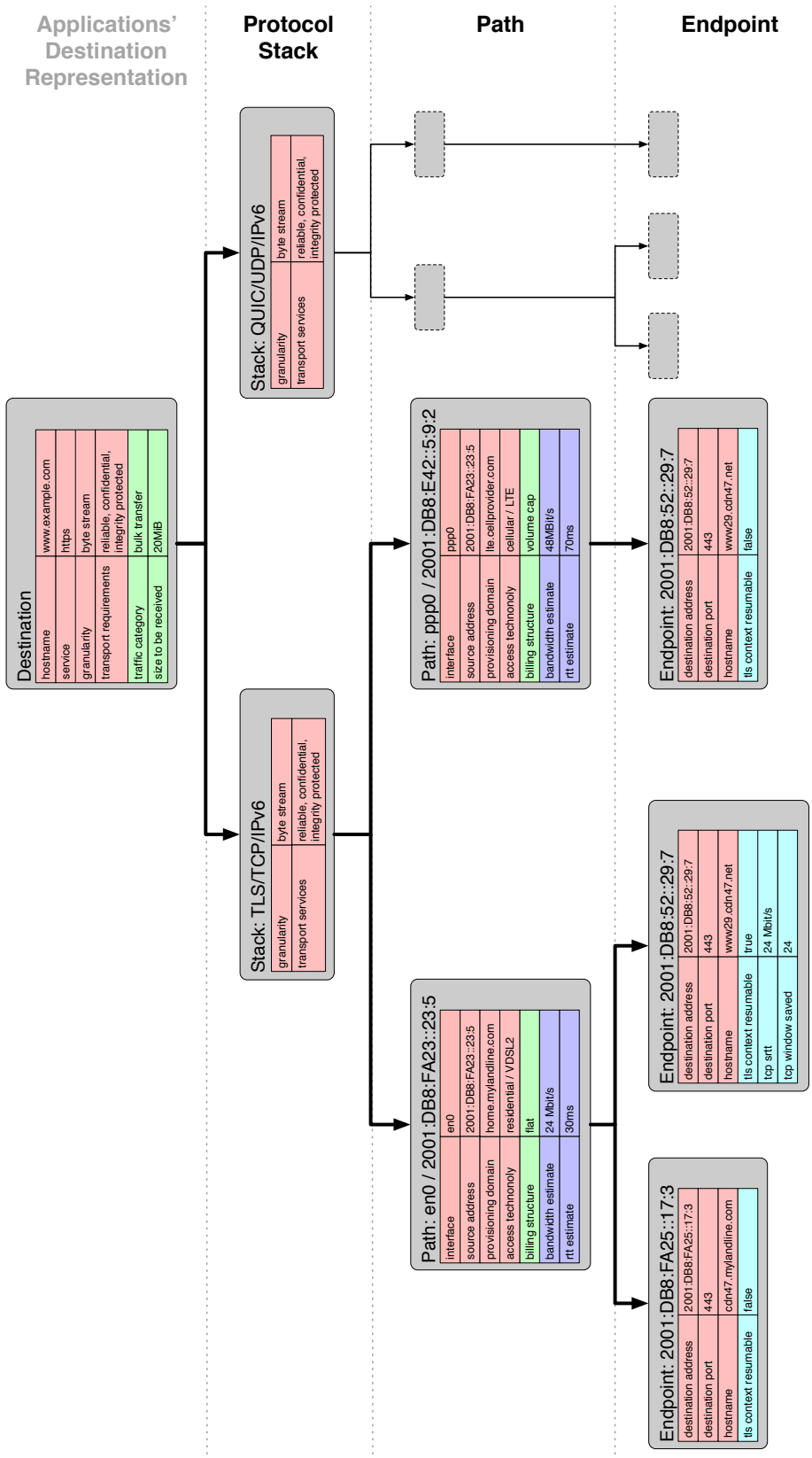


Figure 4.2: Partial example of a tree representation used by our generic policy framework.

4.3 Policy entries

In order to explain the ranking and purging process, we first introduce the concept of *policy entries*. Policy entries allow the stakeholders involved to express their preferences and constraints towards transport option selection. Therefore, the actual policy is being composed from a set of policy entries and expressed in a domain specific language.

Policy entries are tuples of the form (P, A), whereby P is a pattern that matches paths in the decision tree, and A is a set of actions to apply to matching paths. Policy entries are considered if they match a transport option or transport configuration while their actions determine how to resolve trade-offs or purge the matching transport configurations, while their effects cascade, i.e., incrementally modify the transport configuration tree introduced in Section 4.2.

A policy entry can match on almost any information present in the transport configuration tree. That means it can match on application requirements, Socket Intents, transport option properties annotated on the nodes and other system state reflected on the transport options including cache state and existing connections. In order to prevent blocking the evaluation, e.g., by requiring name resolution results of other paths, policy entries cannot reference or compare against other transport configurations (they are not allowed to look into other branches of the tree). This also prevents introducing circular dependencies. Comparison across branches is solely based on assigning a *weight* on them which is inherited by its children and is taken into account by policy entries matching later on and the probing process. We currently do not consider external services for the same reason — instead, we assume that information from these services is either available from system state or collected as part of the name resolution.

If a policy entry matches, it can perform any of the following actions on the matching branch of the transport configuration tree:

- Assign or change the weight on the matching branch.
- Purge branches of the tree, e.g., in case these transport configurations do not meet basic requirements or the combination of transport options is depreciated.
- Retain branches of the tree to override a purge request, e.g., to override policy entries provided by other stakeholders.
- Add annotations on nodes, i.a., to enabling other policy entries to use them or match on them.
- Tune the transport option within the respective transport configuration, e.g., set DSCP for IP or disable the Nagle Algorithm [71] for TCP.

To provide a sensible default behavior, we expect an OS to be shipped with a reasonable set of default policy entries, e.g., to prefer IPv6 over IPv4. Note that policy entries can also include requirements specified by application provided policy entries as well as configured policies, e.g., to force certain applications to use a VPN. Therefore, policy entries need to be applied in an order that guarantees that the

preferences of certain stakeholders can override the preferences of the others. To reduce complexity, we do not consider an additional priority mechanism. In the next section, we explain how policy entries are applied and how conflicts between actions are resolved to compose the actual policy.

4.4 Filtering and Ranking Transport Configurations

The actual transport option selection is done by filtering and ranking transport configurations. For each level of the tree from Section 4.2, this is done in two steps: First, our generic policy framework purges all transport options that do not meet hard requirements specified as a property of the destination (as shown in red on level 0 of Figure 4.2). Second, the matching policy entries are applied. To avoid latency, both steps are applied as early as possible.

When transport options are placed into the decision tree, hard requirements and matching policy entries are checked immediately and executed if possible. As actions of policy entries may conflict, it is necessary to check for conflicts with higher priority actions that may still match later on. This can happen in cases when a policy entry with higher priority matches multiple levels of a transport configuration while the conflicting lower priority entry only matches a lower level of the tree. For example, an application supplied policy entry may purge all transport options that use IPv6, but a user policy entry to prefer IPv6 for certain PvDs may cancel this action. In these cases, executing actions must be delayed and determining transport configurations must continue until the conflicting entry matches or does not match. Once all actions on a node are executed or marked as delayed, the next level of the tree can be populated.

After determining the transport options at all levels and applying all matching policy entries, the resulting decision tree only contains acceptable transport configurations. Also, each transport configuration/ leaf of the tree has a weight assigned that can be used as a preference for Happy Eyeballs on Steroids (HEoS). Still, the tree may still be incomplete as other transport configurations/ leaves can still be added by the determination process.

4.5 Probing Transport Configurations: Happy Eyeballs on Steroids

As stated earlier, purely relying on using the highest transport configurations is not sufficient. We can neither be sure that our path characteristics are accurate nor can we be certain that the transport configuration is practical, e.g., whether the derived endpoint is available and supports the chosen protocol stack composition. Yet, testing all transport configurations in order of their rank is also not sufficient, as even a short timeout of a few RTTs dramatically impacts user experience for many applications. Thus, we generalize the ideas of Happy Eyeballs [49] to parallelize probing

of transport configurations. Recall, Happy Eyeballs is a biased connection race that was originally developed as an IPv6 transition technology. It starts connection via IPv4 and IPv6 in parallel and uses the first connection. By giving IPv6 a few *ms* advantage, the "connection racing" is biased towards IPv6.

Instead of "'just"' trying to connect via IPv4 and IPv6 in parallel, our generalized[4] Happy Eyeballs on Steroids (HEoS) tries a small set of transport configurations and uses the weight from the ranking process as a bias for a connection race. Hereby, we use small weight-differences as time bias while large weight-differences will usually result in using the transport configurations as fallback only. Once one or more of the connections competing in the race are set up, HEoS picks the highest ranked connection that was established within a policy-determined time frame, e.g., within a few tenths of milliseconds.

As the connection racing starts immediately when the transport option that completes a transport configuration is placed in the tree, the time needed for name resolution already generates a bias towards endpoints with fast name servers. If this effect is not wanted, e.g., for an application that is willing to trade delay for bandwidth, HEoS should delay probing for transport configurations that are below a certain weight or add an initial delay before using the highest ranked transport configuration.

4.6 Conclusion

In this chapter, we complete the general discussion on how to exploit transport diversity. We introduce a generic policy framework that allows us to combine protocol stack composition, path selection, and endpoint selection without introducing unnecessary latency. In our framework, the policy that decides which transport options to use is composed from a set of policy entries which allow applications, the user, and other stakeholders to express the interests towards transport option selection. The composed policy is then executed by the OS.

While we provide this generalized policy mainly as outlook how transport option selection can be implemented, some of the ideas discussed in this chapter have already become state-of-the-art during assembling of this thesis. A simpler version of the transport option selection tree discussed in Section 4.2 is being standardized within the IETF TAPS working group [12] and (partially) implemented as part of Apple's next-generation networking API. However, a concrete representation of the policy entries, their application and their pattern of interaction to derive useful rankings is left for future research. Also, it is an open question how many parallel connection probes HEoS should attempt to choose decent transport options without putting too much load on the Internet and server infrastructure.

[4] Papastergiou et al. already show a version of Happy Eyeballs generalized for protocol selection beyond IP version [72], but do not include the two other dimensions of transport option selection.

5

Performance Study: Web Site Delivery

In the previous chapters, we focus on the potential of transport option selection and on the building blocks that enable exploiting transport diversity. In this chapter, we study an example use-case that (potentially) can benefit from transport option selection: Web browsing.

To quantify these benefits, we use a custom, flow-based *Web transfer simulator* to study the effects of different means to combine multiple paths on page load times. Our study uses the landing pages of the Alexa Top 100 and Top 1000 Web sites as a representative set of workloads and covers sets of a wide range of different network characteristics. For these network characteristics, we vary RTT and bandwidth of the two paths. The strategies either just use one interface or distribute the loading of the individual Web objects over the two paths. As a means of distribution, we either assign requests to paths or use MPTCP to use both paths at the same time. The two resemble the architecture for exploiting transport option selection used throughout the remainder of this thesis, we call the combination of scheduling strategy and means *policies*. Some of these policies make use of the *size to be received* Intent annotating the object size of objects to be received, as described in Section 3.4.

The remainder of this chapter is structured as follows: First, we explain the overall methodology (Section 5.1) and the rationals behind it. Then, we introduce the policies (Section 5.2) that model the scheduling strategies as a means to distribute the individual requests and discuss the workload (Section 5.3) used throughout this chapter. Afterwards, in Section 5.4, we present our Web transfer simulator used in this evaluation and validate it against our workload and the Multi-Access Prototype (Section 5.5). Finally, we put all together and evaluate the overall benefits of using multiple paths for Web site delivery in Section 5.6.

5.1 Methodology

Before discussing the details of our simulator study, we want to revisit the rationals and assumptions behind our study. The basic idea of this study is to evaluate the potential benefits of exploiting one dimension of transport diversity (multiple paths) and compare the effects of application-aware and application-agnostic policies. Therefore, we focus on using realistic workloads — the landing pages of the Alexa Top 100 and Top 1000 Web sites — and a basic network scenario — two paths towards a single server that share no common bottleneck. Within this scenario, we evaluate the effects of the policies under a wide range of network characteristics. Given this goals, we now revisit some central decisions with regards to the metric used in the evaluation (Section 5.1.1), the choice to using a custom simulator (Section 5.1.2) and the basic network scenario (Section 5.1.3). Afterwards, we highlight important details on how we model connection reuse (Section 5.1.4) and how we simulate TCP (Section 5.1.5) and MPTCP (Section 5.1.6) behavior.

5.1.1 Metric: Page Load Time

To evaluate the influence of our policies, i.e., path selection strategies, on the Web site delivery performance, we focus on page load time as a metric. We use the definition of the page load time as the time between the initial HTTP request and the completion of the last response. While the complete time to display a Web page also includes times for DNS resolution, page rendering and possibly client-side JS computation, these factors are not subject to our policies and we, therefore, exclude them from our evaluation. The page load time is one of the dominant factors of the time needed to display the Web page, and it is considered a reasonable approximation of the end-user Quality of Experience metric [73]. Related work criticizes using page load time as performance metric and suggests to use a different performance metric [74] that focuses on the load times of the Web page objects that have the highest user impact. We refrained from using the metric as it would require an extensive user study which would prevent us from using a large set of Web pages and repetitions. Still, our remaining methodology could be applied to these subsets as future work.

5.1.2 Using a Custom Simulator

Page load time is often measured as part of Web page profiling within a Web browser. While we use this approach in Section 6.4, for a smaller study, we could not use this approach for a large study covering a large set of Web pages, in our case, the Alexa Top 100 and Top 1000. This limitation originates from the structure of Web pages and the complexity involved: Page load events are triggered by DOM events (referencing/rendering of contents or driven by JavaScript). This makes the page load time dependent on the rendering and execution of JavaScript. As

JavaScript is turing complete and contains non-deterministic operators, the effect of these dependencies are theoretically undecidable and practically often heavily skew the page load time.

Depending on whether using the profiling tools in a testbed or on the internet, this results in different problems we can avoid by using a simulator:

- Experiments with mirrored Web pages in a testbed show that JavaScript often behaves differently than on the original page — this is especially the case for advertisements, as they often contact multi-level backend services or request randomized resources. This often causes Web pages to stall in the testbed and therefore renders the majority of the Alexa Top 100 Web sites useless for a testbed study. As the Simulator only uses a single dependency tree extracted from the Web browsers profiling data and models the downloads itself, we do not have to care about stalls and non-deterministic behavior.
- When moving from the testbed to the Internet, we must also take non-deterministic behavior into account. In addition to that, the contents of some Web pages change dramatically over time. Network conditions (RTT and bandwidth) also change, e.g., through time-of-day effects or changed content delivery network (CDN) configurations. Carrying out a study on the Internet would get realistic, but nor reproducible results for which we can hardly determine the influence of our policies. As the simulator gets a single dependency tree of Web objects for all network scenarios and policies and does not depend on external network conditions, we get fully reproducible results without external factors skewing the results.

In addition to these problems, the anticipated use of the *size to be received* Intent can be simulated much easier than it can be emulated within an instrumented Web browser : For the simulator, we can assume to know the sizes of the objects a priory. In the instrumented Web browser, we either have to probe them, as done in our testbed study in Section 6.4, or use cache the file sizes for known URLs.

While using a simulator has great advantages with respect to reproducibility, isolation of influencing factors, and complexity, these come at cost of realism. We acknowledge that our study ignores effects such as choices made by JavaScript code, interactions with Web browser and CDN optimizations. We also ignore all packet-level effects and may miss additional un-anticipated effects.

5.1.3 Network Scenario

Our network scenario is based on the assumption that, for mobile devices, the access networks almost always dominates the overall network performance, i.e., the bandwidth bottleneck is most likely located within the access network. Also, the access network often contributes the largest part of the path delay, e.g., due to bandwidth allocation and Forward Error Correction (FEC) schemes used within the access networks' data link layer. In comparison, Internet backbone delays are in the order of a few milliseconds while access delays are typically significantly larger.

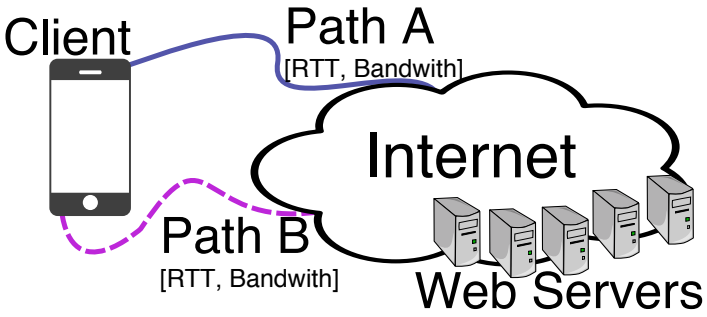


Figure 5.1: Simplified Network Scenario.

Thus, our network scenario, see Figure 5.1, consists of a client device, Web servers, and the paths between them. We presume that all Web servers are reachable via both network paths and that the paths share no bandwidth bottleneck. Moreover, we choose to neglect the RTT variability introduced by the Internet since queuing delays on Internet core links ($\geq$10 Gbit/s bandwidth) are negligible [31]. Therefore, we can capture the path characteristics as "interface" RTT and bandwidth.

5.1.4 Connection Limits and Connection Reuse

For multiple requests retrieved from the same server, our simulator supports persistent connections with and without pipelining. To decide whether pipelining is possible, we assume a separate server for each hostname. Pipelining is used whenever it reduces loading time for the respective objects, or when a new connection would be faster, but any of connection limits apply. It uses a default connection timeout of 30 seconds and limits the number of parallel connections per server to 6 and the overall number of connections to 17. These values correspond to the defaults of the browser we use to retrieve our workload. We acknowledge that the parallel connections somehow defeat the idea of TCP fairness on a per-connection level, but still provide some fairness as the number of parallel connections is limited.

5.1.5 TCP Simulation

We assume loading a Web site results in a lot of new connections to different hosts with only a small number of these connections being bandwidth limited. See Section 5.3.2 to see that our workload matches this assumption. Despite not being limited by the path bandwidth, some of these connections may be limited by *TCP slow-start*. To capture this effect of TCP slow-start, we go beyond a traditional flow-based simulation and include it in our simulation. In our simulator, a TCP connection starts in slow-start with a congestion window of ten times MSS.

Rather than fully simulating the congestion avoidance of TCP, we assume instantaneous convergence to the appropriate bandwidth share. Once the connection leaves slow-start by reaching the available bandwidth share, it never returns to slow-start. Our underlying assumption is that TCP tries to fairly share the available bandwidth between all parallel connections once TCP leaves slow-start [75].

5.1.6 MPTCP Simulation

Our simulation of MPTCP is rather simplistic: We simulate two regular TCP connections (subflows) and map bytes to be sent over the MPTCP connection to one of these subflows once the congestion window allows additional bytes being sent. Still, our simulation of MPTCP is fair against regular TCP: As our network scenario (Section 5.1.3) assumes the two paths have no shared bottleneck, only one subflow of each MPTCP connection has to compete with a regular TCP connection. Therefore, we do not need to implement coupled congestion control as coupled congestion control would not effect fairness.

5.2 Simulator Policies

In Chapter 4, we defined policies as entities that filter and rank transport options. In this chapter, we only focus on path selection based on different path properties. Therefore, for this chapter, we define policies as entities that decide for each communication unit which path to use. These policies still can range from simple, static configurations up to complex dynamic algorithms that try to take full advantage of the available information.

For deciding which path to choose, we present the *Earliest Arrival First* ($\mathcal{EAF}$ policy) — a policy that fits our use case of Web browsing: Objects of different sizes can be fetched over different interfaces and the download time largely depends on the object's file size as well as the RTT and available bandwidth on the path. Assuming that there are at least two access networks and they vary in RTT and bandwidth, our intuition is that if the communication unit is small, the policy should choose the interface with the shorter RTT. If the communication unit is large, it should prefer the interface with greater available bandwidth. Thus, each unit is scheduled on the interface with the earlier arrival time, resulting in shorter overall completion time. As input for the $\mathcal{EAF}$ policy, we use the *size to be received* Intent (see Section 3.4). The *size to be received* Intent allows an application, e.g., an HTTP client, to communicate the size of each object as a communication unit which is to be transferred. We also combine the $\mathcal{EAF}$ policy with MPTCP to split large objects and distribute their download over both interfaces.

To evaluate the benefits of our $\mathcal{EAF}$ policy for Web traffic, we realize them in our Web transfer simulator along with other basic policies used as a comparison baseline. Since the simulator tries to provide an upper bound of the benefits, it uses global knowledge about all currently active transfers and, in contrast to the $\mathcal{EAF}$ policy in Section 6.3.5, does not rely on heuristics. The RTT and maximum interface bandwidth, as well as the size of the objects for the *size to be received* Intent, are known a priori — differing from the $\mathcal{EAF}$ policy in Section 6.4, no two-step download is needed. Our simulator supports the following policies:

5.2.1 Baseline Policies

The *Single Interface* policy always chooses the path using a particular, statically configured interface. Therefore, this policy is equivalent to a client that only has the specific interface or only uses this interface. As we only have two paths in our scenarios, we name the instances of this policy after the interface they use *Interface 1* and *Interface 2*.

The *Round Robin* policy uses multiple interfaces on a round robin basis. It accepts the interface used for the first request (of a simulation) as a parameter.

5.2.2 MPTCP

This policy uses MPTCP with the full-mesh path manager across all interfaces. It presumes that MPTCP subflows can be opened on all local and remote interfaces. With two network interfaces at the client and one interface at the server, the policy establishes two subflows. The interface for the initial subflow is given as a parameter. This policy considers neither the Socket Intents nor the current network performance. For *MPTCP*, we considered two variants: starting the initial MPTCP subflow on the same statically chosen interface or always on a different, randomly chosen interface. The simulation of *EAF_MPTCP* is analogous to the EAF policy, but it includes predictions with MPTCP for all interface combinations, therefore using the interface for the first subflow that is predicted to give the best results.

5.2.3 Earliest Arrival First Policy

The *Earliest Arrival First* (*EAF*) policy variants use the *size to be received* Intent to predict the transfer completion time. For each communication unit, i.e., each HTTP request, the *EAF* policy predicts the transfer completion time for each available path or path combinations and chooses the one where the communication unit will arrive first.

For our first variant, *EAF*, our prediction is based on an estimation of the interface RTT, the available bandwidth and the other transfers already scheduled on this path: First, we estimate the available bandwidth. As we assume TCP fairness, we assume to get a fair share of the path bandwidth for each concurrent transfer. To approximate the download duration, we then divide the size of the communication unit by the estimated available bandwidth. This approximate is refined by taking TCP slow-start[1] into account. Additionally, we model the connection setup and request times: we add one RTT if a connection can be reused, or two RTTs if a new connection has to be established; we add two additional RTT for each TLS handshake.

[1] See Section 5.4.1 for details

While the first variant, *EAF*, distributes whole communication units, the *Earliest Arrival First with MPTCP* variant (*EAF_MPTCP*) combines the *EAF* Policy with MPTCP. In addition to predicting the arrival time for each path, it also considers MPTCP for all possible path combinations. The intuition behind *EAF_MPTCP* is that MPTCP is beneficial for some, not all cases. For example, this policy can avoid scheduling small communication units on a path combination that includes high RTT paths.

Overall, we expect that either *EAF* or *EAF_MPTCP* yields the best performance. The advantage of the former is that it can distribute the objects over multiple parallel TCP connections. The advantage of the latter is that it can distribute data at a finer-grained level than per communication unit/request granularity and choose a good interface for the initial subflow.

5.3 Simulator Workload

As we want to evaluate the benefits of different path selection strategies for Web browsing, the simulator needs a description of the Web pages as well as the dependencies among the Web objects as input. Accordingly, we next discuss which Web pages we focus on, how we extract the dependencies and the characteristics of the Web workload.

Since Web pages vary significantly in size, number of objects, number of servers, etc., we do not focus on any particular Web page but rather a diverse set of popular Web sites. Therefore, we acquired the landing pages of the *Alexa Top 100 Web sites* on 26 consecutive days starting on December 07 2015 and the *Alexa Top 1000 Web sites* on October 10 2016[2]. We focus on the mobile version of the pages by overriding the user-agent of our Firefox browser, impersonating a generic Android device[3].

5.3.1 Web Workload Acquisition

Due to the popularity of JavaScript, it is not sufficient to fetch the pages and all its embedded resources using tools such as *wget*. Therefore, we use *Firefox* version 38.4.0 automated with *Selenium*, as well as the *Firebug* 2.0.13 and *NetExport* 0.9b7 plugins to record the crawled Web pages in the HTTP Archive (HAR) format.

Each HAR file contains a summary of all objects of the page as well as their sizes, types, origins (remote sites), and timings. Since some HAR files can be malformed or contain no objects due to Web site downtime, connection loss, socket timeouts, or problems with the Firebug plugin, we trigger a repeated download of Web sites until we get at least one complete HAR file. We do not impose any network constraints mobile devices might experience. Our motivation is that we want to gather

[2] `http://www.alexa.com/topsites`

[3] We use the User Agent string `Mozilla/5.0 (Linux; Android 4.2.2; SOL22 Build/10.3.1.D.0.220) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/38.0.2125.102 Mobile Safari/537.36`

the dependencies within the Web page rather than restrictions imposed by limited network bandwidth. For the Web crawl, we use a single vantage point with a high available network bandwidth: A virtual machine within a university network. Due to the client's geolocation, the results may be biased towards Germany.

5.3.2 Web Workload Properties

To highlight how different the various Web pages are Figure 5.2 shows the empirical cumulative distributions of the number of objects per page as well as the number of hosts per page from our Alexa Top 100 crawl.

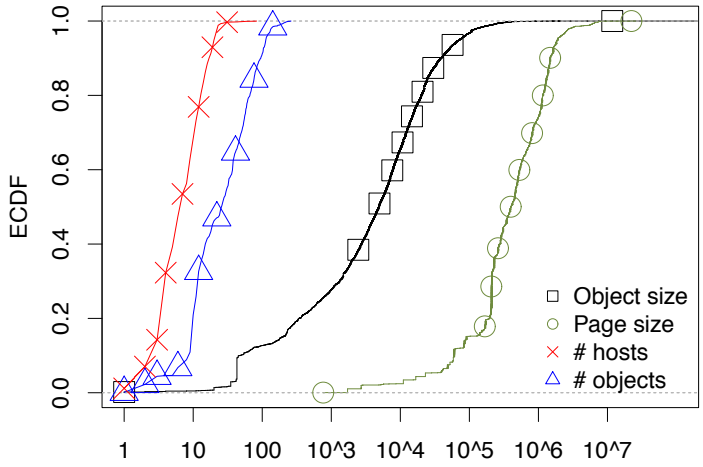


Figure 5.2: Web workload properties.

While most of the pages comprise between 1 and 50 objects, there are some with more than 100 objects or even up to 260 objects. Moreover, many Web pages have a low median object size. Furthermore, the number of hosts that have to be contacted ranges from a single one to more than 20 with a median of 7. Figure 5.2 also shows the empirical cumulative distribution of the individual Web objects as well as the total number of bytes per Web page using a logarithmic x-axis. Both distributions are highly skewed. In particular, we find that the object size distribution is consistent with a heavy-tailed distribution. The total size of the Web pages is between 23.1 KB (5th quantile) and 1.8MB (95th quantile) with a large fraction of pages below 300 KB. These results are in line with previous work [76, 77].

5.3.3 Web Object Dependencies

While identifying all objects of a Web page from the HAR files is straightforward this does not apply to the object dependencies. Some object dependencies are obvious from the base page, the HTML document, and the client-side DOM. However, JavaScript or other Web objects can modify the DOM, by adding or removing Web page objects, at any point during the page load. For example, when a Web site uses

JavaScript to load pictures dynamically, the simulator should not start downloading these pictures before the JavaScript object has been retrieved. After all, the browser first has to parse the JavaScript before it can download the pictures.

We decided against using sophisticated systems, like *Polaris* [78], since their focus is on finding the true dependency tree to speed up future downloads. Thus, using these dependencies often leads to much more optimistic results compared to the capabilities of current browsers. To ensure compatibility, we use a more conservative heuristic. We identify the dependencies from the download times contained in the HAR files. This method is feasible since we use a non-bandwidth limited client to gather the HAR files. Given that a browser cannot start a transfer when it has not yet downloaded the object it depends upon, we assume that Web objects that are downloaded in parallel do not depend on each other. The same holds for an object that has started to download while a previous object has not yet finished. If the first object is not yet done, it cannot initiate the transfer of the second one.

5.4 Web Transfer Simulator

To evaluate the benefits of seamlessly using multiple paths and scheduling requests according to our policies, we build an event-based Web transfer simulator. We choose to focus on a simulator rather than an emulator since it allows us to cover a much larger parameter space and allows us to experiment with more policies.

For each run, the simulator takes the following input: A Web page including all Web objects and their dependencies represented as a HAR file, see Section 5.3), the policy, and a list of network interfaces with their path characteristics. The simulator replays the Web page download by transferring all Web page objects while respecting their inter-dependencies. It uses the policy to distribute the object transfers across the interfaces and calculates the total page load time.

5.4.1 Simulator Design

Our Web transfer simulator is a discrete event simulator, driven by the Web objects and their dependencies. Since our simulator knows all object inter-dependencies a priori, it can decide when a transfer can be scheduled, i.e., whether all objects that it depends upon have already been loaded. To schedule a transfer, we assign it to a connection. This assignment is the job of the policy module which returns either an existing TCP or MPTCP connection, an interface, or a list of interfaces to use for the new connection. If the limit of parallel connections has been reached, it postpones the transfer. A connection is reused if the hostname matches and it is either idle or it is expected to become idle before a new connection can be established.

The simulator then determines the next event for this connection and updates the global event list. A connection event can be the completion of a transfer or a TCP event. TCP events are triggered by connection handling, TLS handshake, changed

available bandwidth share, and once per RTT during slow-start. To simulate slow-start and fair bandwidth sharing, we keep track of the current throughput for each connection. The throughput is updated according to TCP slow-start and capped by the congestion window or the available bandwidth share of that interface to assure TCP fairness. The available bandwidth share of each interface is potentially adjusted by each connection event for that interface. If needed the time of the next event is then adjusted accordingly. When a transfer finishes, the simulator records the time, marks all transfers that depend on it as enabled, and schedules them. If the last transfer finishes, the total page load time is reported.

The simulator supports persistent connections with and without pipelining for TCP as well as MPTCP connections across multiple interfaces. It uses a default connection timeout of 30 seconds and limits the number of parallel connections per server to 6 and the overall number of connections to 17[4]. We simulate TCP slow-start using a configurable initial congestion window size with a default value of 10 segments [79].

The policies in our simulator can reuse the simulator logic to obtain predictions. They can obtain an estimate of the completion time given the current state of the simulator and an interface/connection option for the transfer to be scheduled. This prediction process is realized by partially cloning the simulator's state, including all currently active transfers, and running the simulation loop till the given transfer completes.

5.4.2 Simulator Implementation

We implemented our data transfer simulator as a heap-based discrete event simulator. It consists of 3k lines of Python code and is available under a relaxed CRAPL license and is published on *GitHub* (`https://github.com/fg-inet/dtsimulator`). It models the process of loading a Web page by keeping track of the status of the transfers, connections, and interfaces.

Each *transfer* corresponds to a Web object which contains the object size, its relationship to other transfers, whether the object is transferred via HTTPS, and the server hostname. The *connections* are responsible for estimating and updating the completion times of the transfers which are assigned to them and for simulating TCP or MPTCP. In the case of MPTCP, we maintain a master connection and per-interface subflows. The *interfaces* bundle the connections and are used to calculate the available bandwidth shares.

The *transfer-manager* keeps track of all transfers and informs the policy if a transfer can be scheduled. The *policy* is the central decision-making entity of the simulation. The policy determines which interface(s) to use or which connection to re-use for each transfer by choosing the most appropriate one. The policy then notifies the transfer-manager to schedule the transfer.

[4]These values correspond to the defaults of the browser we use to retrieve our workload.

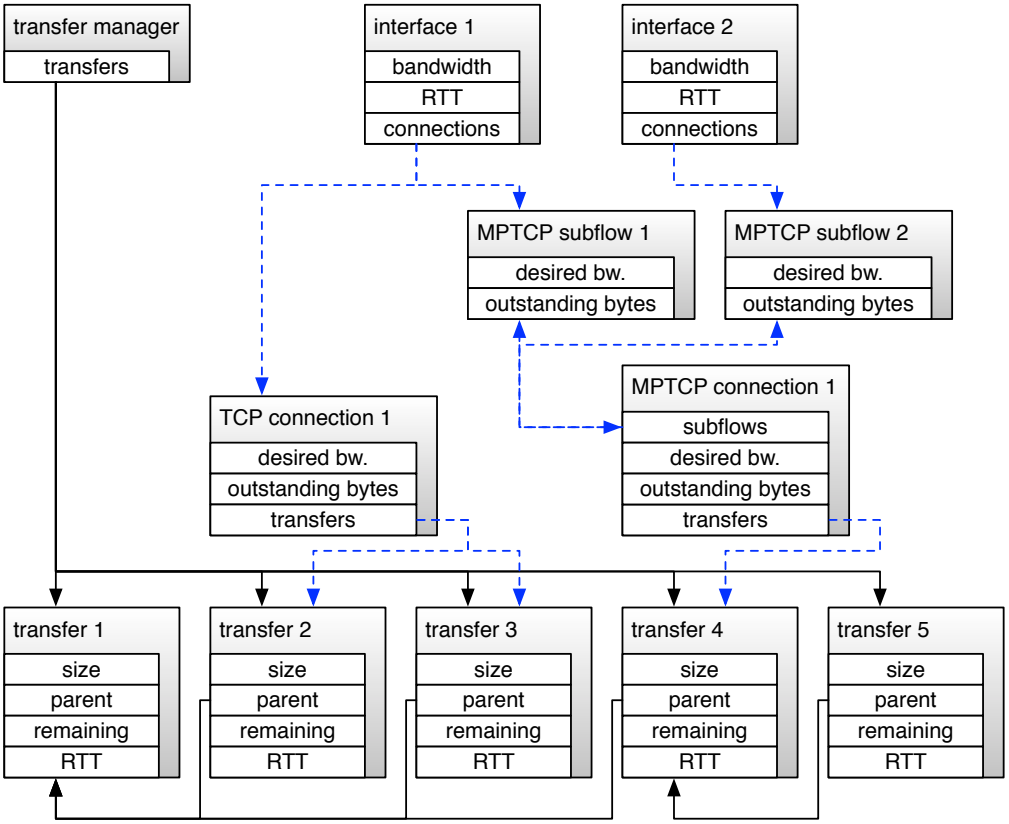


Figure 5.3: Simplified Simulator State Example.

Figure 5.3 shows a simplified example state of a simulator run. The solid black arrows show relationships that are stable during a simulator run, while the blue dashed arrows indicate relationships that change. In this case, *transfer 1* has finished and enabled *transfer 2-4*. The policy (omitted in the figure) has assigned *transfer 2* and *transfer 3* (after *transfer 2* has finished) to *TCP connection 1*, that uses *interface 1* and *transfer 4* to *MPTCP connection 1*. The *transfer 5* is not enabled yet as it depends on *transfer 4*. While the transfers progress, the *outstanding bytes* and *desired bandwidth* fields get updated. The MPTCP connection uses subflow objects to interface with multiple interface objects. The *outstanding bytes* and *desired bandwidth* fields of the subflows mirror the one of the MPTCP connection.

5.5 Web Transfer Simulator Validation

To check the appropriateness of the assumptions and simplifications underlying our simulator we use two different settings for validating the simulator: We compare the expected timings for a set of handcrafted scenarios and the page load times of our web crawls against the simulated timings. To cross-check the consistency of our policy implementations, we also compare the policy implemented in our Multi-Access Prototype, see Section 6.3.5, against the simulated policies.

5.5.1 Handcrafted Scenarios

We choose twelve different scenarios to test the basic functionality of the various simulator policies. The core idea of the scenarios is to test various corner cases including

- Cases those that require connection reuse or cannot benefit from it
- Traffic pattern that can specifically take advantage of either the *MPTCP*, *EAF*, or *EAF_MPTCP* policy
- Cases that stress-test the simulator

For these scenarios, we manually calculate the expected page load times and check the simulator results against them. The simulator passes all of them. Besides, we consequently use assertions and cross checks within the simulator to be able to identify implementation bugs impacting the results.

5.5.2 Simulator vs. Actual Web Load Times

Our validation of the actual Web load times is based on the timings in the HAR files of the crawls. For comparing the page load times with the simulator, we only consider the network timings and ignore other timing information, e.g., rendering, execution time of JavaScript, …

Accordingly, we parse the HAR file, infer the inter-object dependencies, and use these to calculate the cumulative network time of the longest chain of objects fetched in sequence. Next, we compare for all Web pages of our workload the actual page load time to the simulated one. Given that our crawl uses a machine with a single interface we also use a single interface with the policy *Single Interface*. To determine the interface parameters, we estimate the available bandwidth as well as the RTT to the servers from the actual download. To estimate the available bandwidth, we use all objects larger than a minimum size of 50 KB and their download times. Hereby, we take into account that several of these can occur in parallel. Using the median of the estimated bandwidth results in a typically used bandwidth of 67.13 Mbit/s — this suggests that none of the transfers were bandwidth bound. To estimate the RTT the simulator issues a series of pings for each Web page. The median of all measured RTT towards that Web page is then used as an estimator for the interface for the validation run for that Web page.

The simulator, as well as the validation, does several simplifications: The simulator assumes that all Web objects share a single network bottleneck and that the RTT is the same for all servers. In reality, some embedded objects of Web pages are fetched from hosts with different network bottlenecks and RTTs. For the validation, we use ICMP ping rather than TCP ping and the pings are not executed while the HAR files are gathered.

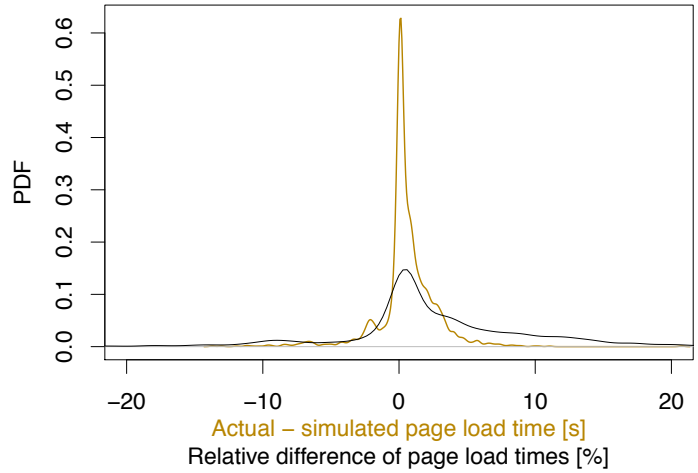


Figure 5.4: Simulator validation: Probability distribution of relative and absolute difference of simulated time vs. actual page load time.

Figure 5.4 shows the absolute as well as the relative differences of the simulated vs. the actual page load times for all Alexa Top 100 Web pages from Section 5.3. The main mass of both distributions is around zero indicating that the simulated page load times are very close to the actual ones. This is confirmed by the median value which is 0.3548/1.5% for the absolute/relative differences. This highlights that the simplifying assumptions of the simulator still enable us to approximate the actual page load times and that we capture most intra-dependencies of the Web page.

There are some differences for some Web pages. We manually checked them and find a majority is caused by differences in the estimated bandwidth, server delays, and name resolution overhead. These are, e.g., related to Web back-office interactions [80]. Overall, the results are rather close and show that our simulations result in reasonable approximations of the actual Web page load time.

5.5.3 Simulator vs. Multi-Access Prototype

As part of our proxy-based evaluation in Section 6.4, we also cross-validate our testbed results with and without our proxy. As expected, the simulator is slightly more optimistic than the testbed results with- and without proxy. However, these differences are consistent across scenarios and small enough to support the simulator results. See Section 6.4.2 for an extensive discussion of the cross-validation.

5.6 Evaluation

To explore the benefits of seamlessly combining multiple paths for speeding up Web page load time, we use our Web transfer simulator. Given the number of possible parameters, we use a full factorial experimental design which allows us to explore in detail the speedups that can be achieved by our policies under different network scenarios and for various Web pages. Each factor can, in principle, influence the page load time. For each factor, we consider multiple values that cover the possible value ranges. By simulating all combinations, see Table 5.1, we run 9M simulations.

5.6.1 Experimental Design

In our experimental design, the primary factor is the **Policy** used with all of our policies, see Section 5.2, as levels. For the *MPTCP* variant that uses a static interface for the initial subflow, we always use *Interface 1*, as it is the lower RTT interface in most scenarios.

The **Web pages** of our workloads, see Section 5.3, are the second factor: Here, the levels are the different Web pages (with their 26 repeated crawls for Alexa Top 100 and the one crawl for the Alexa Top 1000).

The remaining four factors describe the network scenario: **Interface 1 RTT** and **Bandwidth** as well as **Interface 2 RTT** and **Bandwidth**, according to our simplified network scenario introduced in Section 5.1.3. The levels of these were chosen to reflect typical interface characteristics: We consider mobile devices that have WiFi as well as cellular connectivity. *Interface 1* should resemble the possible characteristics of home broadband connectivity (e.g., DSL or cable) and *Interface 2* should resemble the range of possible 3G/LTE coverages. We do not consider costs or restrictions of data plans in our evaluation. The resulting levels are shown in Table 5.1.

Table 5.1: Levels of the Factorial Experimental Design.

Factor	**Levels**
Policy:	*Interface 1*, *Interface 2*, *Round Robin if1*, *MPTCP if1*, *MPTCP rnd*, *EAF*, *EAF_MPTCP*.
Web page:	Alexa Top 100 and Top 1000.
Interface 1 RTT:	10, 20, 30, or 50 ms.
Interface 1 Bandwidth:	0.5, 2, 6, 12, 20, 50 Mbit/s.
Interface 2 RTT:	20, 50, 100, or 200 ms.
Interface 2 Bandwidth:	0.5, 5, 20, or 50 Mbit/s.

5.6.2 Benefits of Combining Multiple Paths

To explore the benefits of combining multiple paths using our policies, we compare the speedups of the page load times against the baseline policy *Interface 1*. The baseline policy *Interface 1* resembles what most current mobile OSes do: Use only WiFi and, therefore, the home broadband if available.

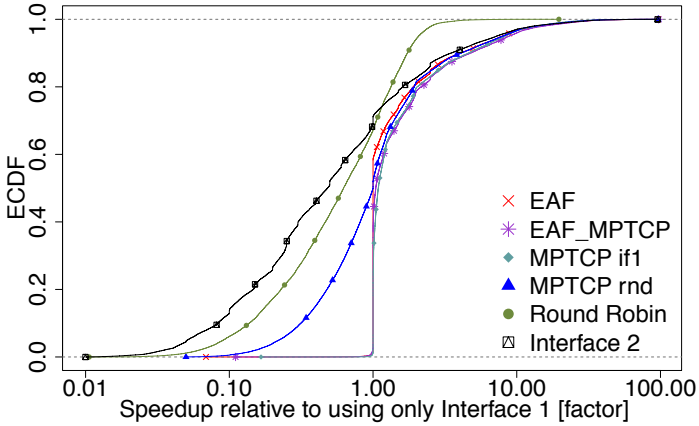


Figure 5.5: ECDF of Speedups vs. *Interface 1* for the Alexa Top 100 workload.

Figure 5.5 shows the Empirical Cumulative Distribution Function (ECDF) of the speedups achieved in our simulation across all network scenarios outlined in Table 5.1, based on the Alexa Top 100 Web pages, and categorized by the policy used. We see that in more than 42% of the cases for *EAF* and 63% of the cases for *EAF_MPTCP* these policies provide a speedup of more than 1. This means that loading a Web page using these policies is faster than using *Interface 1* in the same scenario. In the remaining cases, they almost always provide a speedup of 1. This means that they neither gain nor lose from using multiple interfaces. In these cases, the page load was not bandwidth limited and simply loading the page over Interface 1 was the fastest option. Thus, using the other interface in addition did not provide any speedup. Therefore *EAF* and *EAF_MPTCP* simply choose to use *Interface 1*. We also see that in about 1.5% of cases *EAF* and *EAF_MPTCP* are slower than *Interface 1*, which turned out to be a limitation of the simulator[5]. Overall these results highlight that using *EAF* and *EAF_MPTCP* is a good choice and improve Web page load in almost all bandwidth-bound cases.

The speedups of the *MPTCP* policies are very unlike: When establishing the first subflow over *Interface 1* (*MPTCP if1*), it shows a speedup greater than 1 in 78% of the cases and neither improvement nor penalty in the other cases. In contrast, if starting the first subflow for MPTCP over a randomly chosen interface (*MPTCP rnd*), MPTCP performs worse than *Interface 1* in 48% of the cases and can be up to 10x slower. We take a closer look at these effects in Section 5.6.3.

[5] In these cases, the simulator fetches a single huge object via the less suitable interface while the connection limit prevents starting a new connection on the more suitable one.

The other baseline policies, *Interface 2* and *Round Robin*, unsurprisingly show a slowdown in about 70% of cases as in most network scenarios *Interface 2* has a much higher RTT than *Interface 1*.

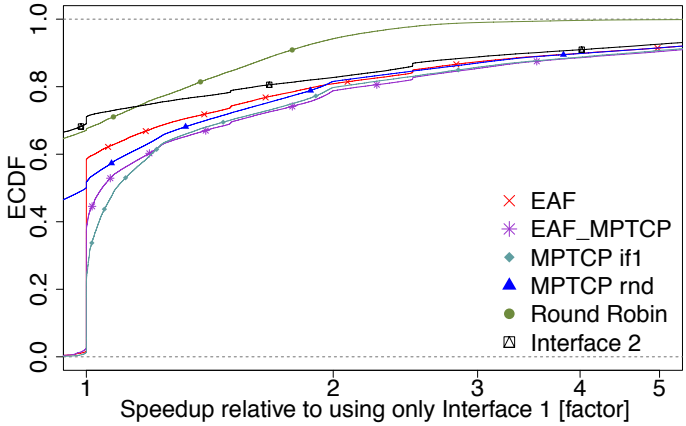


Figure 5.6: ECDF of Speedups between 1 and 5. vs. *Interface 1* for the Alexa Top 100 workload.

Figure 5.6 shows the speedups between 1 and 5 from 5.5 in more detail. From our data, we find that *EAF* was up to 2x faster than *Interface 1* in about 23% of the cases and from 2 to 5x faster in about 11% of the cases. We even see speedups of more than 5x in 8.5% of the cases. *EAF_MPTCP* and *MPTCP if1* shows negligibly higher speedups than *EAF*. Overall, all three policies perform similarly and can take significant advantage of combining multiple paths.

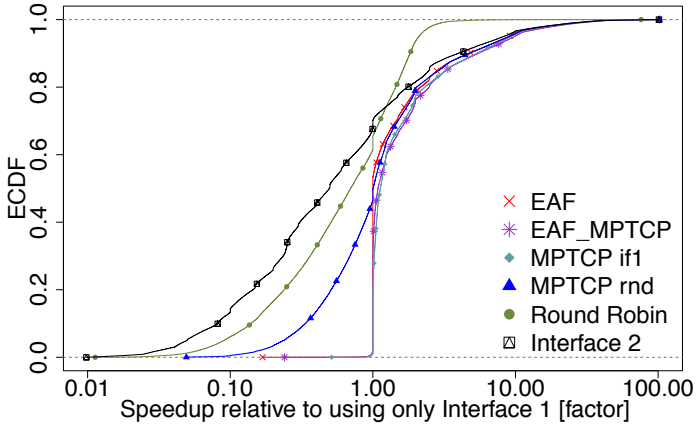
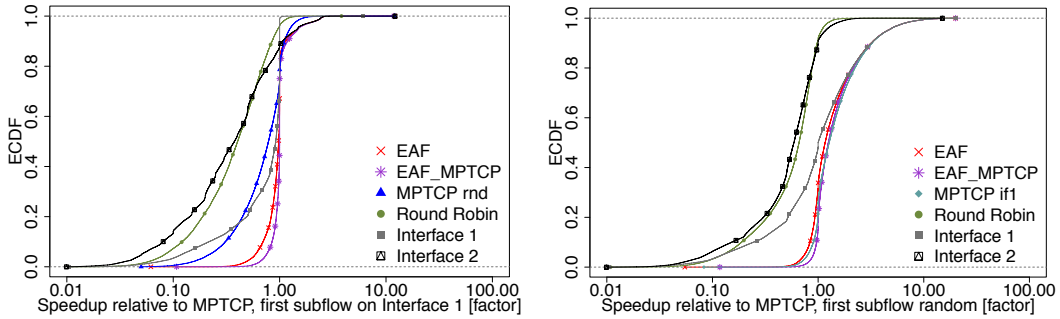


Figure 5.7: ECDF of Speedups vs. *Interface 1* for the Alexa Top 1000 workload.

Finally, Figure 5.7 shows the ECDF of the speedups against *Interface 1* for the Alexa 1000. These look very similar to the ones for the Alexa 100 in Figure 5.5. This allows us to conclude that our benefits are not specific to the Alexa 100.

5.6.3 Benefits of Using the Application-Aware Policies with MPTCP

As described in Section 5.6.2, for our dataset *MPTCP if1* and *MPTCP rnd* behave very differently. While both show gains in the majority of the cases, *MPTCP rnd* is at a disadvantage in 48% of the cases while *MPTCP if1* almost never imposes a penalty.



(a) Speedups vs. *MPTCP if1* (b) Speedups vs. *MPTCP rnd*

Figure 5.8: ECDF of Speedups vs. *MPTCP if1/rnd* for the Alexa Top 100 workload.

In Figure 5.8a, we compare the speedups of our policies for all scenarios and Web pages against *MPTCP if1*. The curves for *EAF* and *EAF_MPTCP* are close to 1, which means that the page load times were similar to MPTCP in most cases and never considerably worse. Figure 5.8a, in contrast, shows that establishing the first subflow for MPTCP over a randomly chosen interface (*MPTCP rnd*) performs worse and can be up to 10x slower than using *Interface 1* and about 30x slower than *MPTCP if1*. This happens because in most network scenarios *Interface 1* has a shorter RTT. As many Web page downloads in our workloads are not bandwidth bound, MPTCP will often perform most of the download over the initial subflow. Thus, not picking the most suitable one in almost 50% of the cases bears a considerable performance penalty.

EAF_MPTCP can always choose the most suitable interface for the first subflow and therefore improves over *MPTCP if1* in cases where *Interface 1* is not the most suitable interface for the first subflow. Note, *EAF* shows similar performance as *MPTCP if1*, but has a higher variability because it cannot adapt, i.e., re-distribute downloads when additional downloads increase the download times predicted by the policy.

There are also cases where *EAF* and *EAF_MPTCP* perform slightly worse than *MPTCP if1*. These cases occur because *EAF* and *EAF_MPTCP* do not take future transfers into account. Both policies cannot change their decision whether to use MPTCP. In contrast, using MPTCP allows rebalancing traffic between subflows if needed. Given the benefits of establishing the first subflow over the lower RTT interface, these corner cases seem negligible to us. A policy with an application-aware heuristic should be able to address this shortcoming.

5.6.4 Explaining Page Load Time Speedups

To understand how the factors of the scenario, i.e., bandwidth, RTT, and Web page, effect the speedups of our $\mathcal{EAF}$ policy, we take a closer look at the cases when $\mathcal{EAF}$ is slower, similar to, or faster than *Interface 1*.

In Figure 5.9 and Figure 5.10, we bin the simulation results of $\mathcal{EAF}$ into six categories of benefits and show how these distribute among the network and page factors of our experimental design. Note that these categories contain different numbers of observation, i.e., $\mathcal{EAF}$ *is slower* accounts for just 1.5% of all cases while *equal to Interface 1* account for 56.6% of all cases — see Table 5.2 for the exact percentage of Observations within each bin of speedups.

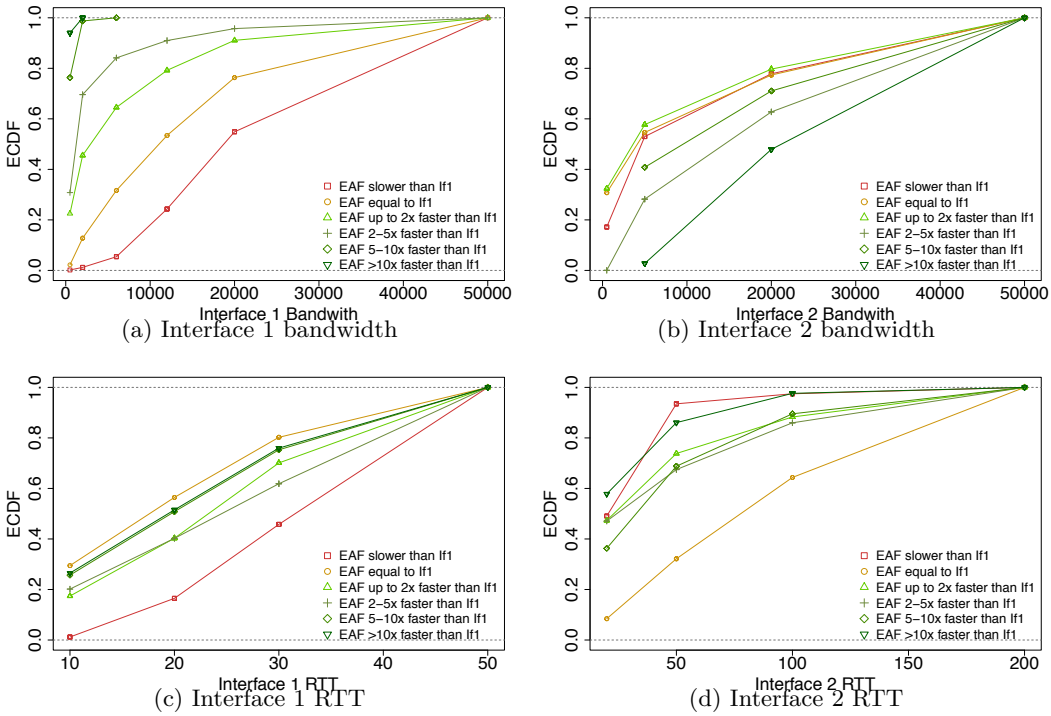
Table 5.2: Observations within the Levels of Speedup

Level of speedup	**Observations**
$\mathcal{EAF}$ slower than *If1*	1.46 %
$\mathcal{EAF}$ equal to *If1*	56.68 %
$\mathcal{EAF}$ up to 2x faster than *If1*	22.48 %
$\mathcal{EAF}$ 2-5x faster than *If1*	10.60 %
$\mathcal{EAF}$ 5-10x faster than *If1*	5.22 %
$\mathcal{EAF}$ >10x faster than *If1*	3.31 %

The ECDFs in Figure 5.9 shows the frequency of the speedup categories over the different levels of our network factors. Figure 5.9a shows the frequency of the speedup categories over the different levels of Interface 1 bandwidths from Table 5.1. In cases when $\mathcal{EAF}$ is slower or equal to *Interface 1*, higher values for the *Interface 1* bandwidth are more prevalent, while high speedups mostly occur when the *Interface 1* bandwidth is low. The ECDF in Figure 5.9c shows that $\mathcal{EAF}$ can achieve gains more often in cases with a moderate or high *Interface 1* RTT. Opposing, as expected, we tend to see high speedups for higher levels of *Interface 2* bandwidth, see Figure 5.9b, and for lower levels of *Interface 2* RTT, see Figure 5.9d.

To explore what kind of Web pages can benefit from our $\mathcal{EAF}$ policy, we plot the ECDFs of the speedup categories for different aspects of the Web pages in Figure 5.10. High speedups occur much more frequently for large Web pages, see Figure 5.10a. We conclude that unsurprisingly these take more advantage of using multiple paths. For the number of objects of a Web page, see Figure 5.10c, and the number of hosts involved in its delivery, see Figure 5.10d, we see similar results: Higher speedups occur more frequently in cases where the Web page consists of many objects, or where its download involves many hosts. For the median object size in Figure 5.10b, we do not see such clear results.

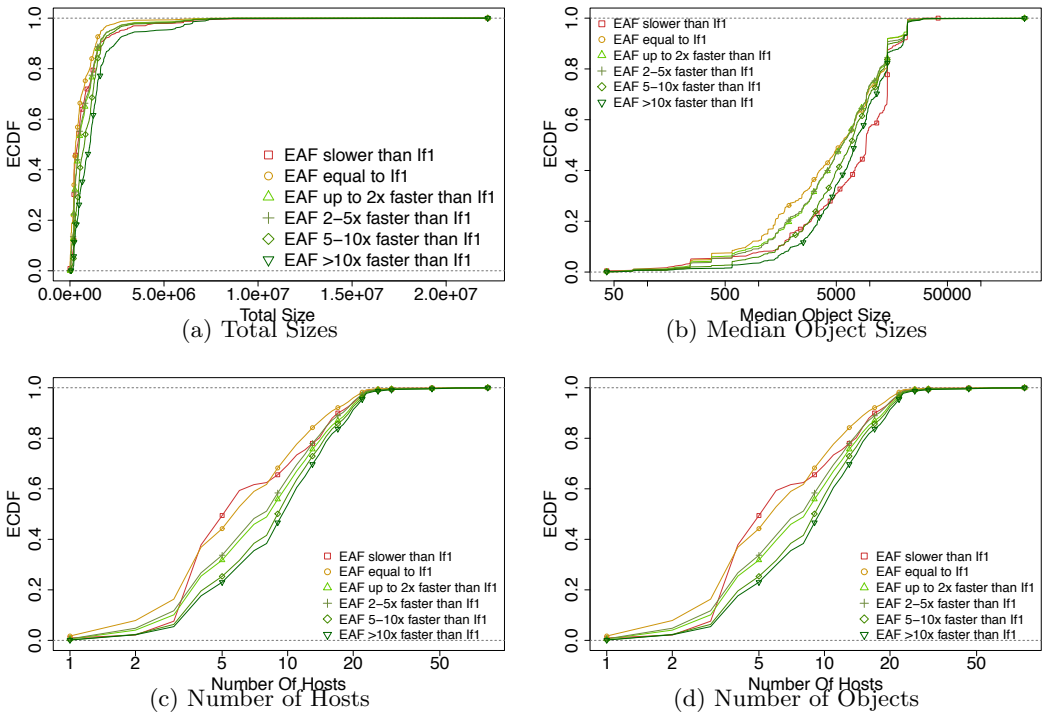
Both analyses show that the $\mathcal{EAF}$ poly is most useful when Web page download is bandwidth limited. Together with the results from Section 5.6.3, we can also conclude that in cases where the RTTs of the paths differ, $\mathcal{EAF}$ can assure that latency critical communications are mapped to the most appropriate path.



(a) Interface 1 bandwidth (b) Interface 2 bandwidth

(c) Interface 1 RTT (d) Interface 2 RTT

Figure 5.9: Level of speedup of the $\mathcal{EAF}$ policy achieved for Alexa Top 100: Network Scenario Factors



(a) Total Sizes (b) Median Object Sizes

(c) Number of Hosts (d) Number of Objects

Figure 5.10: Level of speedup of the $\mathcal{EAF}$ policy achieved for Alexa Top 100: Web Page Properties

5.7 Conclusion

In this chapter, we evaluate the potential performance benefits of combining two paths for Web browsing using a custom *Web Transfer Simulator*. Our simulation study uses a full factorial experimental design covering different policies, the Alexa Top 100 and Top 1000 Web sites and a wide range of network characteristics as factors The levels modeling the network characteristics are chosen to reflect typical interface characteristics for a mobile device that has WiFi as well as cellular connectivity. The policies include application-agnostic strategies — just using a single interface, simple round-robin distribution of HTTP requests and using MPTCP— as well as one application strategy: our $\mathcal{EAF}$ policy, optionally in combination with MPTCP.

We see that the $\mathcal{EAF}$ policy provides a speedup in more than 42% of the cases without using MPTCP and in 63% of the cases when using MPTCP. In about 20% of the cases, the $\mathcal{EAF}$ policy provides a speedup of two or more. In the remainder of cases, the page load is not bandwidth bound within our scenario and only benefits from choosing the lowest latency path. We also compare our $\mathcal{EAF}$ policy to vanilla MPTCP and find that, for our use cases, distributing HTTP requests on can achieve the same performance benefits as MPTCP in most cases. By analyzing the factors that show the highest performance gains, we conclude that overcoming bandwidth limitations and choosing the lowest-latency path for a latency-sensitive website can indeed improve web performance. Therefore, application awareness provided by Socket Intents, especially the *size to be received* Intent used in this study, enables choosing the right optimization criteria and reduces variability caused by optimizing for the wrong objective.

While these results are promising, we acknowledge that the realism of this study is limited: The assumption of knowing the size of the majority of the Web objects is mostly realizable, e.g., by providing them as annotations or by caching them, but requires significant efforts. The exclusion of effects caused by packet-level effects, browser optimizations, and cross-traffic are the major limitations chosen in order to quantify the benefits of application awareness for choosing multiple paths.

6

Multi-Access Prototype for BSD Sockets

As stated in Chapter 1, transport diversity is usually not exploited due to the lack of OS support. In this chapter, we present a Multi-Access Prototype that adds transport option selection to the BSD Socket API. We chose to extend the BSD Socket API, as it is the template for the current networking APIs of nearly all OSes today. In contrast to the generic transport option selection discussion in Chapter 1 to 4, our Multi-Access Prototype focuses on two dimensions of transport option selection: path selection and endpoint selection.

We start the discussion of our Multi-Access Prototype by first revisiting the limitations of using multiple access networks with vanilla BSD Sockets in Section 6.1 and explain why the BSD Socket API is not well suited to support automated protocol stack composition. Therefore, we are mostly excluding the problem of protocol stack composition in this chapter. Afterwards, we derive design criteria for a BSD Socket API OS-based support for multiple access network selection (Section 6.2).

We present our Multi-Access Prototype in Section 6.3 — it is a wrapper for the BSD Socket API that communicates with a central Multiple Access Manager that makes the actual decisions. The whole implementation consists of about 15k lines of C code. It is available on Github (`https://github.com/fg-inet/socket-intents/`) under BSD License.

In Section 6.4, we present an application using our Multi-Access Prototype— an HTTP proxy. Using this proxy, we show that even a basic policy that takes advantage of multiple paths can achieve significant performance benefits.

Finally, in Section 6.5, we summarize the lessons learned and point out why the BSD Socket API is not particularly well suited to integrate transport option selection. Stating the limitations of the BSD Socket API, we conclude that one should rethink the programming interface for network communication and provide some outlook how a new abstract *Transport Services API [10–12] (TAPS API)* overcomes these limitations.

6.1 Lecacy of the Socket API

More than thirty years ago, the BSD Socket API was designed as an IPC extension to the filesystem API [81]. The protocol domain `PF_INET` was added to support IPC using the Internet protocol family. Later on, this was complemented with the protocol domain `AF_INET6` for the current version of the Internet Protocol. While being the default programming interface for communication on the Internet, the BSD Socket API did not undergo substantial changes since then [81], except for changes to name resolution[1].

Applications that want to connect to a server usually have to resolve the server's hostname using `getaddrinfo()`. Then they create a socket file descriptor using `socket()` passing address family, socket type, and protocol obtained from `getaddrinfo()`. Finally, the application calls `connect()` to establish the communication using the address obtained from `getaddrinfo()`. In the `connect()` call the address obtained from `getaddrinfo()` is passed back to the OS.

While this process looks quite natural, the design and implementation details of these calls and the structures used have a strong influence on the applicability of automatic transport option selection on top of the BSD Socket API. In the following sections, we discuss some of the problem areas of the vanilla BSD Socket API.

6.1.1 File Descriptor vs. Transport Protocol Semantics

When using BSD Sockets, file descriptors are the abstraction for network communication. Within this abstraction, the transport protocols available get mapped to the IPC or *socket types* they fit best: TCP is mapped to `SOCK_STREAM`, UDP to `SOCK_DGRAM`. In the case of SCTP, one can choose between `SOCK_STREAM` and `SOCK_SEQPACKET`. Depending on this choice, the usage and semantics of the socket file descriptors change as follows.

- Sockets using `SOCK_STREAM` resemble Unixpipes. They represent byte streams, implicitly guarantee reliable, in-order delivery and do not preserve message boundaries. The operations that can be used on these sockets are `read()` and `write()` as used on regular files.
- Sockets using `SOCK_DGRAM` represent association sets or associations, depending on whether `bind()` was called on the socket. The operations used on these sockets are `sendmsg()` and `recvmsg()`.
- Sockets using `SOCK_SEQPACKET` behave like `SOCK_DGRAM`, but implicitly guarantee reliable, in-order delivery. With special `sctp_sendmsg()` and `sctp_recvmsg()`, message stream semantics are emulated.

[1] In the original BSD Socket API, name resolution was done by using `gethostbyname()`, which by design could only support one protocol domain. It was replaced by `getaddrinfo()` to support returning address family and, thus, allows applications to use dual-stack IPv4/IPv6

Given the above protocol to socket type mapping, these communication units do not match the communication units derived from protocols in Section 2.4 except in case of TCP/`SOCK_STREAM`. Therefore, file descriptors are an inappropriate abstraction for automated protocol stack composition as applications have to adapt to different semantics that depend on the transport protocol chosen. For each protocol, they have to implement different semantics and adapt the communication units accordingly.

Communication units of actual applications, e.g., an HTTP request for HTTP-based applications — the dominant protocol on the Internet [82, 83] — are typically not aligned with the communication units provided by the BSD Socket API. In the HTTP case, the application has to choose for each request to either open a new TCP connection or reuse an existing one — they operate at message granularity. The operation they have to perform on a TCP socket are at stream granularity: Opening a new stream allows choosing among multiple interfaces using `bind()`. Reusing an existing one saves 2 RTTs for the TCP handshake, a few 100 KB for the TLS handshake (if applicable), and time spent in TCP slow-start. The adaption between these quite different semantic is left to the application.

In conclusion, the overall system is not a unified transport API, but is merely an artifact of squeezing networking into the Unixphilosophy of *Everything is a file*. For further discussion of other file descriptor weirdness see Section 6.5.2 for issues regarding name resolution and end of Section 6.5.3 for issues regarding asynchronous I/O.

6.1.2 Multi-Homing and Multiple Access Networks

The availability of multiple paths in today's Internet usually implies having multiple interfaces at the host or multiple addresses from different IP prefixes on the same interface. Back in the time the Internet and the BSD Socket API was designed, hosts having multiple interfaces (multi-homed hosts) and hosts with multiple addresses were considered a corner case [84]. Thus, Vanilla BSD Sockets do not offer reasonable support for those "corner cases": Applications that want to use multiple paths usually have to apply their own heuristics to select an address and interface. To place traffic on a specific interface, applications have to use the following hack: The `bind()` socket call allows applications to override the source address of an outgoing communication. Otherwise, the OS uses the IP address of the paths via which it routes to the given destination as the source address. Once the source address of the communication is selected, a system is configured with an appropriate routing policy[2], which will use the outgoing path chosen with the source address.

In practice, the application logic becomes even more difficult as obtaining the necessary information often requires special privileges and the respective API differs heavily by OS flavor. Therefore, vanilla BSD Sockets do not assist the application in distributing traffic among multiple interfaces.

[2] It should route traffic with a specific source address over the interface associated with that source address.

6.1.3 Name Resolution

Since in the original IPC context name resolution was not needed, name resolution is not integrated with the BSD Socket API, but is provided via a support library. The calls `getaddrinfo()` or its predecessor `gethostbyname()` are not directly linked to a socket file descriptor as they are typically called before a socket is created.

Instead, a call to `getaddrinfo()` returns a linked list of `sockaddr` structs, where each entry contains an `ai_family` (socket domain / IP version), the pair of `ai_socktype` and `ai_protocol` (transport protocol), and a `sockaddr` struct containing an address and port to connect to. Using this list, an application can implement automatic endpoint selection itself. As there is no portable non-blocking variant of the `getaddrinfo()` call, implementing endpoint probing mechanisms like Happy Eyeballs as part of the application logic is rather challenging when relaying on the vanilla BSD Socket API.

In addition, as described in Section 2.7.1, name resolution has to be performed on a per-path basis. Neither the `getaddrinfo()` call nor the `addrinfo` struct has means to realize the per-path separation of name resolution results as needed. Therefore, Applications that want to do per-path name resolution cannot rely on the name resolution library shipped with the BSD Socket API at all. They must use other means for path aware name resolution, usually provided by an OS specific proprietary library or an external one shipped with the application.

6.2 Design Criteria for Multi-Access Prototype

Given the complexity of transport option selection laid out in Chapter 2 and the limitation of the BSD Socket API, we next design a system which **adds OS support for path selection and endpoint selection**. Hereby, we define the term OS to explicitly includes the kernel, the standard libraries including the socket API, as well as the system services including the daemons of a base install. Our system should **enable applications to jointly optimize their network performance** by distributing their communication **across all available paths**. As a simplification, we assume to have exactly one path per interface. As different applications have different requirements, the system should **allow them to specify their "intent" for a given communication unit** as a hint to the system which aspects to optimize. Since there is not always a universally "best" interface, the system should **choose an appropriate interface for each communication unit**.

As we want to evaluate rather basic policies, we use **exchangeable policy modules**, i.e., small pieces of code that decide which access network to use in a given situation based on the available information, instead of our generic policy presented in Section 4.4. For our prototype, we want the policy modules to focus on a few parameters to quantify their particular impact. Thus, a policy needs **access to interface parameters and statistics**. To enable joint optimization across applications, we need to have a single component that has **knowledge about the intents** of *all* applications that use the Multi-Access Prototype. Moreover, the policy needs to **respect the optimization of external communication partners**, i.e., it should only use DNS results for communication on the interface they were requested through.

In cases like, e.g., HTTP, where the communication units of the application do not align with the ones provided by the vanilla BSD Socket API, our system needs to **allow applications to specify *Intents* for each of the applications' communication units** or even finer granularities. If an application's communication unit, e.g., an HTTP request, is not aligned with the underlying transport, e.g., a TCP stream, the former has to be assigned to the latter. For example, an HTTP-based application typically tries to assign multiple HTTP requests to the same TCP stream to reduce overhead. This connection reuse logic has to be implemented by each application individually, although it is, typically, not application-specific. Therefore, besides choosing an appropriate interface for a given communication unit, the system must also **decide whether to reuse an existing socket/connection or to set up a new one** for that communication unit and **communicate this decision** to the application. The latter is essential for supporting TLS and other protocols that need to explicitly set up a per-connection context for each new connection. Finally, the application needs to **specify if and how a socket can be reused**.

To enable easy deployment and portability, our system should **be compatible with BSD Sockets** and **require minimal changes to applications**. Moreover, it should be possible to **support other transport protocols besides TCP**.

6.3 Implementation

The Multi-Access Prototype consists of two components, see Figure 6.1: An augmented BSD Socket API, the Multi-Access Socket API (white), which is realized as a shared library, and the Multi-Access Manager (MAM) (gray), that runs as a service available to all applications on the client. This service runs in userspace and, thus, does not require any modifications to the OS kernel.

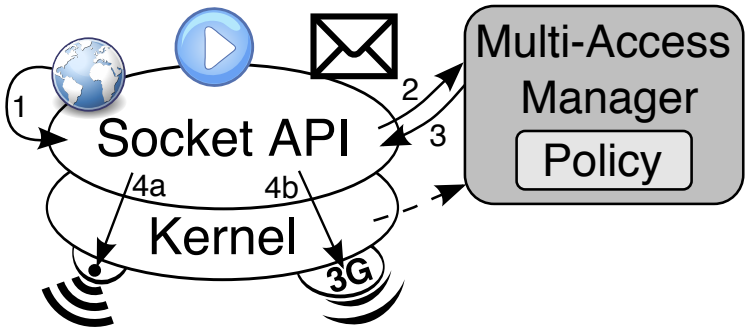


Figure 6.1: Interactions between Network Stack and Multi-Access Manager.

When an application using our Multi-Access Socket API starts a new communication, our augmented Socket API involves the Multi-Access Manager (MAM) as shown in Figure 6.1: First an application specifies its Intents through the API (1), then our augmented socket library queries the policy within the MAM via Unix-domain sockets (2). Within the MAM, a policy module decides which interface(s) to use and which endpoints to prefer. The MAM communicates this decision back (3), and, finally, the socket library applies the decision by selecting an interface and providing an re-ordered endpoint list (4a/b).

The remainder of this section is structured according to the individual components of our Multi-Access Prototype: We first describe the *Multi-Access Socket API variants* in Section 6.3.1. The three different variants explore different approaches how to integrate path and endpoint selection into the BSD Socket API. Next, we shift our focus to the MAM and its components: We describe the overall architecture of the MAM in Section 6.3.5. We explain how we gather per-interface characteristics (Section 6.3.3) and how we orchestrate Multi-Path TCP [14–16] (MPTCP) from within the MAM (Section 6.3.4). Finally, in Section 6.3.5, we describe how to implement policy modules for our Multi-Access Prototype.

6.3.1 Augmented Socket API

Our Multi-Access Socket API is implemented as a wrapper library around the BSD Socket API. Instead of directly calling the functions of the vanilla BSD Socket API, our wrapper first sends a requests to the MAM containing all information about the communication, e.g., hostname to connect to, Socket Intents set. Our wrapper library then waits for the results from the MAM and uses the vanilla BSD Socket API to carry out the actions advised by the MAM, e.g., to bind to a specific source address and set additional socket options.

There exist different variants of the Multi-Access Socket API. Each of these variants fits different use cases and explores different approaches how to integrate path and endpoint selection into the BSD Socket API:

- The *classic* variant, see Section 6.3.1.1, sticks as close as possible to the call sequence of BSD Sockets, but adds an additional context parameter to all socket calls. It is meant as a baseline to explore which aspects of automated transport option selection can be integrated into the vanilla BSD Socket API without changing the application logic.
- The *augmented name resolution* variant performs automated transport option selection as part of the name resolution, see Section 6.3.1.2. This variant tries to simplify the implementation without diverging too much from the vanilla BSD Socket API. It minimizes the changes to the BSD Socket API, but adds additional overhead to the application.
- The *message granularity* variant, see Section 6.3.1.3, adds support for access selection at message granularity, e.g., to enable connection caching for HTTP. It moves the whole connection setup into a single API call replacing the usual call sequence of BSD Sockets.

For all variants, name resolution is offloaded to the MAM and handled by the policy module, as the regular `getaddrinfo()` does not allow per-path name resolution.

6.3.1.1 Classic API Variant

Our system augments the individual calls of the BSD Sockets to take advantage of the MAM, but leaves the general call sequence unchanged. The calls and parameters are shown in Table 6.1. First, it adds the parameter (`muacc_context`) to link all calls related to one communication unit and stores policy-related information, e.g., DNS replies and possible source addresses. Our `muacc_context` is needed to link `getaddrinfo()` with all other calls[3]. Second, we add support for Socket Intents by adding an `INTENT` socket option level for use with `setsockopt()`.

[3] All other calls could have also been linked using shadow-structures identified by the file descriptors — please also see Section 6.5.4 for the limitations of this approach.

Table 6.1: Classic API Variant: Socket API with Socket Intents.

Call	parameters (excerpt)	in/out
`muacc_getaddrinfo`	`muacc_context_t *ctx`	in
	`const char *hostname`	in
	`const char *servname`	in
	`const struct addrinfo *hints`	in
	`struct addrinfo **res`	out
`muacc_socket`	`muacc_context_t *ctx`	in
	`int domain, int type`	in
	`int protocol`	in
`muacc_setsockopt`	`muacc_context_t *ctx`	in
	`int socket`	in
	`int level`	in
	`int option_name`	in
	`const void *option_value`	in
	`socklen_t option_len`	in
`muacc_connect`	`muacc_context_t *ctx`	in
	`int socket`	in
	`const struct sockaddr *address`	in
	`socklen_t address_len`	in
`muacc_close`	`muacc_context_t *ctx`	in
	`int socket`	in

The typical use of our modified BSD Socket API involves the following steps:

1. Resolve the hostname by calling `muacc_getaddrinfo()`.
2. Creating a new socket file descriptor using `muacc_socket()`.
3. Set Socket Intents by calling `muacc_setsockopt()` using the `INTENT` socket option level and the respective Socket Intents.
4. Call `muacc_connect()`.

See Listing 6.1 for a minimal example.

In contrast to vanilla BSD Sockets, the client does not need to pass the address it obtained from `muacc_getaddrinfo()` since this is kept in the `muacc_context`. Unless the application explicitly chooses a source address (and, therefore, a source interface) our augmented BSD Socket implementation consults the MAM for choosing a suitable source and destination address. `muacc_connect()` automatically binds the socket to the source address and connects the socket to the destination address.

Internally, our implementation uses vanilla BSD Socket calls to realize the communication for the application and execute the choices of the MAM. Therefore, it is possible to extend our Multi-Access Prototype to all transport protocols that are available via vanilla BSD Sockets. An unpublished UDP support for our Multi-Access Prototype from the University of Aberdeen exists. The example program in Listing 6.1 demonstrates that automated path- and endpoint selection are feasible with this API, but features such as Happy Eyeballs are not easy to accommodate.

Listing 6.1: Classic API Variant: Usage Example.

```
// Create and initialize a context to retain information across function
// calls
muacc_context_t ctx;
muacc_init_context(&ctx);

int socket = -1;

struct addrinfo *result = NULL;

// Set Socket Intents for this connection. Note that the "socket" is
// still invalid, but it does not yet need to exist at this time. The
// Socket Intents prototype just sets the Intent within the
// muacc_context data structure.

enum intent_category category = INTENT_BULKTRANSFER;
muacc_setsockopt(&ctx, socket, SOL_INTENTS,
    INTENT_CATEGORY, &category, sizeof(enum intent_category));

int filesize = LENGTH_OF_DATA;
muacc_setsockopt(&ctx, socket, SOL_INTENTS,
    INTENT_FILESIZE, &filesize, sizeof(int));


// Resolve a host name. This involves a request to the MAM, which can
// automatically choose a suitable local interface or other parameters
// for the DNS request and set other parameters, such as preferred
// address family or transport protocol.
muacc_getaddrinfo(&ctx, "example.org", NULL, NULL, &result);

// Create the socket with the address family, type, and protocol
// obtained by getaddrinfo.
socket = muacc_socket(&ctx, result->ai_family, result->ai_socktype,
    result->ai_protocol);

// Connect the socket to the remote endpoint as determined by
// getaddrinfo.  This involves another request to MAM, which may at this
// point, e.g., choose to bind the socket to a local IP address before
// connecting it.
muacc_connect(&ctx, socket, result->ai_addr, result->ai_addrlen);

// Perform some communication ...

// Close the socket. This de-initializes any data that was stored within
// the muacc_context.
muacc_close(&ctx, socket);
```

6.3.1.2 Augmented Name Resolution API Variant

This API variant moves the path selection and endpoint selection process into a modified variant of `getaddrinfo()`, but leaves the general call sequence used by the vanilla BSD Socket API in place. Compared with the Classic API (see Section 6.3.1.1), this requires more changes to the application but also enables the application to implement mechanisms like Happy Eyeballs.

Table 6.2 presents the only socket API call we changed for this API variant. For all other socket API calls, we use the versions from the vanilla BSD Socket API. Socket Intents, alongside with other socket options, are passed directly to our modified `getaddrinfo()` as part of the `hints` parameter. To do so, we extended the `addrinfo` struct, see Listing 6.2, to include a list of socket options and the source address for the outgoing connection. We also provide a new `socketopt` struct to pass a list of socket options as part of our extended `addrinfo` struct. The name resolution

Table 6.2: Augmented Name Resolution API Variant: Modified Socket API Calls.

Call	parameters (excerpt)	in/out
`muacc_ai_getaddrinfo`	`const char *hostname`	in
	`const char *servname`	in
	`const struct muacc_addrinfo *hints`	in
	`struct muacc_addrinfo **res`	out

Listing 6.2: Augmented Name Resolution API Variant: Modified `addrinfo` Struct.

```
/** Extended version of the standard library's struct addrinfo
 *  used as hint and as result parameter for muacc_ai_getaddrinfo
 */
struct muacc_addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;

    /** Not included in struct addrinfo. Purpose:
      * 1. Provide Socket Inetnts to the MAM / policy
      * 2. Allow MAM to return  recommended socket options
      */
    struct socketopt *ai_sockopts;

    int ai_addrlen;
    struct sockaddr *ai_addr;
    char *ai_canonname;

    /** Not included in struct addrinfo.
      * Contains the source address / path an application should bind to.
      */
    int ai_bindaddrlen;
    struct sockaddr *ai_bindaddr;

    struct muacc_addrinfo *ai_next;
};
```

implementation of `getaddrinfo()` is done by the MAM, which makes all decisions and returns them in the `result` parameter as a list of endpoints ordered by policy preference. Each endpoint is annotated with the source address the application should bind to and socket options that should be set on the socket. Applications use this information as parameters to the vanilla BSD Socket API calls or other APIs. We provide helpers to set all socket options from the `result` data structure on a given socket.

In Listing 6.3, we provide a minimal example of how the augmented name resolution API variant is used. It begins with the construction of the linked socket option list containing the Socket Intents and the construction of the `hints` for `getaddrinfo()`. For the remainder, we use the regular `socket()`, `bind()`, and `connect()` calls of the vanilla BSD Socket API. This interface imposes additional complexity on the application, but it is more flexible then the Classic API. Still, it only supports communication units that match those of the vanilla BSD Socket API.

Listing 6.3: Augmented Name Resolution API Variant: Usage example.

```
// Define Intents to be set
enum intent_category category = INTENT_BULKTRANSFER;
int filesize = LENGTH_OF_DATA;

struct socketopt intents = { .level = SOL_INTENTS,
    .optname = INTENT_CATEGORY, .optval = &category, .next = NULL};
struct socketopt filesize_intent = { .level = SOL_INTENTS,
    .optname = INTENT_FILESIZE, .optval = &filesize, .next = NULL};

intents.next = &filesize_intent;

// Perform name resolution
struct muacc_addrinfo intent_hints = { .ai_flags = 0,
    .ai_family = AF_INET, .ai_socktype = SOCK_STREAM, .ai_protocol = 0,
    .ai_sockopts = &intents, .ai_addr = NULL, .ai_addrlen = 0,
    .ai_bindaddr = NULL, .ai_bindaddrlen = 0, .ai_next = NULL };
struct muacc_addrinfo *result = NULL;
muacc_ai_getaddrinfo("example.org", NULL, &intent_hints,
    &result);

// Create the socket, bind it to the path provided by
// muacc_ai_getaddrinfo and connect it
int fd;
fd = socket(result->ai_family, result->ai_socktype,
    result->ai_protocol);
bind(fd, result->ai_bindaddr, result->ai_bindaddrlen);
connect(fd, result->ai_addr, result->ai_addrlen);

// perform some communication ...

// clean up
close(fd);
muacc_ai_freeaddrinfo(result);
```

6.3.1.3 Message-Granularity API Variant

To support communication units at message granularity, e.g., multiple HTTP requests over a single TCP stream, we move the logic for choosing which request to send via which socket to the Multi-Access Prototype. Since there is no such functionality in the vanilla BSD Socket API, we add three new calls (see Table 6.3): `socketconnect()` to get a new socket or reuse an existing one, `socketrelease()` to mark a socket as available for reuse, and `socketclose()` to close a socket and prevent its reuse. This API replaces most of the vanilla BSD Socket API's socket calls with two new calls and eliminates the need for glue code between `getaddrinfo()`, `socket()`, `setsockopt`, and `connect()`—code that is otherwise often duplicated.

Table 6.3: Message-Granularity API Variant: Added Socket API Calls.

Call	parameters (excerpt)	in/out
`socketconnect`	`int *socket`	in,out
	`const char *host`	in
	`size_t hostlen`	in
	`const char *serv`	in
	`size_t servlen`	in
	`struct socketopt *sockopts`	in,out
	`int domain`	in
	`int type`	in
	`int proto`	in
`socketrelease`	`int socket`	in
`socketclose`	`int socket`	in

This API variant moves functionality needed by many applications into the Multi-Access Socket API. It is designed to support applications that use BSD Sockets in a straightforward fashion. Bundling this control flow comes at a cost: It makes the porting of applications that already implement optimizations such as name resolution and connection caching much harder. For example, porting the highly optimized Firefox browser becomes near-to-infeasible. For applications that apply their own network optimization, it is advisable to either use the augmented name resolution API variant (Section 6.3.1.2) or directly interface with the MAM.

As our focus is on supporting simple request/response type protocols, e.g., HTTP/1.1, we presume sequential reuse of connections by the same application. We do not support multiple concurrent requests on the same TCP connection as in HTTP/2, since this requires support for protocol-specific message splitting within the API. Also, we do not implement Happy Eyeballs within our `socketconnect()` implementation, which will be required when moving from a prototype to a standard library.

The usage of this API variant is demonstrated in Listing 6.4: When an application wants to send a request, it uses `socketconnect()` to ask our Multi-Access Prototype for a socket for a specific host, service, and socket options (including Socket Intents) tuple. The in/out parameter for the socket file descriptor allows to explicitly request a new socket or reuse one of a certain set of sockets. The return value informs the

Listing 6.4: Message-Granularity API Variant: Usage Example.

```
// Define Intents to be set later
enum intent_category category = INTENT_BULKTRANSFER;
int filesize = LENGTH_OF_DATA;

struct socketopt intents = { .level = SOL_INTENTS,
    .optname = INTENT_CATEGORY, .optval = &category, .next = NULL};
struct socketopt filesize_intent = { .level = SOL_INTENTS,
    .optname = INTENT_FILESIZE, .optval = &filesize, .next = NULL};

intents.next = &filesize_intent;

// Initialize a buffer of data to send later.
char buf[LENGTH_OF_DATA];
memset(&buf, 0, LENGTH_OF_DATA);

int socket = -1;

// Get a socket that is connected to the given host and service,
// with the given Intents
socketconnect(&socket, "example.org", 11, "80", 2, &intents, AF_INET,
    SOCK_STREAM, 0);

// Send data to the remote host over the socket.
write(socket, &buf, LENGTH_OF_DATA);

// Close the socket and tear down the data structure kept for it
// in the library
socketclose(socket);
```

application whether the socket is a new one or an existing one. This allows the application to decide if it needs to add any per-connection actions, e.g., if a new TLS handshake has to be started for a new connection. Once the application is done, it can either release or close the file descriptor using the second new call. The former enables reuse; the latter does not.

The Multi-Access Socket API implementation manages a set of active sockets per destination host/service pair. The implementation of `socketconnect()` checks whether there are currently unused open sockets to the same host and port. It then checks with the MAM if any of these existing sockets satisfy the needs of the request according to the Socket Intents. If not, the MAM chooses an interface for a new socket to be created. The call then either returns the chosen existing socket or the newly opened one. The `socketrelease()` call marks a socket as unused.

This design implicitly prevents connection reuse across processes and, thus, addresses the most critical security concern of connection reuse. However, connection reuse within a single process, e.g., a Web browser, can still have security and privacy implications, e.g., a disclosure of parallel browsing sessions through a timing channel. Applications that implement multiple protection domains already have to mitigate these kind of implications in many other contexts. As authentication in HTTP based applications is usually done on a per-request basis, the design has no authentication and authorization implications on these.

The actual implementation has a few additional implications for applications using it: It makes failures on write, which are only poorly handled in many applications, more likely, e.g., if the remote side has closed a connection that is scheduled for reuse in the meantime. We mitigate this problem by testing the connection before scheduling it for reuse.

Our design requires DNS caching to be an integral part of the policy module. For this, the policy module has to guarantee that DNS replies are kept separate on a per-interface basis and therefore should only be cached and used for communication on the same interface from which they were acquired. This separation is necessary to avoid interference with DNS-based server selection and load balancing as laid out in Section 2.7.1 and [3]. While this design is optimal for destination selection, this clashes with the architecture of today's Web browsers that tightly integrate DNS caching with their connection management because of its considerable performance impact.

In conclusion, this API variant allows access selection on message granularity and is much more comfortable to use than the vanilla BSD Socket API and the other two variants, but it massively changes the call sequence of the socket API. It addresses many issues of the BSD Socket API with regards to path selection and destination selections, but do not address the principal problems with protocol stack composition.

6.3.2 The Multiple Access Manager (MAM)

The MAM is the central place for deciding which path, and, thus, which source and destination address pair to use. It is shared by all applications of a host that use our augmented socket interface. As shown in Figure 6.2, the MAM consists of three components: (1) The actual policy module that implements the access network selection strategy (see Section 6.3.5), (2) a set of data collectors that polls various network statistics, see Section 6.3.3, and (3) the MAM Master (mamma) itself, which handles the communication towards the Multi-Access Socket API and provides the infrastructure for the policy module and the data collectors.

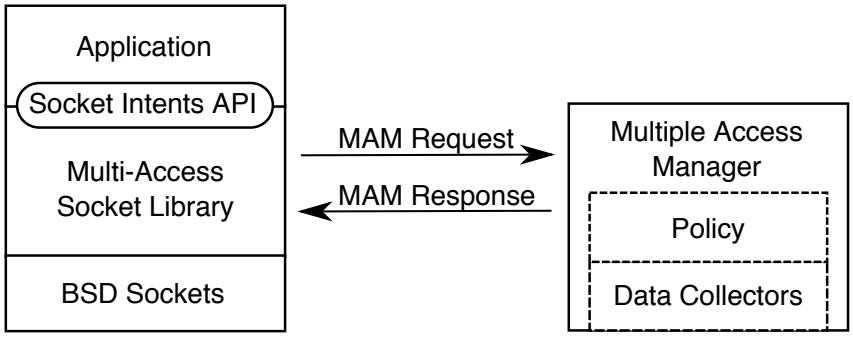


Figure 6.2: Architecture of the MAM.

The MAM is implemented using *libevent*. It is designed in a way that does not need to keep any per-request state, as the requests of the Multi-Access Socket API carry all application originated information needed for a simple policy. This avoids complicated inter-process state management and avoids memory leaks in case an application using the Multi-Access Socket API crashes. If the policy module needs to keep state, it can use the context pointer provided by *libevent* to pass information between callbacks or a per-prefix dictionary provided by the MAM.

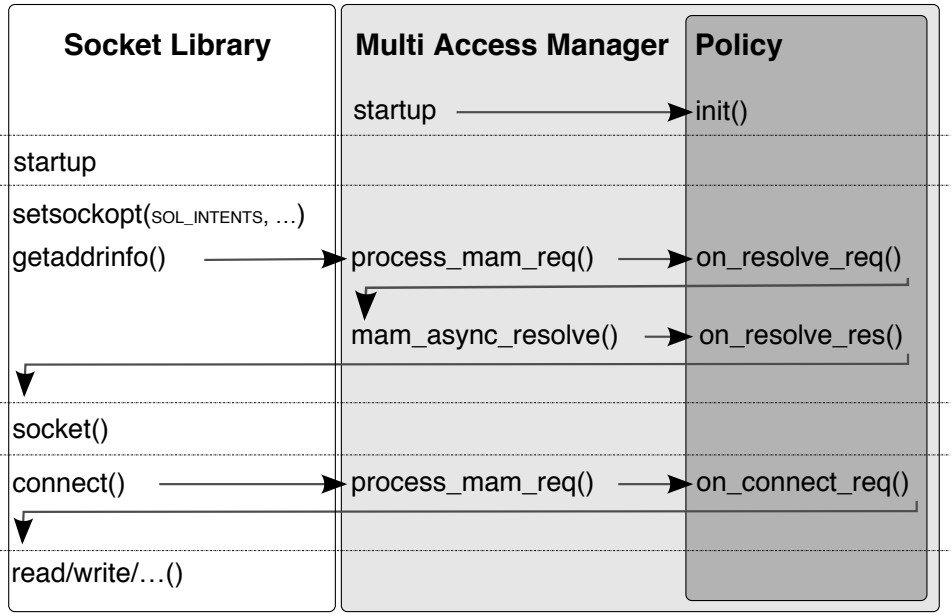


Figure 6.3: Interactions between Multi-Access Prototype components.

The control flow across the components of our Multi-Access Prototype is shown in Figure 6.3: After startup, the MAM creates a list of all local interfaces and their network prefixes, loads the policy, and initializes it. To be able to make decisions, the policy can use various network statistics from the MAM, see Section 6.3.3, which is stored on a per-prefix basis within the MAM. If an application requests name resolution or connection setup from any variant of the Multi-Access Socket API, the implementation of the Multi-Access Socket API collects all information available, including Socket Intents, reusable sockets, and the parameters issued with the call. This information is serialized using a simple TLV protocol and sent to the MAM using a *Unixdomain socket*. The MAM de-serializes the information and calls the respective callback of the policy module. It is the responsibility of the policy module to compile an answer and instruct the MAM to pass the information back to the Multi-Access Socket API. This can happen asynchronously, e.g., in case of name resolution, and is supported by a set of helper functions provided by *libevent* and the MAM.

6.3.3 Path Characteristics Data Collectors

The MAM periodically queries the OS for statistics about the current usage and properties of the available local interfaces and stores them in per-prefix data structures. The data collectors are implemented as a component of the MAM using callbacks that are executed periodically with a configurable update interval. Our empirical observations suggest that an interval of 100 ms works well. When this callback is invoked, the MAM gathers a variety of information already available within the OS. Our current implementation gathers the values listed below. More data about the current network performance can easily be gathered by adding code to the MAM or the policy.

- Minimum Smoothed Round Trip Time (SRTT) of current TCP connections using a network prefix, as an estimate for last-mile latency.
- Transmitted and received bytes per second over an interface within the last callback period, as an estimate for current utilization.
- Smoothed transmitted and received bytes per second over an interface, as an estimate for recent utilization.
- Maximum transmitted and received bytes per second over an interface within the last 5 minutes, as an estimate for maximum available bandwidth.
- On 802.11 interfaces, the Received Signal Strength Indicator (RSSI) of the last received frame on that interface, as an estimate for reception strength.
- On 802.11 interfaces, the modulation rate of the last received and the last transmitted unicast data frame on that interface, as an estimate for the available data transmission rate on the first hop.

When a policy callback is invoked, the policy can use the most recent measurements to guide its decisions. Note that we do not perform active measurements from within the MAM to avoid overhead. For later versions, we plan to keep a short history of some of the values to allow more elaborate policies.

6.3.4 Orchestrating Multipath TCP

Multi-Path TCP [14–16] (MPTCP) allows splitting TCP streams across multiple paths. For large transfers, this can achieve almost the combined bandwidth of both paths. Using MPTCP allows to chunk messages into smaller segments. Therefore, it allows finer-grained bandwidth sharing than request scheduling on a per message granularity. Thus, its functionality complements the functionality of our Multi-Access Prototype.

Using the information given by Socket Intents, we can **enable the policy module to control the usage of MPTCP**. This includes determining the set of paths used for a particular connection/association to match the properties of a

given transfer and avoids opening MPTCP subflows on already crowded interfaces or interfaces with a high RTT. We can also combine MPTCP with our message-granularity API from Section 6.3.1.3 and **make MPTCP available via Socket Intents**. Therefore, we can choose appropriate paths for both: small communication units—which the policy can distribute at message granularity—as well as large ones—which MPTCP can chunk and distribute. This also addresses head-of-line blocking MPTCP can introduce for small objects.

To enable the policy module to control the usage of MPTCP we added an additional path manager to the Linux MPTCP implementation. Our user-space MAM uses Netlink [85] sockets to communicate with the kernel-space MPTCP path-manager. If a policy decides to use MPTCP it selects the source address for the initial subflow. If MPTCP is feasible, the path-manager notifies the MAM. The MAM can then decide whether to open additional subflows and over which path(s) and therefore instructs the MPTCP path-manager via the Netlink socket. The actual distribution of the segments belonging to the MPTCP connection is left to the MPTCP segment scheduler.

6.3.5 Policy Implementation

Policy modules for the MAM are implemented as shared libraries. These modules implement the callbacks presented in Table 6.4 for the different API variants. For the message-granularity API from Section 6.3.1.3, there are different callbacks depending on whether the Multi-Access Socket API can reuse a socket (`on_socketchoose_request()`) or not (`on_socketconnect_request()`). In any case, the result is sent back using the functions `_muacc_send_ctx_event()` provided by the MAM, which takes the updated `request_context` and a return code as arguments.

Table 6.4: Callbacks implemented by a Typical Policy Module

Type	Callback	Parameters	in/out
`int`	`init`	`mam_context_t *mctx`	in,out
`int`	`on_resolve_request`	`request_context_t *rctx`	in,out
		`struct event_base *base`	in,out
`int`	`on_connect_request`	`request_context_t *rctx`	in,out
		`struct event_base *base`	in,out
`int`	`on_socketconnect_request`	`request_context_t *rctx`	in,out
		`struct event_base *base`	in,out
`int`	`on_socketchoose_request`	`request_context_t *rctx`	in,out
		`struct event_base *base`	in,out
`int`	`cleanup`	`mam_context_t *mctx`	in,out

Using this framework, we implemented a set of policies we describe in the remainder of this sections. These policies do not keep track of concurrent transfers and rely on measured network utilization and RTT. They allow to roughly predict when a file transfer finishes. The policies implemented in the Multi-Access Prototype are the following:

Single Interface This policy always chooses a particular, statically configured interface.

Round Robin This policy uses multiple interfaces on a round robin basis.

Earliest Arrival First (EAF) This policy uses the *size to be received* Intent to predict the completion time for each available interface. It then chooses the one where the communication unit will arrive first. The prediction is based on an estimation of the interface RTT and available bandwidth. Since the most recent measurement one or more downloads have been scheduled on an interface, we reduce the available bandwidth in our calculation. Finally, we divide the file size by the estimated available bandwidth to approximate the download duration. We add one RTT if a connection can be reused and two RTTs if a new connection has to be established. Finally, the interface with the shortest predicted arrival time is chosen. We do not consider TLS handshakes.

In the next section, we use an HTTP proxy to demonstrate the benefits of the simplified $\mathcal{EAF}$ policy.

6.4 A Web Proxy with Socket Intents

To explore the benefits of path selection in practice, we implemented an HTTP proxy that takes advantage of our Multi-Access Prototype. Our HTTP proxy consists of 2.300 lines of C code. The code specific to our Multi-Access Prototype covers only 20 lines of code.

The proxy uses the *size to be received* Intent, see Section 3.4, in conjunction with the EAF policy, see Section 6.3.5. *EAF* computes for every object an estimate of the load time for both paths. This estimate is based on the *size to be received* Intent, the Interface RTT, bandwidth, and its usage, i.e., the currently observed traffic. It then schedules the object over the path with the shortest predicted download time, reusing connections if possible.

Since the proxy does not know the size of the objects in advance, we use a two-step download process: First, the proxy issues a range request for the first m bytes to get the initial part as well as the size of the object. Then, if the object has not already been transferred completely, the remainder is retrieved via a second range request. The proxy can handle various answers including the full object or the remaining part of the object, with and without chunked-encoding.

The choice of the size of the initial request m enables a trade-off between RTT and network bandwidth. We see good results for values between 4-8K; values that fit within the initial TCP window of today's Web servers.

6.4.1 Testbed Setup

To study the benefit of Socket Intents we set up a testbed, see Figure 6.4. It consists of three physical machines: A Web server, a traffic shaper, and a client. The client has two network paths to the Web server via two separate network interfaces. The network characteristics are emulated by the traffic shaper and include three scenarios which range from fully symmetric to asymmetric, see Table 6.5.

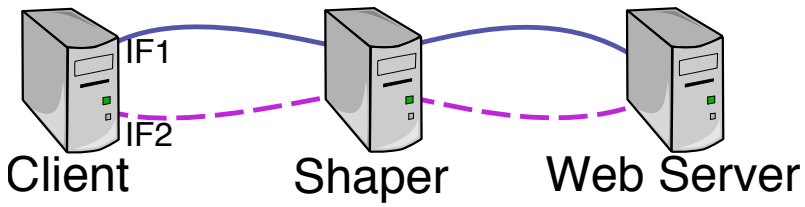


Figure 6.4: Testbed setup used in the emulation.

On the Web client, we run a Web browser along with the proxy and the MAM. Our MAM supports the policies *Single Interface*, *Round Robin* and *EAF* as discussed in Section 6.3.5. The first two policies serve as baselines. We automate a Web browser and restart it for each measurement to ensure that the cache is cold. As the Web browser, we use Mozilla Firefox 38.8 with the Selenium browser automation

Table 6.5: Testbed shaper: Network parameters.

	Interface 1			Interface 2		
	RTT ms	Down MBit/s	Up MBit/s	RTT ms	Down MBit/s	Up MBit/s
Symmetric	45	10.0	1.0	45	10.0	1.0
Asymmetric	20	6.0	0.768	70	13.0	6.0
Highly Asym.	10	3.0	0.768	100	20.0	5.0

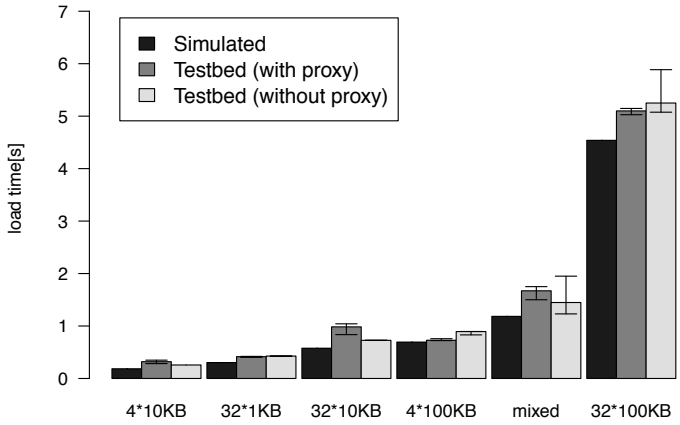
framework, Firebug 2.0.17, and NetExport 0.9b7. As we only want to look at the influence of the access network, we use a single Web server. However, we set up a virtual host for each hostname to restrict connection reuse appropriately. As we want a lower bound of the performance benefits, we set the TCP parameters of the Web server to conservative values: We use TCP/Reno with an initial congestion window size of 10 MSS. We disable TCP metrics saving to prevent congestion window caching as well as TCP segmentation offloading to eliminate interference with the NIC firmware and to avoid interference between our measurements. We also choose to ignore DNS overhead, therefore, we run a DNS server serving all emulated hostnames on the client to ensure that name resolution does not add delays.

The Web server hosts our workload. It consists of handcrafted pages, each with a different number of objects (ranging from 2 to 128) of various sizes (between 1 KB and 1 MB), as well as mirrored versions of several Web pages from the Web workload we use in Chapter 5, see Section 5.3. We craft our workload to only use HTTP - this dramatically simplifies our testbed setup and avoids interference with online certificate checking.

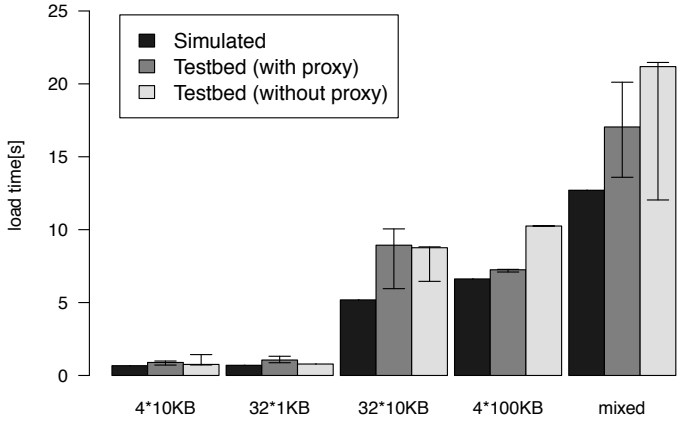
6.4.2 Cross-Validation of Proxy and Simulator

In order to relate the testbed study with the simulator study in Chapter 5 and estimate the overhead incurred by our proxy, we cross-validate both studies and compare them against the Web page load time of our workload in the testbed without proxy.

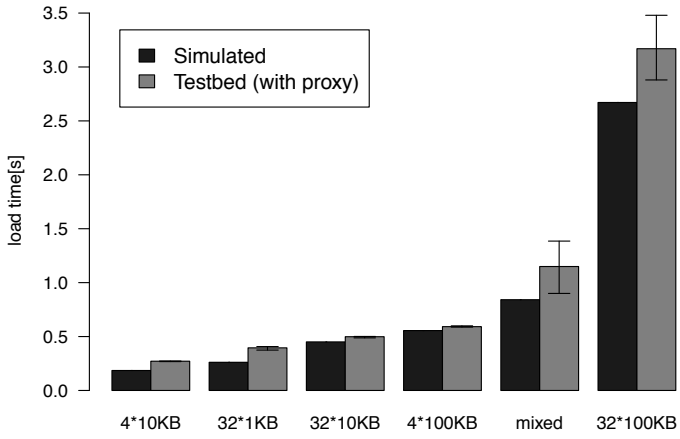
In Figure 6.5 we compare the simulated and the actual load times for the handcrafted workloads of different sizes, showing the median load time and 95% confidence intervals. The mixed workload consists of 32 objects of 1KB, 16 objects of 10 KB, 2 objects of 100 KB and 2 objects of 200 KB. Using a single interface with an RTT of 50 ms and a bandwidth of 6 Mbit/s, see Figure 6.5a, we see slightly higher load times on the testbed both with and without the proxy, especially for large workloads. Using a single interface with only 0.5 Mbit/s, see Figure 6.5b, we do not get a page load time for the workload with 32 objects of 100 KB because the browser times out after 10-20 seconds, so we do not show it in this plot. Using our $\mathcal{EAF}$ policy with symmetric shaping (50 ms and 6 Mbit/s on one interface, 50ms and 5 Mbit/s on the other), we cannot test the case without proxy, as we cannot use $\mathcal{EAF}$ without the proxy. Both our simulator and the proxy in the testbed show



(a) Single Interface (50ms, 6Mbit/s)



(b) Single Interface (10ms, 0.5Mbit/s)



(c) EAF with symmetric shaping

Figure 6.5: Comparison of simulated load time and actual load time in the testbed with different synthetic workloads.

speedups, see Figure 6.5c. Note the differences between the y-axes, which reflect the speedups observed in Section 6.4.3. We get similar results for RTTs up to 200 ms and bandwidths up to 50 Mbit/s.

With regards to the testbed study, we get similar load times with and without the proxy. This underlines that the two-step download in our proxy does not have a considerable influence on the load time. As small objects are already downloaded in the first step, they are not effected by the two-step split. Only larger objects are split, but are usually bandwidth bound and therefore do not suffer much from the additional RTT incurred by the two-step download.

Overall the simulator is more optimistic than the testbed. However, the differences are quite small. The following observations can explain the differences between testbed and simulator: First, the gzip transfer encoding conflicts with range requests: Sometimes the server sends the whole object even though only the initial part is requested. Moreover, disabling compression for the initial request is not feasible as it eliminates compression also for the second request since the content-range refers to the range after compression. Second, the simulator presumes that all independent transfers start immediately, which is not always the case in practice. This can skew timings, in particular for small workloads. These effects are independent of the use of our Prototype. Therefore, we can use the simulator to conduct a relative comparison between scenarios with and without Socket Intents with the limitations discussed in Section 5.1.

6.4.3 Socket Intent Benefits in the Testbed

Based on the testbed setup from 6.4.1, we evaluate the benefits of the *EAF* policy by comparing it against the *single interface* policy – i.e., *Interface 1*, *Interface 2*. See Section 6.3.5 for a description of the policies. For each of them, we download selected web pages and synthetic workloads 7 times. We compute the load time of the individual objects and aggregate the times during which objects were downloaded to compute the total page load time[4]. The resulting page load times are shown in Figure 6.6 using a logarithmic y-axis. It includes all three policies: *Interface 1*, *Interface 2*, and *EAF*. The mixed handcrafted workload shown here consists of 16 objects of 1KB, 8 objects of 10KB, and 4 objects of 100KB. The selected Web pages were chosen from a set of web pages we could adapt for the testbed. We were not able to adapt a representative set of Web pages for the reasons laid out in Section 5.1.2.

We see that *Interface 1* is the better choice if the Web objects are small and the network scenario is asymmetric. *Interface 2* is the better choice if the objects are larger or if there are more objects. Using both interfaces is, in particular, beneficial for the symmetric scenario. While there is still a benefit of using both interfaces it gets smaller for more asymmetric scenarios.

[4]The total display time includes page rendering and client-side JavaScript computation, which we exclude here.

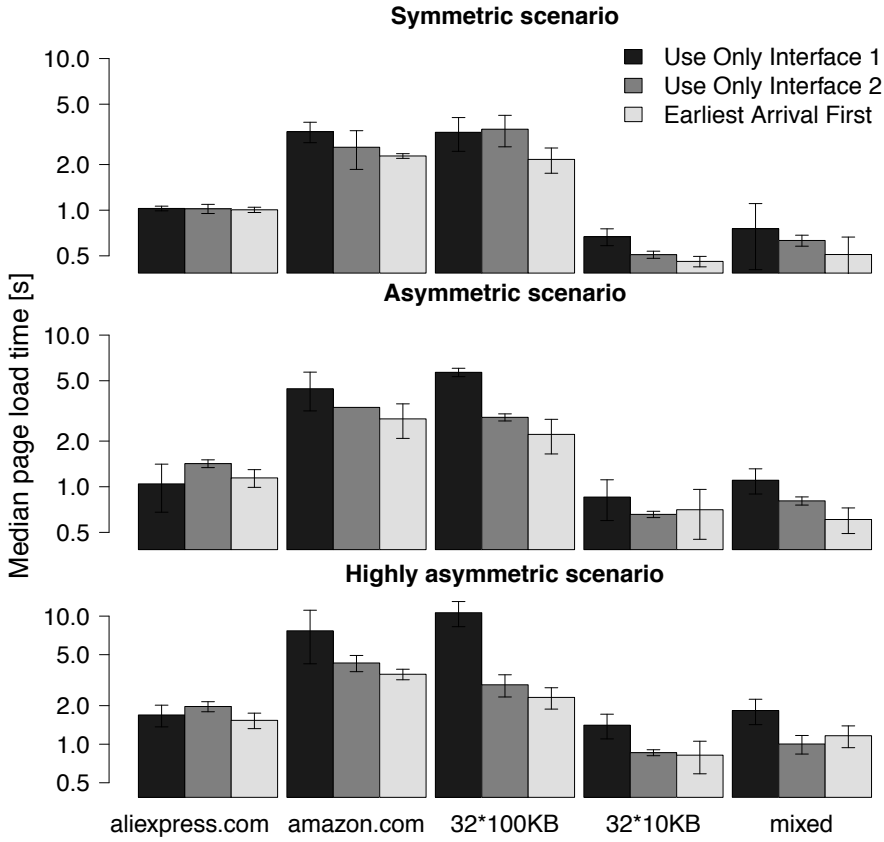


Figure 6.6: Proxy: Page load times.

The $\mathcal{EAF}$ policy takes advantage of the multiple access networks seamlessly. It either uses both interfaces or the better one of the two with only a slight increase in page load time variability. For the handcrafted workload of 32 objects of 100 KB, our $\mathcal{EAF}$ policy outperforms the better of the two interfaces with speedups from 25% to 50%. For some of the actual Web pages that we mirror on our testbed, including `amazon.com`, we get speedups in the range of 20–45%. For other Web pages such as `aliexpress.com`, we only get a speedup of up to 10%.

The reason for the "decreased" benefits compared to the handcrafted pages are that the mirrored Web pages fetch content from different servers, which limits connection reuse. Furthermore, even for mirrored versions of the same Web page, load times vary based on optimizations in the contained JavaScripts, as the Alexa 100 pages are heavily optimized.

Nevertheless, our results highlight the potential of informed transport option selection: Even with a proxy, the page load times improve. Including transport option selection within the browser rather than a proxy is likely to yield even better performance.

6.5 Lessons Learned

While designing and implementing the different parts of the Multi-Access Prototype, as described in this thesis, we faced several challenges. Some of them turned out to be actual limitations of the BSD Socket API, on which we focus in the following sections.

6.5.1 Platform Dependent APIs

In the MAM, we discover the currently available paths as well as performance statistics about these paths. As we intended to run our Multi-Access Prototype on Linux and MacOS X, we had to deal with many platform-specific details of the Socket API. Enumerating network interfaces and their addresses were one of the few things we could implement in a portable way.

The most challenging portability issue for the Multi-Access Prototype is the gathering of performance data and network statistics. There is no standardized or common API to get data like packet counters or TCP timings. On Linux, most statistics can be accessed using the *Netlink* [85] interface. The exact calls and details are barely documented, but it works for most statistics. Getting access to similar statistics on MacOS X turned out to be even more challenging. This problem finally convinced us to drop full MacOS X support. We only support policies that rely on statistics on Linux.

Another portability issues we had to solve is the inconsistent definition of the `sockaddr_in` and `sockaddr_in6` structs on both platforms. It is the result of a historical disagreement whether a socket address should have a member stating its length. Having such a member makes opaque handling of socket addresses without knowing their type possible, omitting it saves a byte of memory. Today, this historical disagreement still manifests in increased code complexity and the use of `#ifdefs`.

6.5.2 The Missing Link to Name Resolution

For many aspects of transport option selection, e.g., name resolution, it is crucial to have information, such as Socket Intents or other socket options, available as early as possible. The primary problem for integration path selection and endpoint selection with the BSD Socket API is the order of the function calls that are involved in name resolution, destination selection, protocol, and path selection, and how they are linked.

In the vanilla BSD Socket API, most functions take a socket file descriptor as an argument. Thus it is possible to link different function calls to the same communication. However, `getaddrinfo()` is not linked to a socket file descriptor but is needed for destination selection. At this point, it is not yet possible to set a socket option including Socket Intents, because the socket does not exist yet.

Our three API variants described in Section 6.3.1 work around this problem in different ways:

- The *augmented name resolution* variant in Section 6.3.1.2 places the whole automation of transport option selection into the `getaddrinfo()` function. The results are returned in an extended `addrinfo` struct and have to be applied manually by the application, including binding to a source address representing the selected path and applying all socket options provided in a list, for each connection attempt.
- The *classic* variant in Section 6.3.1.1 adds a context to all socket- and name resolution-related API calls and delays the actual name resolution until `mucaa_connect()` is called.
- The *message granularity* variant in Section 6.3.1.3 puts all functionality into one call.

All of these approaches add the missing link between name resolution and the other parts of the API but add a lot of state keeping — either to the application or to the Socket Intents library.

6.5.3 Asynchronous I/O

Network I/O is asynchronous. Yet, the original filesystem abstractions were synchronous, i.e., blocking. As asynchronous I/O was only added later on to the Unixfilesystem API, it is hard to use. There are at least four different asynchronous I/O APIs variants, namely `select()`, `poll()`, `epoll()`, and `kqueue()`, whereby most OS implements at least two of them.

To implement asynchronous I/O in our Socket Intents prototype, we wrapped one of the asynchronous I/O APIs that is available on most platforms: `select()`. To make Socket Intents accessible to more applications and on more platforms, a production-grade system needs to wrap all asynchronous I/O APIs and implement most of the socket creation logic, path selection and connection logic within these wrappers. Mixing asynchronous I/O with the different multithreading approaches is challenging and error-prone and may lead to unintuitive behavior, e.g., calling our prototype's select() from different threads could lead to anything from deadlocks to busy waiting.

Another issue is that we use Unixdomain sockets to communicate between our Multiple Access Manager and the Socket Intents API library called by the application. So we need to make sure that the application does not block while communicating with the Multiple Access Manager.

Also, the problems arising from using file descriptors get worse when realizing asynchronous I/O: If a Socket API call should return immediately, it needs to provide the application with a reference to a flow that has not yet been fully set up, i.e., a reference to a socket *future*. An implementation of such an asynchronous API has

to return an unconnected socket file descriptor, on which the application then calls, e.g., `select()`, and starts using it once it becomes readable and writable. If the destination, path, and transport protocol have not been chosen yet at this point, the file descriptor returned by the implementation might not yet have the final address family and transport protocol. When the implementation later creates the final socket of the right type, it can re-bind it to the file-id of the originally returned file descriptor using `dup2()`. This procedure can easily lead to time-of-check / time-of-use confusion. To make things even worse, the application can copy the file descriptor future using `dup()`, which is rarely useful for sockets, but in combination with file descriptors used as future, it leads to unexpected behavior. Some of these issues could be eliminated if the transport option selection is moved into the kernel and implemented within the file descriptor backend itself, but the resulting API would still render some un-intuitive corner cases.

6.5.4 Here Be Dragons hiding in Shadow Structures

The API variants described in Section 6.3.1.3 and Section 6.3.1.1 need to keep a lot of state in shadow structures as this information cannot be passed between the Socket API calls otherwise. This state needs to be cleaned up when the last copy of the file descriptor is closed or the last socket held for reuse has timed out. Besides, access to these shadow structures has to be thread-safe.

Implementing both API variants has turned out to be extremely error-prone and has led of unspecified behavior in the system library and requires platform-dependent extensions. These results indicate that an implementation of transport option selection that nicely integrates with BSD Sockets may come with lots of limitations and may also not be portable across POSIX-compliant operating systems.

6.5.5 Changing Applications to Use Better APIs is Hard

Changing APIs used by existing applications is hard. This is especially true for the BSD Socket API, as it provides a bunch of API calls that gradually change the state of a file descriptor. As this logic is complex and sometimes needs to be used in different parts of the program, it is often wrapped in custom helpers. In applications implementing connection reuse, this results in quite complex structures and control flows which are heavily interwound with the BSD Socket API.

A student, exploring how an MPEG DASH [62] player, the *GPAC multimedia player*, could be augmented to make use of our Message-Granularity API Variant [86], encountered difficulties in how to transfer part of the application's socket handling to our Multi-Access Prototype. Our API required more state handling, and by changing the socket on a per transfer basis, the new code violated some hidden assumptions in the network code of *GPAC*, which needed more code changes than anticipated to integrate our Message-Granularity API. As *GPAC* uses no parallelization, the performance benefits where reasonable small.

6.6 Conclusion and Outlook

With our Multi-Access Prototype, we show that it is feasible to do automated path selection and destination selection based on the information provided by Socket Intents. However, the resulting system also unveils the limitations of the BSD Socket API. These limitations include the loose integration of name resolution in the overall API and the philosophy of *Everything is a file* that only fits reliable byte stream transports well. For all other transport semantics, the BSD Socket API exposes many corner cases and semantic inconsistencies to the application. Overall, this renders the BSD Socket API unfit for protocol stack composition, but still enables path selection and destination selection. However, even with only doing path selection, we can already achieve significant performance benefits.

Our API variants allow overcoming the loose name resolution integration. Still, they are workarounds that change the semantics of the BSD Socket API. After building this prototype, we conclude that the vanilla BSD Socket API is not well suited to exploit transport diversity in an easy and portable way and should be considered broken. Therefore, we believe it is time to rethink the programming interface for network communication and to replace the BSD Socket API with an event-based API that supports different communication unit granularities and transport semantics.

The *Transport Services API [10–12] (TAPS API)* we are designing within IETF Taps Working group solves most of the problems we encountered when adding transport option selection to the BSD Socket API. By being strictly event-based, asynchronous I/O can be realized cleanly without undesired side effects. The TAPS API is designed to integrate Happy Eyeballs as well as protocol and path selection based on the preferences and requirements provided by the application. It uses messages as base communication unit granularity and therefore can implement message de-multiplexing for protocols like HTTP/2 gracefully. To support stream-based communication protocols like TCP, the TAPS API supports providing application protocol-specific *framers* to split a byte stream into individual messages. Unlike our Multi-Access Prototype, it does not support automated connection reuse using implicit connection pools – still, this feature can be gracefully implemented as an extension.

7 Conclusion

Transport option selection within an Internet end host is challenging. This thesis highlights three aspects — characterizing *transport diversity*, how to enable *transport option selection within the operating system*, and the *performance benefits* of exploiting the transport diversity by, e.g., using multiple paths. In this chapter, we summarize the findings of this thesis, give an outlook on future research, and draw an overall conclusion.

7.1 Summary

The initial part of the thesis characterizes the transport diversity provided by today's Internet. In Chapter 2, we provide a comprehensive analysis of the three dimensions of *transport diversity* — *path selection*, *endpoint selection*, and *protocol stack composition*. Each dimension contributes to the set of transport options an application can, in principle, choose from. By focusing on semantic *communication units* rather than Protocol Data Units (PDUs), we analyze on which communication granularity optimizations can be applied. Next, we identify a set of building blocks that are used to compose transport services provided by Internet protocols — our *transport mechanisms*. By combining the perspectives of communication units and transport mechanisms, we can reason about protocol stack compositions and the tradeoffs of choosing among them. Based on this characterization, we use *communication units* and *mechanisms* to analyze a representative set of Internet protocols. Our analysis highlights three aspects: a) which protocol combinations can be used to *compose a protocol stack* for a given communication need, b) which functionality is provided by that protocol stack composition, and c) at which granularity of communication the protocols can operate and be tuned for the applications' needs. We find a diverse set of protocols that provide transport options at almost all granularities. Despite that, most applications on today's Internet use TCP as a result of designing protocols into the most easily usable ecosystem. From a communication unit perspective, this often means forcing message granularity communication into a stream abstraction and, thus, preventing further optimizations. This underlines the need for operating system (OS) based transport option selection to enable applications to seamlessly take advantage of the protocol diversity in the Internet.

The next chapters of the thesis focus on the building blocks needed to realize transport option selection within the OS. In Chapter 3, we introduce the concept of *Socket Intents*. Socket Intents allow applications to share their knowledge about their communication pattern and express performance preferences in a generic and portable way. Therefore, they enable the OS to take the applications' intents into account, thus, enabling us to move transport option selection from the application into the OS. We describe Socket Intents in the version we tried to standardize in the IETF [6] and provide comprehensive examples of how Socket Intents are used. In Chapter 4, we present a *generic policy framework* for realizing transport option selection within the OS. It collects the possible paths, endpoints, and protocol stack compositions together with the Socket Intents provided by the application and uses as input to the policy. The policy itself is composed from *policy entries* that can be provided by different stakeholders. As some of this information needs to be acquired on-the-fly, e.g., using name resolution, the policy is evaluated while its inputs are still being acquired. The best-ranked transport configurations then compete in a connection-establishment race — Happy Eyeballs on Steroids — to select the best transport configuration.

In Chapter 5, we evaluate benefits of using multiple paths to improve Web browsing performance. We use a custom *Web Transfer Simulator* to simulate the Web page load using different policies. These policies include just using a single interface, simple round-robin distribution of HTTP requests, using MPTCP and our $\mathcal{EAF}$ policy, optionally in combination with MPTCP. Our simulation study uses a full factorial experimental design covering the Alexa Top 100 and Top 1000 Web sites for a wide range of network characteristics, and thus, resulting in 9M simulations. We see that our $\mathcal{EAF}$ policy provides a speedup in more than 42% without using MPTCP and in 63% of the cases when using MPTCP. In about 20% of the cases, our $\mathcal{EAF}$ policy provide a speedup of two or more. In the remainder of cases, the page load is not bandwidth bound within our scenario and only benefits from choosing the lowest latency path. We also compare our $\mathcal{EAF}$ policy to vanilla MPTCP and find that, for our use cases, distributing HTTP requests on can achieve the same performance benefits as MPTCP in most cases. By looking at the factors that show the highest performance gains, we conclude that for this use case, the gains are most prevalent in cases where we can use multiple paths to overcome bandwidth limitations or choose the lowest-latency path for latency-sensitive transfers.

Finally, in Chapter 6, we evaluate the implementability of transport option selection as an extension to the BSD Socket API by building a prototype. Our Multi-Access Prototype augments the BSD Socket API and implements three different approaches how to integrate transport option selection as API variants. On the one hand, it shows that it is possible to integrate path selection and endpoint selection into the BSD Socket API. However, on the other hand, integrating the third dimension — protocol stack composition — is shown infeasible due to the structure of the BSD Socket API. Our Multi-Access Prototype also unveils many side effects of the BSD Socket API design that limit the effectiveness of transport option selection as part of the BSD Socket API. For supporting transport option selection at message granularity, the augmented API diverges heavily from the regular usage pattern of

the BSD Socket API. After building this prototype, we conclude that the vanilla BSD Socket API is not suited to exploit transport diversity and should be considered broken beyond repair. Therefore, we believe that we urgently need to rethink the programming interface for network communication and replace the BSD Socket API with an event-based API that supports different communication unit granularities and transport semantics.

7.2 Lessons Learned

Transport option selection is complex – complex to systematize, complex to implement and complex to evaluate. This complexity mainly stems from the externalities we have to consider.

The first example of such externalities is the layered design of the Internet. While at the time the "End-to-end Arguments in System Design" [20] was written, it was hoped that such a design should allow exchanging the protocols at each layer, the reliance on specific protocol behavior and the lack of clearly specified interfaces renders protocol stack composition impossible in today's implementations. In Chapter 2, we give an overview of the design space based on the definition of the protocols, but ignore externalities in the implementations. To realize protocol stack composition in practice, we indeed have to re-implement a lot of software to eliminate a lot of implicit assumptions and have to replace the whole networking API largely used based on them. Therefore, we had to refrain from realizing protocol stack composition in our Multi-Access Prototype. While the IETF is currently trying hard to find, document, and work around such externalities in maintenance and extensions of existing protocols, they try to minimize them in the next generation transport protocols like QUIC [17–19] and enforce clean interfaces. The same may be necessary for a replacement of the BSD Socket API in order to realize protocol stack composition in the way we outline in Chapter 2 and 4.

An example of the number of externalities involved in transport option selection is provided by the implementation of our Multi-Access Prototype, which took the biggest effort of this thesis. To demonstrate the feasibility of transport option selection, we extend the BSD Socket API. While we discuss this in more detail in Section 6.5, we want to acknowledge this here as an example of the trade-off between practical applicability, and the amount of effort to answer a research question: The implementation as an extension of the BSD Socket API is a good argument for the practical applicability, but the external constraints of this choice account for most of the effort needed to show the feasibility of transport option selection within the OS.

Finally, the use-case we chose for our performance evaluation is a prime example of stacking externalities. Web performance is depending on many factors, including the Web site content, embedded JavaScripts, Web browser implementation, object caching, DNS caching, the network stack, the paths used, congestion control algorithms used, network conditions, CDN interactions, and the server implementation.

In a real-world study, most of these factors change over time or on a per-request basis. To get reasonable results for the one factor we wanted to change — the path — we chose two approaches: Simulating the download of the objects of the Web site (see Chapter 5) and using a custom proxy while emulating network conditions in a testbed (see Section 6.4). Both approaches have weaknesses: While the simulator approach lacks realism by ignoring all externalities, the testbed study only provided results for a few Web sites of the Alexa Top 100. The Web sites did not load as anticipated because of externalities, e.g., advertisements loaded based on external recommendation services or objects loaded from non-deterministic Java Scrips. We could not get this study approach to a reasonable size dataset as debugging and somehow fixing these problems requires manual intervention. The complex set of externalities and the resulting complex problems make Web performance a pretty bad use case for showing something actually works.

7.3 Future Work

This thesis provides a foundation of automated multi-path aware transport option selection within the OS. However, there are several aspects in transport option selection that need further work.

The first area of research arises alongside the definition of a policy: While our generalized policy framework allows expressing which transport options to prefer based on a variety of conditions, it does not answer the question when transport options are preferable. To approach this question, one has to answer which metrics are useful as input to transport option selection, what measures based on these metrics are most beneficial to decide upon, and how to combine multiple optimization objectives.

In extension to that, policies might not only want to balance multiple objectives of a single communication unit, but tackle conflicting objectives of multiple communications by multiple applications. While TCP fairness and Quality of Service (QoS) queueing schemes are already balancing conflicts regarding bandwidth and delay, what other cross-application tradeoffs need to be addressed? An example of such a tradeoff we imagine is adjusting the packet loss probability across different kinds of messages, e.g., using policy controlled packet pacing. What additional benefits can be achieved by automatically tuning transport protocols for the overall traffic situations?

A different set of challenges arises with regards to implementing our generalized policy framework. Some of them are engineering challenges, e.g., how to represent policy entries and what are good weights to assign Other challenges relate to computation complexity and whether it is possible to represent policy entries in a non Turing-complete way to address runtime and security concerns.

In the area of Socket Intents, we see open questions with regards to the individual Socket Intents Types: Which Socket Intent Types are useful? How to evaluate usefulness independently of a specific policy? Which Socket Intent Types will be adopted by developers? The feedback from bringing this work to the ITEF affirmed our belief that these are the key questions that will decide about the deployment of Socket Intents.

Another future challenge relates to the limitations of the BSD Socket API we encountered in Chapter 6: Which primitives should a future Socket API provide to enable transport option selection at all granularities of communication? How should this API handle cases where the outcome of transport option selection can only provide transport configurations with a different granularity than requested, i.e., if the system has to fall back from message-granularity to stream-granularity provided by TCP? After bringing our work to the IETF, we joined efforts with other researchers and industry people to design a next-generation Transport Services API [10–12], which is already being implemented by the Apple networking team [87].

Finally, a transport option selection framework does not have to be limited to the end-host, but may have direct interactions with the network. There are already approaches to share network information with the end-host [88, 89] and ideas how software-defined networks can interact with end-host policies [67], but these show no significant deployment yet. While we consider application awareness at the core of the Internet infeasible for various reasons, information provided by Socket Intents and decisions of a local transport option selection policy can provide valuable input to adaptive access technologies like cognitive radios.

7.4 Outlook

When looking at the Internet in 30 years, what do we expect? In our vision, automated transport option selection within the OS is a standard functionality of any end-host, that is not extremely resource constrained. Most applications use *Socket Intents* to make the OS aware of their communication preferences and anticipated communication pattern. With this kind of automation, a variety of transport protocols optimized for different communication needs is not only available, but actively used. New protocols can be deployed easily because Happy Eyeballs on Steroids (HEoS) is used for every communication setup and automatically falls back to proven protocols with a delay of a few milliseconds when the new protocol is not available for a specific endpoint.

To make this vision come true, we have to get rid of legacy BSD Sockets. They are, from our perspective, the main obstacle for automated transport option selection. We ought to replace it with something like the TAPS API. This evolution requires much work within the IETF and other standard bodies and might take as long as the migration from IPv4 to IPv6, but is in our opinion worth the afford.

Glossary

ANDSF access network discovery and selection function.

APN Access Point Name (in cellular networks).

AQM Active Queue Management [57].

BANANA-box An on-path device that is able to split flows across multiple access networks to aggregate bandwidth. The IETF working group "BANdwidth Aggregation for interNet Access" (BANANA) is currently in progress of standardizing such a solution.

CDN content delivery network.

CPE customer premise equipment.

DCCP Datagram Congestion Control Protocol.

DiffServ Differentiated Services.

DSCP Differentiated Services Code Point [63, 64].

EAF Earliest Arrival First.

ECDF Empirical Cumulative Distribution Function.

ECMP Equal Cost Multi-Path Routing [43].

ECN Explicit Congestion Notification [56].

FEC Forward Error Correction.

future A surrogate value returned in place of the actual result of an asynchronous operation.

Happy Eyeballs An IPv6 transition technology that starts connection via IPv4 and IPv6 in parallel and using the first connection, but biases the "connection racing" by giving IPv6 a few *ms* advantage [49].

HAR HTTP Archive.

HEoS Happy Eyeballs on Steroids.

HTTP Hypertext Transfer Protocol.

IFOM	IP flow mobility for Proxy Mobile IPv6 [29].
IMS	IP Multimedia Subsystem.
IntServ	Integrated Services.
IPC	inter process communication.
IPSec	Internet Protocol Security [28].
MAM	Multi-Access Manager.
MPTCP	Multi-Path TCP [14–16].
NEMO	Flow Bindings in Mobile IPv6 and Network Mobility [27].
OS	operating system.
PDU	Protocol Data Unit.
protocol stack composition	The process of choosing a set of protocols for a given communication unit.
PvD	Provisioning Domain [30].
QoS	Quality of Service.
QUIC	A UDP-Based Multiplexed and Secure Transport [17–19].
RTT	round-trip time.
SCTP	Stream Control Transmission Protocol [90].
SIP	Session Initialization Protocol [91].
SLAAC	IPv6 Stateless Address Autoconfiguration [32].
TAPS API	Transport Services API [10–12].
TCP	Transmission Control Protocol [92].
transport configuration	A set of transport options enabling communication between two endpoints.
transport option	A means to transport data, e.g., in the Internet. Technically, this can be an endpoint, a path or a protocol available.

UDP	User Datagram Protocol [93].
WAN	Wide Area Network.

Bibliography

[1] Philipp S. Schmidt, Theresa Enghardt, Ramin Khalili, and Anja Feldmann. "Socket Intents: Leveraging Application Awareness for Multi-access Connectivity". In: *ACM CoNEXT*. Santa Barbara, California, USA: ACM, 2013, pp. 295–300. ISBN: 978-1-4503-2101-3. DOI: 10.1145/2535372.2535405 (cit. on pp. vii, 39, 46).

[2] Philipp S. Schmidt, Ruben Merz, and Anja Feldmann. "A first look at multi-access connectivity for mobile networking". In: *Proceedings of the 2012 ACM workshop on Capacity sharing*. CSWS '12. Nice, France: ACM, 2012, pp. 9–14. ISBN: 978-1-4503-1780-1. DOI: 10.1145/2413219.2413224 (cit. on p. vii).

[3] Philipp S. Tiesel, Bernd May, and Anja Feldmann. "Multi-Homed on a Single Link: Using Multiple IPv6 Access Networks". In: *Proceedings of the 2016 Applied Networking Research Workshop*. ANRW '16. Berlin, Germany: ACM, 2016, pp. 16–18. ISBN: 978-1-4503-4443-2. DOI: 10.1145/2959424.2959434 (cit. on pp. vii, 19, 92).

[4] Philipp S. Tiesel, Theresa Enghardt, Mirko Palmer, and Anja Feldmann. *Socket Intents: OS Support for Using Multiple Access Networks and its Benefits for Web Browsing*. Submitted to ACM/IEEE Transactions on Networking, initial version (June 2017) accepted with major revision, revised version (Apr. 2018) rejected. Apr. 2018. arXiv: 1804.08484 (cit. on p. vii).

[5] Philipp Tiesel, Theresa Enghardt, and Anja Feldmann. *Communication Units Granularity Considerations for Multi-Path Aware Transport Selection*. Internet-Draft draft-tiesel-taps-communitgrany-01. IETF Secretariat, Oct. 2017. URL: http://www.ietf.org/internet-drafts/draft-tiesel-taps-communitgrany-01.txt (cit. on p. viii).

[6] Philipp Tiesel, Theresa Enghardt, and Anja Feldmann. *Socket Intents*. Internet-Draft draft-tiesel-taps-socketintents-01. IETF Secretariat, Oct. 2017. URL: http://www.ietf.org/internet-drafts/draft-tiesel-taps-socketintents-01.txt (cit. on pp. viii, 39, 42, 108).

[7] Philipp Tiesel and Theresa Enghardt. *A Socket Intents Prototype for the BSD Socket API - Experiences, Lessons Learned and Considerations*. Internet-Draft draft-tiesel-taps-socketintents-bsdsockets-01. IETF Secretariat, Mar. 2018. URL: https://www.ietf.org/archive/id/draft-tiesel-taps-socketintents-bsdsockets-01.txt (cit. on p. viii).

[8] Mirko Palmer. "Implementation and Evaluation of Multi-Access Policies for MPTCP Path Management in User-Space". MA thesis. TU Berlin, June 2015 (cit. on p. viii).

[9] Tobias Kaiser. "Enabling Asynchronous I/O for the Socket Intent Framework". BA thesis. TU Berlin, June 2016 (cit. on p. viii).

[10] Tommy Pauly, Brian Trammell, Anna Brunstrom, Gorry Fairhurst, Colin Perkins, Philipp Tiesel, and Christopher Wood. *An Architecture for Transport Services*. Internet-Draft draft-ietf-taps-arch-02. IETF Secretariat, Oct. 2018. URL: https://www.ietf.org/archive/id/draft-ietf-taps-arch-02.txt (cit. on pp. ix, 79, 105, 111, 114).

[11] Brian Trammell, Michael Welzl, Theresa Enghardt, Gorry Fairhurst, Mirja Kuehlewind, Colin Perkins, Philipp Tiesel, and Christopher Wood. *An Abstract Application Layer Interface to Transport Services*. Internet-Draft draft-ietf-taps-interface-02. IETF Secretariat, Oct. 2018. URL: https://www.ietf.org/archive/id/draft-ietf-taps-interface-02.txt (cit. on pp. ix, 39, 41, 79, 105, 111, 114).

[12] Anna Brunstrom, Tommy Pauly, Theresa Enghardt, Karl-Johan Grinnemo, Tom Jones, Philipp Tiesel, Colin Perkins, and Michael Welzl. *Implementing Interfaces to Transport Services*. Internet-Draft draft-ietf-taps-impl-02. IETF Secretariat, Oct. 2018. URL: https://www.ietf.org/archive/id/draft-ietf-taps-impl-02.txt (cit. on pp. ix, 58, 79, 105, 111, 114).

[13] Apple Inc. *About Wi-Fi Assist*. 2016. URL: https://support.apple.com/en-us/HT205296 (visited on 09/27/2017) (cit. on p. 2).

[14] A. Ford et al. *Architectural Guidelines for Multipath TCP Development*. RFC 6182 (Informational). Mar. 2011. URL: http://www.ietf.org/rfc/rfc6182.txt (cit. on pp. 2, 84, 94, 114).

[15] Christoph Paasch and Olivier Bonaventure. "Multipath TCP". In: *Queue* 12.2 (Feb. 2014), 40:40–40:51. ISSN: 1542-7730. DOI: 10.1145/2578508.2591369 (cit. on pp. 2, 84, 94, 114).

[16] Olivier Bonaventure and S Seo. "Multipath TCP deployments". In: *IETF Journal* 12.2 (2016), pp. 24–27 (cit. on pp. 2, 84, 94, 114).

[17] Janardhan Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft draft-ietf-quic-transport-07. IETF Secretariat, Oct. 2017. URL: http://www.ietf.org/archive/id/draft-ietf-quic-transport-07.txt (cit. on pp. 7, 109, 114).

[18] Janardhan Iyengar and Ian Swett. *QUIC Loss Detection and Congestion Control*. Internet-Draft draft-ietf-quic-recovery-07. IETF Secretariat, Nov. 2017. URL: http://www.ietf.org/archive/id/draft-ietf-quic-recovery-07.txt (cit. on pp. 7, 109, 114).

[19] Martin Thomson and Sean Turner. *Using Transport Layer Security (TLS) to Secure QUIC*. Internet-Draft draft-ietf-quic-tls-07. IETF Secretariat, Oct. 2017. URL: http://www.ietf.org/archive/id/draft-ietf-quic-tls-07.txt (cit. on pp. 7, 109, 114).

[20] J. H. Saltzer, D. P. Reed, and D. D. Clark. "End-to-end Arguments in System Design". In: *ACM Trans. Computer Systems* 2.4 (Nov. 1984), pp. 277–288. ISSN: 0734-2071. DOI: 10.1145/357401.357402 (cit. on pp. 7, 8, 109).

[21] B. Carpenter. *Architectural Principles of the Internet*. RFC 1958 (Informational). Updated by RFC 3439. June 1996. URL: http://www.ietf.org/rfc/rfc1958.txt (cit. on p. 8).

[22] R. Bush and D. Meyer. *Some Internet Architectural Guidelines and Philosophy*. RFC 3439 (Informational). Dec. 2002. URL: http://www.ietf.org/rfc/rfc3439.txt (cit. on p. 8).

[23] J. Kempf, R. Austein, and IAB. *The Rise of the Middle and the Future of End-to-End: Reflections on the Evolution of the Internet Architecture*. RFC 3724 (Informational). Mar. 2004. URL: http://www.ietf.org/rfc/rfc3724.txt (cit. on p. 8).

[24] *Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*. standard. Geneva, Switzerland: International Organization for Standardization, 1994. URL: https://www.iso.org/standard/20269.html (cit. on p. 8).

[25] D. D. Clark et al. "Tussle in cyberspace: defining tomorrow's Internet". In: *IEEE/ACM Transactions on Networking* 13.3 (June 2005), pp. 462–475. ISSN: 1063-6692. DOI: 10.1109/TNET.2005.850224 (cit. on p. 8).

[26] M. Belshe, R. Peon, and M. Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540 (Proposed Standard). May 2015. URL: http://www.ietf.org/rfc/rfc7540.txt (cit. on p. 12).

[27] G. Tsirtsis et al. *Flow Bindings in Mobile IPv6 and Network Mobility (NEMO) Basic Support*. RFC 6089 (Proposed Standard). Jan. 2011. URL: http://www.ietf.org/rfc/rfc6089.txt (cit. on pp. 16, 17, 22, 114).

[28] S. Kent and K. Seo. *Security Architecture for the Internet Protocol*. RFC 4301 (Proposed Standard). Updated by RFCs 6040, 7619. Dec. 2005. URL: http://www.ietf.org/rfc/rfc4301.txt (cit. on pp. 16, 114).

[29] CJ. Bernardos. *Proxy Mobile IPv6 Extensions to Support Flow Mobility*. RFC 7864 (Proposed Standard). May 2016. URL: http://www.ietf.org/rfc/rfc7864.txt (cit. on pp. 17, 20, 22, 114).

[30] D. Anipko. *Multiple Provisioning Domain Architecture*. RFC 7556 (Informational). June 2015. URL: http://www.ietf.org/rfc/rfc7556.txt (cit. on pp. 17–19, 25, 114).

[31] Srikanth Sundaresan et al. "Broadband internet performance: a view from the gateway". In: *ACM CCR*. Vol. 41. 4. ACM, 2011, pp. 134–145 (cit. on pp. 18, 62).

[32] S. Thomson, T. Narten, and T. Jinmei. *IPv6 Stateless Address Autoconfiguration*. RFC 4862 (Draft Standard). Updated by RFC 7527. Sept. 2007. URL: http://www.ietf.org/rfc/rfc4862.txt (cit. on pp. 19, 114).

[33] Pierre Pfister et al. *Discovering Provisioning Domain Names and Data*. Internet-Draft draft-bruneau-intarea-provisioning-domains-02. IETF Secretariat, July 2017. URL: http://www.ietf.org/archive/id/draft-bruneau-intarea-provisioning-domains-02.txt (cit. on p. 19).

[34] B. Carpenter and S. Brim. *Middleboxes: Taxonomy and Issues*. RFC 3234 (Informational). Feb. 2002. URL: http://www.ietf.org/rfc/rfc3234.txt (cit. on p. 19).

[35] Michio Honda et al. "Is It Still Possible to Extend TCP?" In: *ACM IMC*. Berlin, Germany: ACM, 2011, pp. 181–194. ISBN: 978-1-4503-1013-0. DOI: 10.1145/2068816.2068834. URL: http://doi.acm.org/10.1145/2068816.2068834 (cit. on p. 19).

[36] Gregory Detal et al. "Revealing Middlebox Interference with Tracebox". In: *ACM IMC*. Barcelona, Spain: ACM, 2013, pp. 1–8. ISBN: 978-1-4503-1953-9. DOI: 10.1145/2504730.2504757. URL: http://doi.acm.org/10.1145/2504730.2504757 (cit. on p. 19).

[37] Adnan Aijaz, Hamid Aghvami, and Mojdeh Amani. "A survey on mobile data offloading: technical and business perspectives". In: *Wireless Communications, IEEE Transactions on* 20.2 (2013), pp. 104–112 (cit. on p. 20).

[38] Joohyun Lee et al. "Economics of WiFi offloading: Trading delay for cellular capacity". In: *Wireless Communications, IEEE Transactions on* 13.3 (2014), pp. 1540–1554 (cit. on p. 20).

[39] Dave Allan and Hongyu LI. *TR-348: Hybrid Access Broadband Network Architecture*. Tech. rep. Broadband Forum, Aug. 2016. URL: https://www.broadband-forum.org/technical/download/TR-348.pdf (cit. on p. 20).

[40] T. Melia and S. Gundavelli. *Logical-Interface Support for IP Hosts with Multi-Access Support*. RFC 7847 (Informational). May 2016. URL: http://www.ietf.org/rfc/rfc7847.txt (cit. on p. 20).

[41] S. Gundavelli et al. *Proxy Mobile IPv6*. RFC 5213 (Proposed Standard). Updated by RFCs 6543, 7864. Aug. 2008. URL: http://www.ietf.org/rfc/rfc5213.txt (cit. on p. 20).

[42] Olivier Bonaventure, Mohamed Boucadair, and Bart Peirens. *0-RTT TCP converters*. Internet-Draft draft-bonaventure-mptcp-converters-01. IETF Secretariat, July 2017. URL: http://www.ietf.org/archive/id/draft-bonaventure-mptcp-converters-01.txt (cit. on pp. 20, 23).

[43] D. Thaler and C. Hopps. *Multipath Issues in Unicast and Multicast Next-Hop Selection*. RFC 2991 (Informational). Nov. 2000. URL: http://www.ietf.org/rfc/rfc2991.txt (cit. on pp. 22, 113).

[44] Varun Singh et al. *Multipath RTP (MPRTP)*. Internet-Draft draft-ietf-avtcore-mprtp-03. IETF Secretariat, July 2016. URL: http://www.ietf.org/archive/id/draft-ietf-avtcore-mprtp-03.txt (cit. on p. 23).

[45] R. Stewart et al. *Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)*. RFC 6458 (Informational). Dec. 2011. URL: http://www.ietf.org/rfc/rfc6458.txt (cit. on p. 23).

[46] J. Jeong et al. *IPv6 Router Advertisement Options for DNS Configuration*. RFC 6106 (Proposed Standard). Obsoleted by RFC 8106. Nov. 2010. URL: http://www.ietf.org/rfc/rfc6106.txt (cit. on p. 25).

[47] C. Contavalli et al. *Client Subnet in DNS Queries*. RFC 7871 (Informational). May 2016. URL: http://www.ietf.org/rfc/rfc7871.txt (cit. on p. 25).

[48] T. Savolainen, J. Kato, and T. Lemon. *Improved Recursive DNS Server Selection for Multi-Interfaced Nodes*. RFC 6731 (Proposed Standard). Dec. 2012. URL: http://www.ietf.org/rfc/rfc6731.txt (cit. on p. 25).

[49] D. Wing and A. Yourtchenko. *Happy Eyeballs: Success with Dual-Stack Hosts*. RFC 6555 (Proposed Standard). Apr. 2012. URL: http://www.ietf.org/rfc/rfc6555.txt (cit. on pp. 26, 52, 57, 113).

[50] M. Mathis et al. *TCP Selective Acknowledgment Options*. RFC 2018 (Proposed Standard). Oct. 1996. URL: http://www.ietf.org/rfc/rfc2018.txt (cit. on p. 27).

[51] *QUIC FEC v1*. Feb. 2016. URL: https://docs.google.com/document/d/1Hg1SaLEl6T4rEU9j-isovCo8VEjjnuCPTcLNJewj7Nk/ (visited on 09/27/2017) (cit. on p. 28).

[52] Jari Arkko. *Security and Pervasive Monitoring*. Sept. 2013. URL: https://www.ietf.org/blog/2013/09/security-and-pervasive-monitoring/ (visited on 09/27/2017) (cit. on p. 28).

[53] Giorgos Papastergiou et al. "De-ossifying the internet transport layer: A survey and future perspectives". In: *IEEE Communications Surveys & Tutorials* 19.1 (), pp. 619–639. DOI: 10.1109/COMST.2016.2626780 (cit. on p. 28).

[54] Michio Honda et al. "Is It Still Possible to Extend TCP?" In: *ACM IMC*. Berlin, Germany: ACM, 2011, pp. 181–194. ISBN: 978-1-4503-1013-0. DOI: 10.1145/2068816.2068834 (cit. on p. 29).

[55] Mirja Kuehlewind, Tommy Pauly, and Christopher Wood. *Separating Crypto Negotiation and Communication*. Internet-Draft draft-kuehlewind-taps-crypto-sep-00. IETF Secretariat, July 2017. URL: http://www.ietf.org/archive/id/draft-kuehlewind-taps-crypto-sep-00.txt (cit. on pp. 29, 36).

[56] K. Ramakrishnan, S. Floyd, and D. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168 (Proposed Standard). Updated by RFCs 4301, 6040. Sept. 2001. URL: http://www.ietf.org/rfc/rfc3168.txt (cit. on pp. 29, 113).

[57] F. Baker and G. Fairhurst. *IETF Recommendations Regarding Active Queue Management*. RFC 7567 (Best Current Practice). July 2015. URL: http://www.ietf.org/rfc/rfc7567.txt (cit. on pp. 29, 113).

[58] Olga Bondarenko et al. "Ultra-low Delay for All: Live Experience, Live Analysis". In: *Proceedings of the 7th International Conference on Multimedia Systems*. MMSys '16. Klagenfurt, Austria: ACM, 2016, 33:1–33:4. ISBN: 978-1-4503-4297-1. DOI: 10.1145/2910017.2910633. URL: http://doi.acm.org/10.1145/2910017.2910633 (cit. on p. 29).

[59] C. Perkins. *IP Mobility Support for IPv4, Revised*. RFC 5944 (Proposed Standard). Nov. 2010. URL: http://www.ietf.org/rfc/rfc5944.txt (cit. on p. 33).

[60] C. Perkins, D. Johnson, and J. Arkko. *Mobility Support in IPv6*. RFC 6275 (Proposed Standard). July 2011. URL: http://www.ietf.org/rfc/rfc6275.txt (cit. on p. 33).

[61] Monia Ghobadi et al. "Trickle: Rate Limiting YouTube Video Streaming". In: *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, 2012, pp. 191–196. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/ghobadi (cit. on p. 40).

[62] *Dynamic adaptive streaming over HTTP (DASH) - Part 1: Media presentation description and segment formats*. standard. Geneva, Switzerland: International Organization for Standardization, 2011. URL: https://www.iso.org/standard/65274.html (cit. on pp. 45, 104).

[63] K. Nichols et al. *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*. RFC 2474 (Proposed Standard). Updated by RFCs 3168, 3260. Dec. 1998. URL: http://www.ietf.org/rfc/rfc2474.txt (cit. on pp. 45, 113).

[64] D. Grossman. *New Terminology and Clarifications for Diffserv*. RFC 3260 (Informational). Apr. 2002. URL: http://www.ietf.org/rfc/rfc3260.txt (cit. on pp. 45, 113).

[65] H. Abbasi et al. "A Quality-of-Service Enhanced Socket API in GNU/Linux". In: *Real-Time Linux Workshop*. 2002 (cit. on p. 46).

[66] B. D. Higgins et al. "Intentional networking: opportunistic exploitation of mobile network diversity". In: *ACM MobiCom*. ACM, 2010, pp. 73–84 (cit. on p. 46).

[67] N. Khademi et al. "NEAT: A Platform- and Protocol-Independent Internet Transport API". In: *IEEE Communications Magazine* 55.6 (2017), pp. 46–54. ISSN: 0163-6804. DOI: 10.1109/MCOM.2017.1601052 (cit. on pp. 46, 50, 111).

[68] J. Babiarz, K. Chan, and F. Baker. *Configuration Guidelines for DiffServ Service Classes*. RFC 4594 (Informational). Updated by RFC 5865. Aug. 2006. URL: http://www.ietf.org/rfc/rfc4594.txt (cit. on p. 48).

[69] Brian Trammell, Colin Perkins, and Mirja Kuehlewind. "Post Sockets: Towards an Evolvable Network Transport Interface". In: *IFIP Networking Conference*. Stockholm, Sweden, July 2017. ISBN: 978-3-901882-94-4. URL: http://dl.ifip.org/db/conf/networking/networking2017/1570348319.pdf (cit. on p. 50).

[70] Tommy Pauly. *Guidelines for Racing During Connection Establishment*. Internet-Draft draft-pauly-taps-guidelines-01. IETF Secretariat, Oct. 2017. URL: http://www.ietf.org/archive/id/draft-pauly-taps-guidelines-01.txt (cit. on pp. 52, 54).

[71] J. Nagle. *Congestion Control in IP/TCP Internetworks*. RFC 896 (Historic). Obsoleted by RFC 7805. Jan. 1984. URL: http://www.ietf.org/rfc/rfc896.txt (cit. on p. 56).

[72] Giorgos Papastergiou et al. “On the Cost of Using Happy Eyeballs for Transport Protocol Selection”. In: *Proceedings of the 2016 Applied Networking Research Workshop*. ANRW ’16. Berlin, Germany: ACM, 2016, pp. 45–51. ISBN: 978-1-4503-4443-2. DOI: 10.1145/2959424.2959437 (cit. on p. 58).

[73] S. Egger et al. “Waiting times in quality of experience for web based services”. In: *Quality of Multimedia Experience (QoMEX), 2012 Fourth International Workshop on*. IEEE. July 2012, pp. 86–96. DOI: 10.1109/QoMEX.2012.6263888 (cit. on p. 60).

[74] Conor Kelton et al. “Improving user perceived page load times using gaze”. In: *USENIX NSDI*. Vol. 17. USENIX Association, 2017, pp. 545–559 (cit. on p. 60).

[75] Jörg Wallerich et al. “A Methodology for Studying Persistency Aspects of Internet Flows”. In: *ACM CCR* 35.2 (Apr. 2005), pp. 23–36. ISSN: 0146-4833. DOI: 10.1145/1064413.1064417. URL: http://doi.acm.org/10.1145/1064413.1064417 (cit. on p. 62).

[76] Sunghwan Ihm and Vivek S Pai. “Towards understanding modern web traffic”. In: *ACM IMC*. ACM. 2011, pp. 295–312 (cit. on p. 66).

[77] Michael Butkiewicz, Harsha V Madhyastha, and Vyas Sekar. “Understanding website complexity: measurements, metrics, and implications”. In: *ACM IMC*. ACM. Nov. 2011, pp. 313–328. DOI: 10.1145/2068816.2068846 (cit. on p. 66).

[78] Ravi Netravali et al. “Polaris: Faster Page Loads Using Fine-grained Dependency Tracking”. In: *USENIX NSDI*. Santa Clara, CA: USENIX Association, Mar. 2016. URL: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/netravali (cit. on p. 67).

[79] J. Chu et al. *Increasing TCP’s Initial Window*. RFC 6928 (Experimental). Apr. 2013. URL: http://www.ietf.org/rfc/rfc6928.txt (cit. on p. 68).

[80] Enric Pujol et al. “Back-Office Web Traffic on The Internet”. In: *ACM IMC*. ACM. 2014, pp. 257–270 (cit. on p. 71).

[81] Stuart Sechrest. *Tutorial Examples of Interprocess Communication in Berkeley UNIX 4.2 BSD*. Tech. rep. 191. University of California, Berkley, 1984. URL: http://digitalassets.lib.berkeley.edu/techreports/ucb/text/CSD-84-191.pdf (cit. on p. 80).

[82] Lucian Popa, Ali Ghodsi, and Ion Stoica. “HTTP As the Narrow Waist of the Future Internet”. In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Hotnets-IX. Monterey, California: ACM, 2010, 6:1–6:6. ISBN: 978-1-4503-0409-2. DOI: 10.1145/1868447.1868453 (cit. on p. 81).

[83] Philipp Richter et al. “Distilling the Internet’s Application Mix from Packet-Sampled Traffic”. English. In: *Passive and Active Measurement*. Ed. by Jelena Mirkovic and Yong Liu. Vol. 8995. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 179–192. ISBN: 978-3-319-15508-1. DOI: 10.1007/978-3-319-15509-8_14 (cit. on p. 81).

[84] Carl A Sunshine. *Host Software*. Tech. rep. 178. 1981. URL: http://www.ietf.org/rfc/ien/ien178.txt (cit. on p. 81).

[85] Linux Foundation. *Netlink(7) Linux Programmer's Manual*. URL: http://man7.org/linux/man-pages/man7/netlink.7.html (cit. on pp. 95, 102).

[86] Patrick Kutter. "Improving Video Streaming QoE Through Multi Access Policies". MA thesis. TU Berlin, Sept. 2015 (cit. on p. 104).

[87] Apple Inc. *About Wi-Fi Assist*. 2018. URL: https://developer.apple.com/documentation/network (visited on 08/29/2018) (cit. on p. 111).

[88] V. K. Gurbani et al. "A survey of research on the application-layer traffic optimization problem and the need for layer cooperation". In: *IEEE Communications Magazine* 47.8 (Aug. 2009), pp. 107–112. ISSN: 0163-6804. DOI: 10.1109/MCOM.2009.5181900 (cit. on p. 111).

[89] R. Alimi et al. *Application-Layer Traffic Optimization (ALTO) Protocol*. RFC 7285 (Proposed Standard). Sept. 2014. URL: http://www.ietf.org/rfc/rfc7285.txt (cit. on p. 111).

[90] R. Stewart. *Stream Control Transmission Protocol*. RFC 4960 (Proposed Standard). Updated by RFCs 6096, 6335, 7053. Sept. 2007. URL: http://www.ietf.org/rfc/rfc4960.txt (cit. on p. 114).

[91] J. Rosenberg et al. *SIP: Session Initiation Protocol*. RFC 3261 (Proposed Standard). Updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630, 5922, 5954, 6026, 6141, 6665, 6878, 7462, 7463, 8217. June 2002. URL: http://www.ietf.org/rfc/rfc3261.txt (cit. on p. 114).

[92] J. Postel. *Transmission Control Protocol*. RFC 793 (INTERNET STANDARD). Updated by RFCs 1122, 3168, 6093, 6528. Sept. 1981. URL: http://www.ietf.org/rfc/rfc793.txt (cit. on p. 114).

[93] J. Postel. *User Datagram Protocol*. RFC 768 (INTERNET STANDARD). Aug. 1980. URL: http://www.ietf.org/rfc/rfc768.txt (cit. on p. 115).