NTNU – Trondheim
Norwegian University of
Science and Technology

Topology in WebRTC Services

Tarjei Klinge Husøy

Master of Science in Communication Technology
Submission date: June 2015
Supervisor: Poul Einar Heegaard, ITEM
Co-supervisor: Svein Willassen, appear.in

Norwegian University of Science and Technology
Department of Telematics

Title: Topology in WebRTC Services

Student: Tarjei Husøy

Problem description:

Current video conferencing services use different topologies and architectures to realize real-time communication. One possible architecture, used by the service appear.in, is a full mesh architecture where each participant in a conversation has a full duplex connection with every other participant in the conversation. Another possible architecture, used by many traditional video conferencing services, is to use a Multipoint Control Unit (MCU). This unit will take in video and voice feeds sent from multiple participants which will then be combined into one video/voice feed that can be sent to all participants. This requires decoding and re-encoding of the streams in the MCU. A further possible architecture, is to send all streams through a central Selective Forwarding Unit (SFU), which will forward streams to select participants, based on available bandwidth and other preferences.

The different architectures for video conferencing have different properties. For example, for a conference with only two participants it is usually desirable to use the full mesh architecture, because the direct communication implies lower latency and better quality. However, with growing number of participants requirements on both bandwidth and CPU (due to extensive encoding and decoding) will imply that using full mesh becomes undesirable. It is therefore likely that the optimal architecture uses a combination of different topologies, depending on the number of participants in a conversation and the resources available to each participant.

The task is to investigate what attributes and requirements a WebRTC conversation needs to perform optimally, and study how different topologies support these requirements.

Responsible professor: Poul Heegaard, ITEM

Supervisor: Svein Yngvar Willassen, appear.in

Abstract

Bandwidth efficient, low latency, cheap – pick two. This has been the traditional trade-off for video conferencing providers, where the network topology has limited achievable performance in many conversation types. Consumers have also suffered under this scheme, as only the biggest companies have been capable of delivering a system that performs in a wide enough range of conversations to grow sustainable. This has limited innovation and made it hard for new providers to enter the market.

This thesis demonstrates how a video conferencing solution can be built using a hybrid network topology, to combine the best properties of peer-to-peer and centralized topologies. For providers utilizing a centralized topology, adopting this work can yield lower costs and better performance for users, while providers utilizing peer-to-peer topologies today can increase the capacity and coverage of their service.

The proposed method dynamically selects the best topology for a given conversation based on characteristics of each device in the conversation, and will balance routed video to best suit each device. The solution is extensible to include arbitrary characteristics of each device or network link when balancing, and special-purpose nodes like supernodes, Selective Forwarding Units (SFUs) and Multipoint Control Units (MCUs) can further enhance the quality. Conversations are modelled as multi-commodity flow networks, and can be solved by any standard LP-solver. Non-linear properties like queuing delays are approximated by piecewise linear functions.

The peer-to-peer video conference solution appear.in is benchmarked, to see how well peer-to-peer services perform over WebRTC, and to illustrate the potential for a solution that can transcend the boundaries of peer-to-peer. The benchmarking results show severe performance issues for Firefox in constrained conversations, and more moderate potential improvements for Chrome. Tools to assist the benchmarking were developed and is included in the appendices.

Sammendrag

Lite båndbreddebruk, lav forsinkelse, billig – velg to. Dette er et kompromiss videokonferansetilbydere har måttet inngå så lenge de har vært bundet til en gitt nettverkstopologi. Topologien setter grensene for hva som er mulig, og drift av et system med tilstrekkelig ytelse til at det adopteres av forbrukere har vært så dyrt at det kun er de største aktørene som kan konkurrere. Innovasjon er vanskelig i et system med så høye inngangskostnader, og både forbrukere og tjenestetilbydere lider som en konsekvens.

Denne oppgaven presenterer en hybrid-topologi for videokonferanser, som kan øke opplevd kvalitet med små ekstra kostnader. For eksisterende tilbydere som baserer seg på sentraliserte nettverk kan denne fremgangsmåten senke kostnader og forbedre ytelsen i mange tilfeller. For eksisterende tilbydere som er basert på jevnbyrdsnett kan metoden redusere tjenestens ressurskrav og øke ytelsen i mange situasjoner som er vanskelige i dag.

Metoden velger dynamisk den beste topologien for hver enkelt samtale, basert på egenskaper ved enhetene i samtalen. Videostrømmene vil rutes i nettet tilpasset hver enkelt enhets kapabiliteter. Løsningen kan utvides til å ta vilkårlige egenskaper ved enhetene og nettverket inn i beregningen, og kan benytte både supernoder, Selective Forwarding Units (SFU) og Multipoint Control Units (MCU) for å øke kvaliteten. Hver samtale modelleres som et multi-commodity flytnettverk og kan løses ved lineærprogrammering. Ikke-lineære egenskaper som køforsinkelse tilnærmes ved stykkevis lineære approksimasjoner.

Jevnbyrdsnettløsningen appear.in testes for å se hvordan videokonferanseløsninger over jevnbyrdsnett bygd på WebRTC yter, og for å illustrere potensialet for løsninger som kan overkomme begrensningene til en gitt topologi. Testene viser store ytelsesproblemer i Firefox på sterkt begrensede klienter, og et mer moderat potensiale for forbedring i Chrome. Verktøy for å bistå testingen ble utviklet og ligger vedlagt.

Acknowledgements

I want to thank my supervising professor at Department of Telematics (ITEM), Poul Einar Heegard, for lots of valuable feedback and many interesting discussions during my work on this thesis, and Svein Willassen, my supervisor at appear.in, for proposing the topic, supplying me with production data from appear.in and valuable insight both into the industry and to many technical aspects of WebRTC.

I'd also like to extend my gratitude to Martin Kirkholt Melhus, both for constructive input on the execution of the experiments and early feedback on the thesis.

Lastly, to the wonderful people I've been sharing an office with the last year, thank you for listening.

Contents

List of Figures

List of Tables

List of Symbols

Continuous Presence	Everyone in a conversation being visible at the same time.
YAML	Human-readable data serialization format, superset of JSON.

List of Acronyms

API Application Programming Interface.

CSV Comma-Separated Values.

GLPK GNU Linear Programming Kit.

ICE Interactive Connectivity Establishment.

IF Influence Factor.

ITEM Department of Telematics.

LP Linear Programming.

MCU Multipoint Control Unit.

NAT Network Address Translation.

NTNU Norwegian University of Science and Technology.

NTP Network Time Protocol.

PSTN Publicly Switched Telephone Network.

QoE Quality of Experience.

QoS Quality of Service.

QoSE Quality of Service Experience.

RTCP RTP Control Protocol.

RTP Real-time Transport Protocol.

RTT Round-Trip Time.

SFU Selective Forwarding Unit.

SIP Session Initiation Protocol.

SLA Service Level Agreement.

STUN Session Traversal Utilities for NAT.

SVC Scalable Video Coding.

TURN Traversal Using Relays around NAT.

VAS Voice-Activated Switching.

VoIP Voice over IP.

W3C World Wide Web Consortium.

WebRTC Web Real-Time Communication.

WFL Weber-Fechner Law.

Chapter 1
Introduction

If I have seen further, it is by standing on the shoulders of giants

Isaac Newton

The web is becoming the primary platform for all communication. People are gradually moving away from solutions provided by their telecommunications company, such as telephony and text messaging, and over to Internet-based services. Moving audio conversations to the Internet has been relatively easy, but as we're now trying to commoditize video conversations, we have a bigger challenge ahead of us. Video conferencing has traditionally been the domain of custom rooms and dedicated hardware, we're now trying to replicate that experience in regular laptops and phones. This has led to performance requirements greater than most user equipment and their connections can handle. This thesis aims to lessen those performance requirements, and make video conferencing feasible in cases where it is not today.

1.1 Problem

appear.in, Firefox Hello and OpenTok are just a few examples of new video conferencing services that have emerged in recent times built on WebRTC, a standard for peer-to-peer communication in the browser. Without any further magic behind the scenes, such solutions will demand a linear increase in bandwidth both upstream and downstream as the number of peers in a conversation grows. This follows from the fact that in a peer-to-peer video conversation, each peer has to encode its own video to each of the other peers, send it to each of those peers, and receive that peer's video. This is expensive in terms of both CPU and bandwidth, and quickly outgrows what many devices are actually capable of.

However, many other video conferencing services does not have this problem, as they ship all video through their own servers. Skype, Google Hangouts, custom

rooms – none of those has this scalability problem[1]. On the downside, they don't have the small latency that is achievable when you route video directly to the receiver, like you would in a peer-to-peer topology. They are also much costlier to operate; peer-to-peer systems only require a provider to help peers find each other, and will never see any of the actual video being transmitted[2]. Which begs the question at the heart of this thesis – can we design a solution that transcends these boundaries and provides high quality service for all combinations of user equipment and connections, without being prohibitively expensive to run?

The main objective of this thesis is to maximize the system's Quality of Experience (QoE). We'll here adhere to the Qualinet white paper definition of QoE, "Degree of delight of the user of a service ..." [LCMP12]. The white paper identifies three primary Influence Factors (IFs) that interrelate to together form the QoE; Human IFs, System IFs and Context IFs. This thesis will focus on two sub-categories of the System IF, namely network-related and device-related System IFs, since these are the ones most easily accessible to WebRTC services. However, the approach is designed with adaptability in mind, such that more IFs can be included in the QoE model if they're available. This enables the system to grow as we gain a better understanding of QoE for video conversations.

However, QoE is a hard thing to maximize, as it's very dependent on the users, which we don't know anything about. We will thus focus on a related term, Quality of Service Experience (QoSE), which is a quantitative measure of how we believe a normal user perceives the quality of a service[3]. Research on psychophysics have yielded a stimulus-perception model known as the Weber-Fechner Law (WFL) [Web34], which says that our perception is a logarithmic function of the magnitude of physical stimuli. [RESA10] has shown this effect to also apply in several domains relevant to this thesis, namely QoE assessment for Voice over IP (VoIP) and data services. This fuels our model of QoSE as logarithmic function of the performance delivered by the service. We aggregate the QoSE for all users in a conversation and try to maximize this value in our approach, the underlying assumption here being that the QoSE will act as a proxy for the QoE, thus by maximizing the QoSE we also maximize the QoE. A formal breakdown of the approach is given in chapter 5.

The problem description mentions studying different topologies and to what extent they can deliver an optimal service. This was re-focused towards *finding* an optimal service, due to how hard it is to study if a service outside your control is optimal. Optimal implies that it needs to respond well in *all* situations, thus the service needs to be tested in many uncommon configurations. This is hard

[1]They do however have another scalability problem: The number of servers they need to accommodate their users grows linearly with the total number of users on the platform.

[2]Generally. In some cases video is sent through the provider for firewall-traversal.

[3]Definitions vary, thus the explicit statement of how QoSE is interpreted in this thesis.

to do without controlling the entire environment. Pure peer-to-peer services like appear.in makes it easier to control the entire environment, as there's no outside server that influences the communication. When peers have found each other in WebRTC-services, the provider doesn't influence the conversation at all, thus by controlling the local network we can control everything. This motivated the pursuit of how a peer-to-peer service performs in hard situations, and from that we could reason about what an optimal service would do, and finally unto studying how that could be accomplished.

1.2 Structure and Methodology

Before trying to solve the problem we'll first evaluate the current video conversation landscape in chapter 2, to get a sense of the status quo. To limit the scope of what we're trying to accomplish, I'll define some test cases in chapter 3 that we'll use throughout the thesis. In chapter 4 we'll evaluate one of the providers on the market today by putting it to the test, running all the test cases from chapter 3 to see how the service performs. Knowing this benchmark helps us evaluate the potential for the approach outlined in this thesis, which we'll take a look at in chapter 5. We'll discuss how the approach can be implemented and its strengths and weaknesses in chapter 6, before summarizing what we've learned in chapter 7.

1.3 Contribution

This thesis proposes an approach to modelling video conferences with known inter-node latencies and bandwidths as a flow network, and shows how an efficient routing for video can be derived from the model using linear programming. The method demonstrates how video conversation services can bridge the performance gap between traditional MCU-backed solutions and peer-to-peer solutions.

The two most popular web browsers as of the time of writing, Google Chrome and Mozilla Firefox, is benchmarked in a set of test cases, which reveals severe flaws in how Firefox handles constrained nodes. Both browsers are shown to have lots of potential for increased performance, which the approach outlined in this thesis could help accomplish. The tests were run with tools developed for this purpose, which are included in the appendices.

1.4 Terminology

A video conference will often be called a conversation in this thesis. The term "video conferencing" carries a lot of luggage from its early history, when the technology was cumbersome, expensive, and only applicable in business scenarios. The movement

WebRTC represents is about the opposite, commoditizing the technology to make it cheap and accessible to everyone, allowing it to enter the private domain. Friends don't "confer" between themselves; they converse. Names say a lot about a technology's intended application, thus if we want the technology to enter the private domain we need a name for it that does not convey business usage. Hence the term used in this thesis: video conversations. The even less formal "video chat" could also have fit the bill, but that feels like it leaves the business side entirely in the dark; conversation feels like a good middle ground that applies to both sides.

1.5 Disclaimer

This thesis does not try to measure or solve for audio transmission, as that's a much simpler problem that can practically always be completed by sending the same stream to all nodes in the conversation. There's always only one stream to encode, it doesn't noticeably affect available bandwidth, and it's already widely deployed. However, results we achieve for video can also be applied to audio streams if the environment is very heavily constrained or further optimization is required, but is out of scope for this thesis.

Chapter 2
Background

In this chapter we'll discuss some technical aspects of video conferencing that affects how we reason about the problem, and evaluate what sort of trade-offs established actors on the market have made.

2.1 WebRTC

This thesis is largely inspired by the efforts of the World Wide Web Consortium (W3C) on Web Real-Time Communication (WebRTC), a technology which enables direct browser-to-browser communication. Building on WebRTC, services like Telenor Digital's appear.in and Telefónica's Hello have come to life, ushering in a new age of communication that does not depend on the traditional GSM infrastructure, but is fueled by faster Internet connections and more capable smartphones. WebRTC is not a finished standard yet, which is why browser support is variable at the moment, but it's expected that support will become more widespread once the specification is finished[1].

It's interesting to note that many of the largest WebRTC communication platforms (like appear.in and Hello, as mentioned) we've seen so far have been developed by the largest players in the traditional communication field, and not from any independent outsider. The big telephone companies do have capabilities other actors don't enjoy, such as being able to freely route calls back over GSM as a fallback solution in case a person is not reachable online, but this has not been a big selling point for the services so far. The services have also largely been focused on video conversations, even though the technology is equally well-suited for pure voice conversations or text-based communication.

In any case, the hard part of the problem is video conversations, as the demands on the user equipment and the connection is far greater than what will ever be

[1] The current status is "working draft", the full specification can be found here: http://www.w3.org/TR/webrtc/

exercised by voice or text. Many services are today artificially limited in size, but often the parties in a conversation will experience trouble before reaching those limits, as their devices have insufficient bandwidth or their CPUs are not capable of encoding enough video streams in parallel.

There are three different implementations of the WebRTC specification that are in extensive use; `libjingle`, which powers Chrome and Opera; Mozilla's, which is tightly coupled with Firefox; and OpenWebRTC, a mobile-first framework for native apps, started by Ericsson Research. There's also WebRTC Microstack[2], but they only provide the data-channel, which means they don't support the audio/video APIs. Some interest for a pure JavaScript implementation has been expressed to ease development of WebRTC-aware server-side applications[3], but that project has not seen much activity since late 2014.

Codec issues have been a heated debate for online video, which we will not reproduce in full here. In summary there's two contenders, H.264 and WebM. The WebM project produces the VPx codecs, which are royalty-free and thus preferred by most browser vendors. H.264 is patent-encumbered and requires licenses for use, but is widely deployed due to its usage on BluRay-discs, most TV-content, etc. Both have their pros and cons, but the web community seems to be slowly moving towards WebM[4].

2.2 A Technical Look at Video Conversations

2.2.1 Encoding

The naïve approach to encoding video is to encode the raw stream from the web camera into several client-optimized streams for transmission. However, H.264 can be encoded with Scalable Video Coding (SVC), which layers several streams with different bitrates into a single stream. A node receiving a SVC stream can then extract layers with the bitrate desired, without re-encoding the entire stream. With VP8 this is sadly not possible, and the only alternative is to send several streams with varying bitrates in parallel. This is not as efficient as sending only a single stream, and the encoding step is costlier in terms of CPU-time. This makes nodes like SFUs more expensive to run with VP8, as splitting a video stream requires decoding and re-encoding the data.

[2] http://opentools.homeip.net/webrtc

[3] https://github.com/webrtcftw/goals/issues/1

[4] The Chromium project announced in 2011 that they would remove H.264 support from the browser, but this has not yet happened. http://blog.chromium.org/2011/01/html-video-codec-support-in-chrome.html

Google has entered a collaboration with Vidyo [Vid13] to bring SVC to VP9, which might bring free-to-use SVC to WebRTC. While both Firefox and Google support decoding VP9 today [Pro14], encoding is not yet supported for either.

Both H.26x and VPx can be hardware-accelerated, and are deployed in several products on the market. Most deployed solutions are decode only, but some, like the Nvidia Tegra 4, also supports encoding [nVi13]. The WebM project maintains open designs for hardware encoders and decoders for VPx.

2.2.2 Continuous Presence vs. Voice-Activated Switching

There are mainly two different ways to do video conversations, Continuous Presence and Voice-Activated Switching (VAS). Continuous Presence means that all parties in the conversation are visible to all other parties at the same time. Voice-Activated Switching (VAS) means that only one party is visible, typically the one detected by the system as talking at any given time. There's also hybrid schemes, like Google Hangouts, where all parties are shown to everyone, but the active speaker is shown bigger than the rest. Each node can override locally who's shown up big.

Clearly, in larger conversations, there's a huge difference in network impact of the two technologies, as a VAS-based solution will always just require a single video link in and out, while Continuous Presence requires bandwidth to scale linearly with the size of the conversation. Figure 2.1 summarizes the possible services that can be provided for different amounts of available bandwidth. A minimal video unit is the smallest bitrate it makes sense to encode video in. This will be service dependent, but we can imagine $\approx$400 kbps to be reasonable.

Continuous presence can be accomplished both in a peer-to-peer topology and centralized topologies, but since a VAS requires insight into the video streams (or at least, the audio streams) to select who is forwarded at any given time, they are only realizable if all the video streams go through centralized servers. In theory it's possible to expand the system to work over peer-to-peer as well, by having each peer use a low-bandwidth data channel to tell nodes whether it's currently speaking or not, but this requires that video streams can be quickly started, stopped, and that the system gracefully handles collisions without falling over. The problem is non-trivial, as it's essentially a question of distributed consensus. Having a single place that handles the question of who's active is a lot simpler to reason about and implement.

Note that even though video is switched in a VAS-system, everyone's audio is usually sent to everyone.

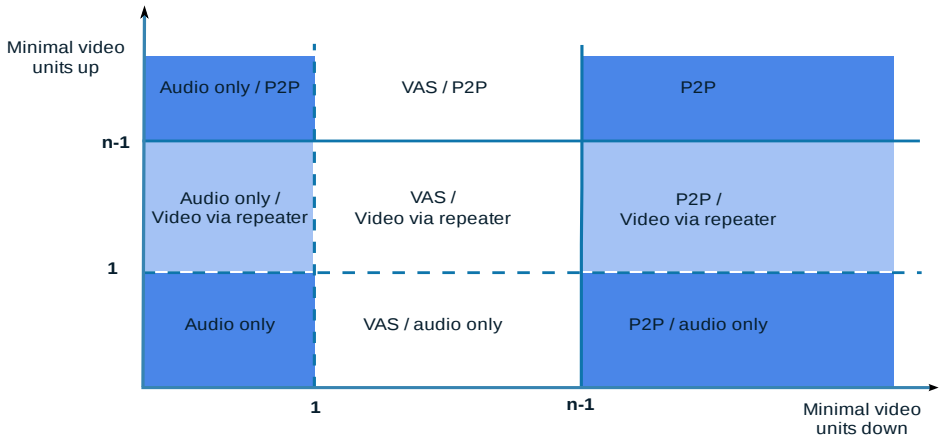


Figure 2.1: The possible services for different upload/download bandwidths. A minimal video is the smallest bitrate where it makes sense to send video. Dark blue is what appear.in can provide today (audio only requires manual configuration), light blue is what's possible with the approach proposed later in this thesis.

2.2.3 NAT and Firewalls

The presence of Network Address Translation (NAT) makes connection establishment between users harder than necessary, as nodes behind a NAT are not aware of their external IP address and port. The Interactive Connectivity Establishment (ICE) framework provides two protocols that can be implemented to alleviate parts of the problem; Session Traversal Utilities for NAT (STUN) and Traversal Using Relays around NAT (TURN). STUN is a very lightweight protocol used to discover your externally visible network address, while TURN-servers act as intermediaries between users that cannot reach each other directly due to firewalls. TURN servers are often the biggest server expense for WebRTC-based solutions, since the provider have to take the cost for the bandwidth consumed by the users.

For WebRTC, STUN would be needed even in the absence of NATs, since JavaScript does not have any APIs for binding to ports, only for establishing outgoing connections. Thus there's no way for a node to announce the port it's accepting connections to before it has established a connection outward, a chicken-and-egg problem STUN resolves.

2.3 The Current Providers

A selection of widely known video conversation services with different network architectures is summarized in section 2.3.

Table 2.1: Network topologies in video conversation services

Service	**Description**
appear.in	Browser-based peer-to-peer WebRTC service
Google Hangouts	Browser-based with Vidyo-powered SFU
Microsoft Skype	Custom application, proprietary peer-to-peer protocol
Cisco TelePresence	Custom hardware, self-hosted or cloud MCUs

Notably absent here is FaceTime, Apple's video conversation service bundled with their devices. FaceTime's absence in this thesis is due to the lack of support for more than two people in a conversation, and the lack of support for non-Apple devices.

We also note that Mozilla just entered the market in collaboration with Telefónica with their Hello service, bundled with recent versions of Firefox[5]. Hello essentially provides the same service as appear.in, just bundled with the browser. Thus anything we say about appear.in applies to Firefox Hello as well (and all other peer-to-peer WebRTC services), and we'll not consider them separately.

2.3.1 appear.in

appear.in is a free peer-to-peer service built on WebRTC that does not require sign-ups or installation of add-ons to your browser. Due to WebRTC not being fully standardized yet, the service is only available on recent versions of either Google Chrome, Mozilla Firefox or Opera, while the OS-provided browsers (Internet Explorer and Safari) have notably not implemented WebRTC yet.[6] appear.in uses Continuous Presence, while providing the user the option of resizing video streams at will.

appear.in is the only solution studied in this thesis that allow true anonymous communication[7]. The combination of continuous presence and a peer-to-peer topology makes the device and network requirements scale linearly with the number of people in a conversation. The service is limited to maximum 8 people in a conversation.

[5]https://www.mozilla.org/en-US/firefox/hello/
[6]Browser support for WebRTC can be tracked at http://iswebrtcreadyyet.com/
[7]Meaning that the provider doesn't know who you are or who you're talking with.

2.3.2 Google Hangouts

Google Hangouts is alongside appear.in the only other service covered in this thesis based in the browser. Hangouts is a merge of several earlier Google communication solutions like Google Talk, Google+ Messenger and the Hangouts feature from Google+. The service uses a Vidyo-provided SFU, and VP8/9 over WebRTC in a non-standard configuration on Chrome [Han14], and requires a plug-in on other browsers [Goo]. A conversation is limited to 10 people. Like was mentioned earlier, Hangouts uses a hybrid-VAS system, where only one party can be shown up big, but the user can override who that is. This strikes a compromise between the high bandwidth requirements of Continuous Presence and the ability to see everyone at the same time.

Hangouts requires you to authenticate with a Google account, which makes anonymous conversations much harder. Hangouts is free to use, but room capacity can be increased to 15 people in the paid Google Apps for Work version.

2.3.3 Skype

Skype is probably the most well-known of the solutions we're looking at, being among the first to offer free video conferencing for personal use way back in 2003. Skype was also among the first to provide a VoIP-service inter-operating with the Publicly Switched Telephone Network (PSTN), easing adoption for new users. The Skype topology is peer-to-peer, built on top of the file-sharing protocol powering Kazaa, which was developed by the same founders [Tho06]. The protocol used is proprietary and requires installing the Skype application. For NAT-traversal, Skype initially used other Skype users known as supernodes as intermediaries, performing the same role as TURN-servers in the ICE framework. After the Microsoft acquisition in 2011, Skype dropped the client-hosted supernodes in favor of Microsoft-hosted ones, justified as a means to improve performance and security for users [Goo12]. A Skype conversation has a soft limit on five people for the best user experience, and a hard limit on 10 users.

Skype requires a standalone application to run, which is available on almost all platforms out there, including Windows, Mac, Linux, Android, iOS, Windows Phone, BlackBerry, most tablets, TVs, video game consoles and more. The benefit of native applications is closer access to hardware for more efficient video processing. The downside is often that they – like Skype – do not use open protocols, are not standardized and offer no transparency.

2.3.4 Cisco

Cisco offers both on-premises, off-premises and hybrid solutions for video conferencing, aimed at the enterprise market. Video is routed through either self-hosted or cloud MCUs or SFUs, using Session Initiation Protocol (SIP) and H.323 for call establishment [CS12]. In the case of only two people in a conversation, calls can be established peer-to-peer. TelePresence enables interoperability with other services that supports SIP and H.323, which through H.320 gateways include devices on legacy networks like the PSTN, such as ISDN videophones. Their core offering is specialized hardware and dedicated rooms for video conferencing (so-called immersive video conferences), but they also have a free service that can be run on end-user computers using a custom application.

2.4 Related Work

Networking and algorithms related to networking is not a new topic, by any stretch of the imagination, and like in most branches of computer science, it's mostly old problems in a new context.

In this thesis we will borrow heavily from previous work on networking and graph algorithms in general, and flow algorithms in particular. Many algorithmic problems can be solved as a linear program, which as a problem was first solved by Fourier in 1827 [Sie01]. Another solution, the simplex algorithm, was first introduced by G.B. Dantzig in 1947 [Sie01], and serves as the basis for the Linear Programming (LP)-solver we'll use for in the sample implementation. Multi-commodity flow networks was introduced to me in [AMO88] by Ahuja, Magnanti and Orlin, which establishes the fundamentals for the approach proposed later in this thesis. The simplex algorithm has been widely adopted for its ease of implementation on computers, and years of exponential growth of computer performance has made solving increasingly large problem sets feasible.

A study at Chalmers in 2014 [GE14] investigated the feasibility of utilizing normal nodes in a video conference as supernodes, routing traffic from less powerful nodes through these nodes to reduce network load. The authors concluded that such a solution is feasible given proper supernode selection, which gives even greater possibilities for a solution utilizing dynamic topologies like presented in this thesis. Pushing as much traffic as possible over client-provided supernodes lowers the cost for the provider, and enables better quality for the users since peers can be closer to each other than to the closest data center. As the study concluded with supernodes being feasible and beneficial, the approach outlined here is developed to be flexible enough to allow nodes to forward video to other nodes.

Chapter 3
Test Cases

Instead of trying to optimize all possible combinations of bandwidths and latencies that occur in the wild, I'll define some test cases here that we can work on. The assumption is that if an approach can be found to efficiently serve these cases, it can serve most others as well.

A summary of the test cases are given in Table 3.1. Note that these are intended to be hard cases, with at least one node being significantly more constrained than the others. Thus, failure to pass these tests do not necessarily imply that the solution is useless, only that there are cases where it'll fail, or perform sub-optimally. Figure 3.1 illustrates the test cases graphically with the all the inter-node latencies.

Are there any trivial cases that can be ignored? As long as there's only two people in a conversation, and they have fairly low latency between each other and sufficient bandwidth, peer-to-peer is the optimal choice in all cases. Initially, it might seem like this would indeed be the case in all conversations with two participants, and not just the good-bandwidth, small-latency onces. However, this is not the case. To illustrate why, consider a conversation between two people, one in Europe and one in Asia. They both have fairly acceptable bandwidth, with 3 Mbps each, which should be plenty to sustain an acceptable video link between them. This might not be the case, as the link quality between them is far more limited due to the long distance and many hops through publicly routed networks, which yields high probability of packet loss and jitter.

However, if both peers have a data center of a distributed VPS provider nearby, to which they can fully utilize their connection, this limitation might be overcome. These distributed VPS providers tend to have established high-quality connections between their own data centers backed by Service Level Agreements (SLAs), which ensures a link quality far greater than what's available to private entities. Because of this, the two peers can improve their video link by routing their traffic through both

data centers.[1]. The latency is however close to unchanged from routing through the data centers, only sustained bandwidth between the peers is improved (and probably less packet loss and jitter).

The test cases are not extensive, but should cover enough corner cases to be able to highlight if services have any issues in constrained environments. The examples cover the low-latency, few peers conversations; the bandwidth-challenged cases; the high-latency conversations; and the very heterogeneous device conversations, where some nodes are severely challenged in terms of either bandwidth or latency compared to the rest.

We assume that the back-end networks are not saturated, and that each user is bandwidth-constrained only by their own connection. By extension, the maximum bandwidth attainable between any pair of nodes in our network is the lesser of the upload bandwidth of the sending party and the download bandwidth of the receiving party. However, latency has to be defined for any pair of the nodes in the network, as this is mostly determined by their geographical location in relation to each other.

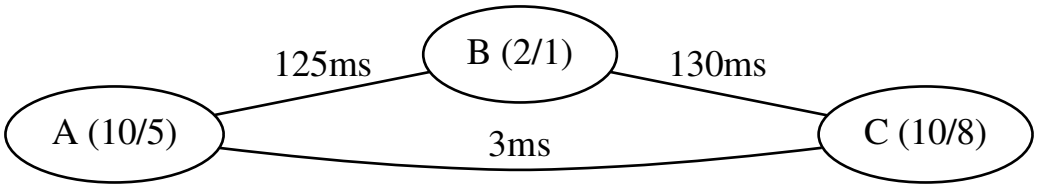
Table 3.1: Summary of Test Cases. n is the conversation size.

Case name	**n**	**Description**
Traveller	3	Two people with decent connections between them, one remote with high latency and severely restricted bandwidth to the others.
Standup	4	Two people on desktop machines with wired connections, one laptop and one tablet on WiFi.
Friends	7	Group split in two locations, each subgroup having short latencies internally, but larger latencies to the other group. Heterogeneous bandwidths across the board.

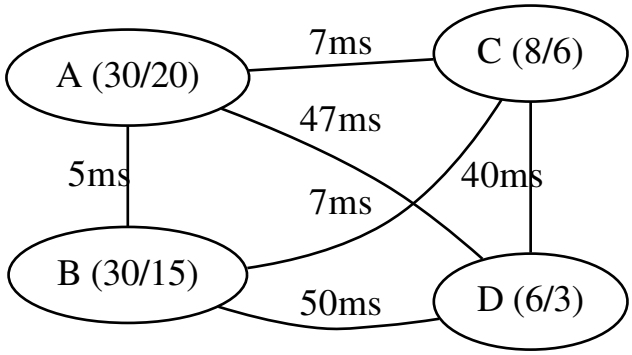
How realistic are these cases? The appear.in data set in Appendix E shows that conversation frequency exponentially decays as a function of conversation size. More than half of the observed conversations are between two people. These conversations were not prioritized for the test cases, as it's easier to make hard test cases with larger conversations. And there's an asymmetry here, browsers that manage to

[1]This is backed by a simple experiment, using DigitalOcean as our VPS provider. From a 100 Mbps university connection in Norway, sustained data rates to their Singapore data center varied greatly, measuring 720 kbps, 29.6 Mbps, 15.2 Mbps and 20.64 Mbps for each test. However, from their Amsterdam data center, a consistent throughput of 196.8 Mbps was measured to Singapore, and between Amsterdam and the university a consistent 89.6 Mbps.

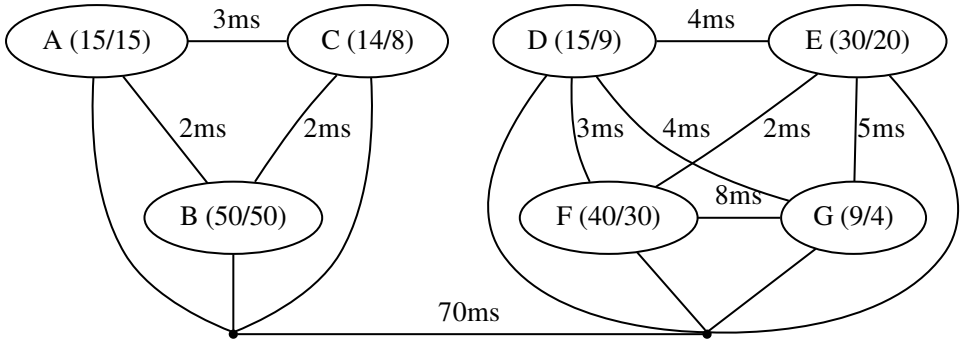
deliver service under hard conditions in larger conversations are likely to ace hard conversations between two people, but the inverse is not necessarily true. No data was found on common inter-node latencies in conversations, thus we have no data to support whether the latencies in the test cases are realistic. If services start monitoring these metrics and combine them with bitrate analysis, they can extract under-performing conversation types based on QoS and add those conversations as future test cases.



(a) Test case “traveller”.



(b) Test case “standup”



(c) Test case “friends”

Figure 3.1: The different test cases. A node with (x/y) indicates x Mbps downlink and y Mbps uplink capacity.

Chapter 4
Experiments

In this chapter we'll benchmark a WebRTC-based video conferencing solution with our test cases, to get a sense of how a peer-to-peer architecture performs. The results from these experiments will be used as motivation for the approach outlined in chapter 5.

4.1 Test Setup

To benchmark appear.in, our WebRTC-based video conferencing solution of choice, we have utilized a small cluster of desktop computers with web cameras, running the most recent versions of Mozilla Firefox[1] and Google Chrome[2]. These two browsers were chosen since they collectively represent 85% of the browser market (according to both appear.in data as seen in Appendix E and the W3C [Con15]), and are powered by two different underlying engines. The goal of the benchmark is to get a sense of how the browsers – and by extension, appear.in – performs with regard to latency and bandwidth usage in our different test scenarios, and to observe how resources are shared among the nodes in a conversation.

Since the test covers two different browsers which do not share a common API (more on this later), measurements were done in two different ways. For Firefox, which do not expose timing data of WebRTC-streams, a browser-external way of measuring end-to-end latencies was necessary. This was achieved by synchronizing all the clocks in the cluster to the same Network Time Protocol (NTP) server, and another independent node – also synced to the same time server – was set to run a timer. Each of the nodes in the cluster filmed this timer, and with the same timer running locally in a terminal, the end-to-end latency could be extracted by taking regular screenshots, and finding the difference between the local timer and the timer

[1] Version 36.0.4, latest version as of 2015.06.05 when the tests were run
[2] Version 41.0.2272.101, latest version as of 2015.06.11 when the tests were run

Figure 4.1: An example screenshot from a Firefox test run on node A. The nodes are, from top left and clockwise, A, C, B and D. We can see how the ≈33 ms refresh rate manifests itself, as the visible times are .928, .959, .990, and 0.021 (node A, barely visible behind the .990).

as sent by the other nodes. See Figure 4.1 for an example of how the screenshots looked. The script that ran this sequence can be found in Appendix B.

Bandwidth usage was measured by running `tcpdump` throughout the test run, and bitrates between each pair of nodes was extracted with `tshark`.

For the Chrome tests, this was a bit simpler and less manual, as Chrome provides both timing data and bitrates through the `getStats` API. Firefox also supports `getStats`, but does not include timing data, even though the data is assumed to be available internally in the browser. Data was extracted from Chrome using the scripts included in Appendix D, and submitted to an external server collecting the data from all nodes.

4.1.1 Sampling

At the start of the test, the nodes join the conversation in alphabetical order (the node names are the letters A-G), as soon as the previous node has established connection to all the other parties already in the conversation. Preferably the join order would be random and the results averaged over several test runs, but due to time constraints this was not possible.

When all nodes have established bi-directional connections, the conversation was left running for a minute, before sampling started. This was done to allow some time to reach a stable state.

For Firefox, where the interpretation of the results is a very tedious and laboursome process, samples were taken every ≈12 s.[3] The last six samples for each node was interpreted and stored, yielding a total sample time of ≈80 s. On Chrome, where there's no interpretation step, samples were submitted every second. The sample time was two minutes, yielding 120 samples for each node.

For all test cases the test was first run without any traffic shaping applied, so to see that the browsers behave as expected in an unconstrained setting. The full data set from these tests are included in Appendix G. Both browsers behaved as expected, which helps validate that the results presented here are regressions because of the constraints applied, and not CPU or other factors not controlled in the experiment.

4.1.2 getStats

The relevant values offered by the `getStats`-API on Chrome[4] and Firefox is presented in Table 4.1 and Table 4.2. The values reported here is what's returned by the browser.

Table 4.1: Incoming video data

Chrome	**Firefox**
bytesReceived:`str/int`	
packetsLost:`str/int`	
packetsReceived:`str/int`	
googCurrentDelayMs:`str`	jitter:`float`
googDecodeMs:`str`	mozRtt:`int`
googJitterBufferMs:`str`	
googMaxDecodeMs:`str`	
googMinPlayoutDelayMs:`str`	
googRenderDelayMs:`str`	
googTargetDelayMs:`str`	

Table 4.2: Outgoing video data

[3] A bit variable, as it's 10 s + the delay for taking and storing the screenshot.

[4] Documentation is very poor for the `getStats`-API as the specification is not completed yet, therefore the most reliable reference is the source: https://chromium.googlesource.com/external/webrtc/+/master/talk/app/webrtc/statstypes.cc

Chrome	**Firefox**
bytesSent:`str/int`	
packetsSent:`str/int`	
googAvgEncodeMs:`str`	bitrateMean:`float`
googCaptureJitterMs:`str`	bitrateStdDev:`float`
googCaptureQueueDelayMsPerS:`str`	droppedFrames:`int`
googCodecName:`str`	framerateMean:`float`
googBandwidthLimitedResolution:`str`	framerateStdDev:`float`
googCpuLimitedResolution:`str`	
googViewLimitedResolution:`str`	
googRtt:`str`	
packetsLost:`str`	

It's sad to see that all values are casted to strings in Chrome. This is not the case on Firefox, where appropriate types are used. As we also see, all of the timing-related values we're interested in are vendor-prefixed on Chrome, which hints to their unspecified nature. Note that both browsers report more data than what is shown here, this is only the data I consider to be relevant for link quality measurements. Chrome is very helpful in providing why resolution is limited[5] (received resolution is present in the full data set), which could be incorporated into more advanced models. When the `getStats` API specification[6] reaches stable in the W3C, I expect most of these differences to disappear. Note that Firefox has the API closest to the proposed specification as of the time of writing.

The values of jitterBufferMs, renderDelayMs, decodeMs and currentDelayMs was summed to get the observed latency. This was based on some trial and error to see what best aligned with the observed latencies using the timer, as outlined in Appendix F, since they are not documented anywhere. A more thorough reading of the source code might reveal a more accurate combination, but there was no time to do this for this thesis.

4.1.3 Constraining Nodes

To configure the cluster according to the different test cases, we utilized the Linux traffic control utility `tc`, which is capable of rate-limiting incoming and outgoing traffic, as well as delaying traffic destined for certain hosts. A small script was

[5] Although it would be preferable to see a single value "limitedResolution", which could be either `false`, `"cpu"`, `"bandwidth"` or `"view"`, to make it a bit less verbose and easier to extend.

[6] http://w3c.github.io/webrtc-stats/

developed to act as a glue layer between a representation of a network and `tc`, making configuration repeatable and easily parametrized. The script is included in Appendix C. The test cases from chapter 3 were serialized into YAML, and the same case definitions could then be used by both the script configuring the nodes, and for the sample solution provided in chapter 5.

Applying a given test case is thus completely independent of the actual network utilized in the test cluster, keeping all intelligence on the nodes themselves. This removed the need for expensive routers or having to customize the application code, thus making the method application-agnostic and applicable to any peer-to-peer solution, not only to appear.in.

4.1.4 Automated Testing?

Ideally, testing would be automated and not require running a graphical environment, to allow tests to be run often and in response to events such as commits. This could be possible using by running a browser in a fake framebuffer like `Xvfb`[7] and faking out a media stream[8]. Both browsers should be able to be tested in such a setting, but data would be limited to what can be extracted through the `getStats`-API as described above. Therefore it is possible to automate this, but was considered out of scope for this thesis.

Chrome runs regular interoperability tests with Firefox[9], but these tests only test that calls can be established, and do not test any network configurations or measure statistics. Integrating the results from this thesis into this test suite is encouraged for more insight into the performance and behavior of WebRTC implementations. The W3C also maintain a test suite for implementations[10], but those only test compatibility with the APIs, and not network behavior.

4.1.5 Caveats

The Firefox method is accurate in the sense that latencies observed are the actual end-to-end latencies that users would observe, but the precision of the timing values observed is not on the millisecond level we'd prefer. This is due to a number of factors, most notably the refresh rate of the screen running the timer and the frame rate of the video, limiting the precision to $1s/60 \approx 17ms$ and $1s/30 \approx 33ms$ respectively. However, we can surpass this precision by averaging several samples taken during the

[7] http://www.x.org/releases/X11R7.6/doc/man/man1/Xvfb.1.xhtml

[8] Chrome: `--use-fake-device-for-media-stream`, Firefox: `getUserMedia({fake: true, <...>})`. More info about this approach can be found at http://images.tmcnet.com/expo/webrtc-conference/presentations/san-jose-14/D3-2_Testing_v2_2.pdf

[9] Google blogged about this: http://googletesting.blogspot.se/2014/09/chrome-firefox-webrtc-interop-test-pt-2.html

[10] http://www.webrtc.org/testing/w3c-conformance-tests

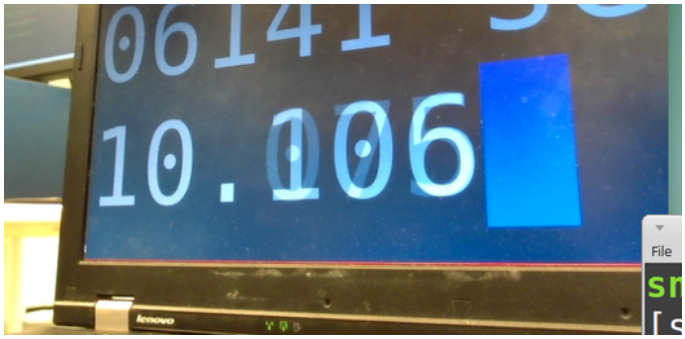


Figure 4.2: A screenshot where a node has sent two overlaying timestamps. In this case interpreted as 10.106, which is reasonable as it's close to the expected ≈33 ms increase from the previous 10.075.

test run, which is why we take six screenshots for each case. The standard deviation of the measurements is reported in the graphs included later in this chapter, which should give some indication towards how accurate the average is. The sample size is very small and should thus be taken with a grain of salt, but it was the best option available at the time.

Taking several samples to improve accuracy leads us to another weakness, which is the manual interpretation of the screenshots. Due to the frequency-related issues discussed above, many of the images include timestamps that are blurred, as the camera captured two underlying screen updates in the same frame, as shown in Figure 4.2.

In general for these cases, the recorded timestamp was consistently interpreted to be the latest of what could be distinguished in the screenshot.

Even assuming that the timestamps are comprehensible and fairly accurate, there's still a possibility of human error when many numbers has to be recorded in this way. To minimize the risk of any mistyped numbers making it into the dataset, any observation outside 1.5 standard deviations of the mean (a range which should include 87% of the numbers observed) was re-interpreted to verify. There's still a chance of smaller errors making it into the dataset, but we assume that these are small enough and distributed evenly among the nodes to not significantly influence any conclusions drawn.

As not enough cameras of any single model was available for the experiments, two different models[11] had to be used. These had slightly different performance

[11]HP Webcam HD-4110 and Tandberg (now Cisco) PrecisionHD

characteristics; the cameras were put side by side with a timer and showed a mean difference of 39.6 ms, but with a relatively large standard deviation of 19.5 ms. As the same effects related to refresh rates as discussed above applies, all samples were at a 30 ms or 60 ms difference of each other.[12] As the difference was assumed to be normally distributed, the mean was simply added to all measurements from the slower camera model to compensate.

For measuring bandwidth utilization between nodes, our method of using `tcpdump` is not entirely satisfactory, as there's no way to report actual *consumed* bandwidth by the application. This is because the traffic control features of the Linux kernel lies above `libpcap`, the library that performs packet capture for `tcpdump` in the network stack. Effectively this means that any incoming bandwidth reported by `libpcap` will be before the rate limiting performed by `tc`. Thus, `tcpdump` cannot report the actual bandwidth consumed by application, only what was received by the network interface. Nonetheless, the bandwidth *sent* by each node is what was actually sent by the application, but there's no guarantee that the receiver was capable of consuming it all. This is good enough for us though, as we can aggregate the data sent by all nodes to determine how saturated a given node's network link is.

While the method itself is application-agnostic, configuring nodes the way we do is not suitable for testing other architectures, such as the ones used by Hangouts and Skype. This is unfortunate, as a performance comparison between the different architectures would have been very interesting, but without running a local instance of the architecture under test, there's no way to achieve the inter-node latencies we desire. This follows from observing that if node A sends its video stream to a Google server, there's no way for it to signal to Google that when the stream is broadcast to nodes B and C, B's stream should be delayed by x ms, and the stream to C should be delayed y ms. It's also not possible for B and C to apply this latency on the receiving side, as they'd have to split the incoming stream for Google into separate streams for each of the transmitting nodes, which would require both getting access to the DTLS keys used by the web browser to encrypt the traffic, and being capable of splitting the stream and rejoining it again without interfering with the browser.

For the most accurate comparison of bitrate, it would have been preferable to use the same method for sampling this on both browsers. However, as Firefox was incapable of delivering timing data, the `getStats` API was discarded entirely, even though it could have been used to sample bitrates as observed by the application. This is unfortunate, but the tools left behind by these experiments allow others who want to repeat the tests to not do this mistake.

[12] Out of 20 samples, 1 was 0 ms, 13 were 30 ms, 5 were 60 ms, and 1 was 90 ms. Which really means that the sample is in the range of 0–29 ms difference, 30–60 ms difference, and so on.

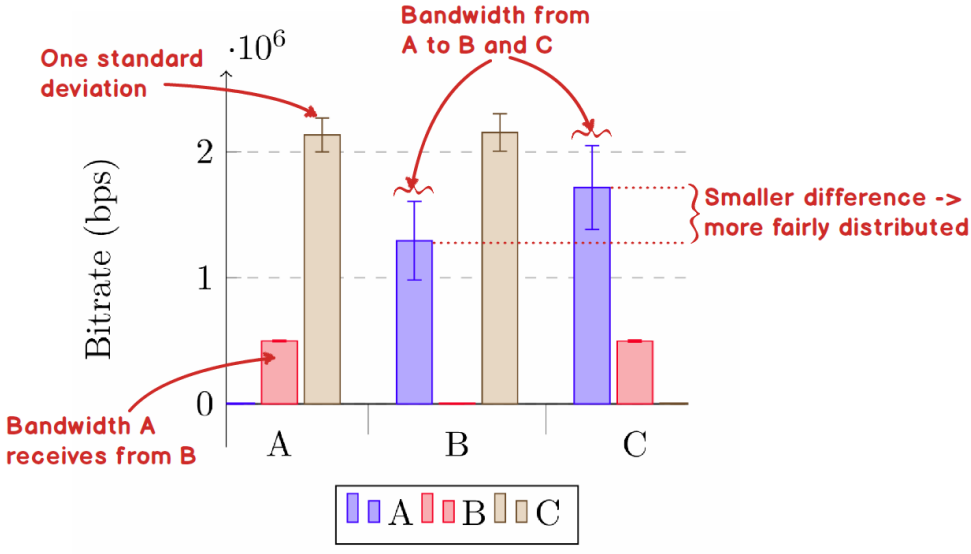


Figure 4.3: How to read bandwidth graphs. Latency graphs are similar, only with different units.

4.2 Results

4.2.1 How To Read the Graphs

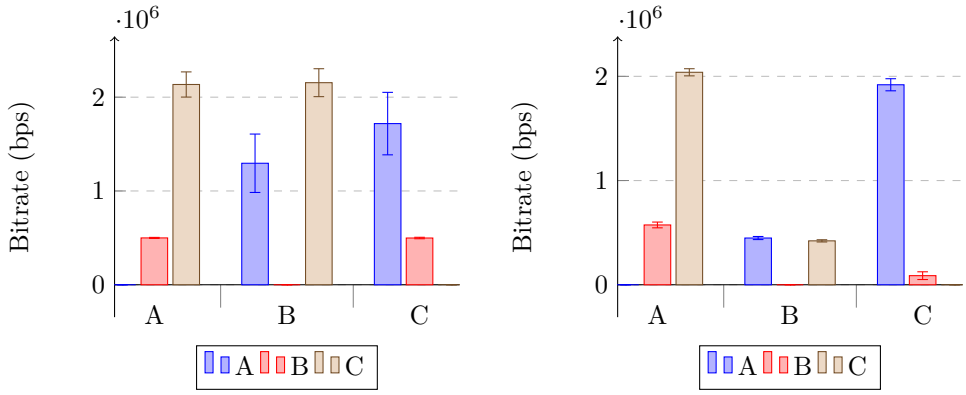
As there will be a lot of graphs in this chapter, a good understanding of how to read them is essential. Figure 4.3 gives a quick primer.

For latency graphs, the lower the observed latency the better. For bandwidth the opposite applies; the more the better. However, this should be seen in context to how widely distributed a node's bandwidth is. If the node does not evenly distribute it's available resources when neither itself nor the peer is constrained by bandwidth, it has failed to reach an even, stable state.

Before we embark on the test cases, we put our two sampling methods up against each other, to see whether the results are comparable. The results were fairly equal across the board, and considered good enough to indicate any serious performance discrepancies. The results are included in Appendix F.

4.2.2 Test Case "Traveller"

A quick recap of the bandwidth limits put on the nodes in the "traveller" test case (read X (7/3) as node X having 7 Mbps downlink and 3 Mbps uplink): A (10/5), B (2/1), C (10/8).



(a) Firefox (b) Chrome

Figure 4.4: Observed bitrates in the "traveller" test case

Figure 4.4 shows the bitrates flowing between the nodes in the "traveller" test case. As was mentioned earlier; the values reported here are only what was received at the node's interface, not what the application consumed. This is important, as it seems from the bandwidths alone that all nodes are doing fairly well in the Firefox case, but look closer. Node B only has a 2 Mbps downlink, but is sent more than 3 Mbps of data. Thus its link is completely saturated, which is reflected in the latencies in Figure 4.5. We can also see that node B is saturating its own uplink as well, which also has a grave impact on the latencies.

Chrome balances this out much better, where A and C communicate unhindered by the constraints of node B (like the Firefox case), but also respect B's constraints and only send what it's capable of receiving. Thus, B's downlink has 43% utilization, and likewise 66% for the uplink. The full link utilization data is given in Table 4.3.

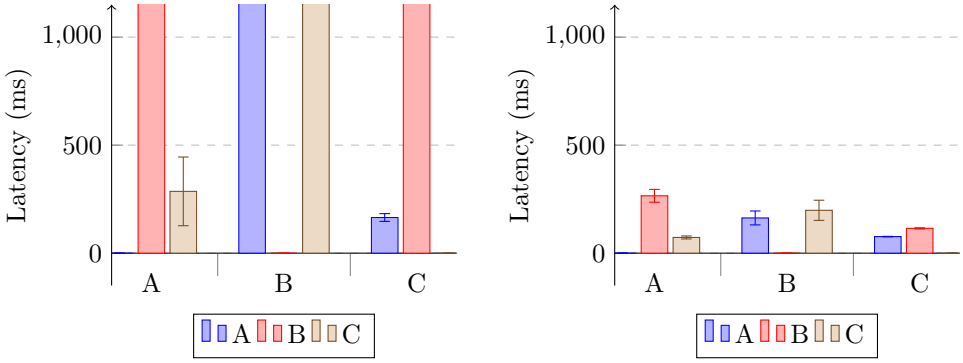
Table 4.3: Link utilization in the "traveller" test case

Firefox

Node	Downlink	Uplink
A	26	60
B	100	100
C	22	54

Chrome

Node	Downlink	Uplink
A	26	47
B	43	66
C	20	31



(a) Firefox (b) Chrome

Figure 4.5: Observed latencies in the "traveller" test case. Actual values for the out-of-bounds values in Firefox, from left to right: 26s, 48s, 48s, 23s.

4.2.3 Test Case "Standup"

Quick refresher on "standup" bandwidth limits: A (30/20), B (30/15), C (8/6) and D (6/3).

The key challenge in this case is node D, with only 6 Mbps available on the downlink, slightly upped by node C with 8 Mbps. Observed bitrates from the test are given in Figure 4.6. Firefox displays much of the same behavior we saw in the "traveller" test case; Node C doesn't have any troubles in this test, but node D is completely saturated. Node D receives 2.1 Mbps from each of the other three nodes, which again destroys the latencies in the conversation. Even though node D sends to its fullest capacity, hardly anything of this is correctly received by the other nodes. This probably implies that among the data Firefox is actually putting onto the wire, not enough of it reaches the destinations unfragmented, and thus the receiver is incapable of reconstructing a complete frame to show to the user. Node C doesn't entirely saturate it's uplink however, so there's obviously *some way* streams are limited in Firefox, but it's clearly not adequate.

Chrome handles the two challenged nodes elegantly, with 61/76% downlink/uplink utilization on node C, and 85/80% utilization on node D. The complete link utilization results are given in Table 4.4.

Table 4.4: Link utilization in the “friends” test case

Firefox			Chrome		
Node	**Downlink**	**Uplink**	**Node**	**Downlink**	**Uplink**
A	16	32	A	13	31
B	17	42	B	16	38
C	64	97	C	61	76
D	100	100	D	85	80

Latencies are depicted in Figure 4.7. While Chrome generally performs okay, we see that even though nodes C and D are not saturating their connections, they observe latencies which are significantly more delayed compared to the two other nodes. Both C and D would have a noticeable delay in this test case. On Firefox only nodes A to C can communicate.

4.2.4 Test Case “Friends”

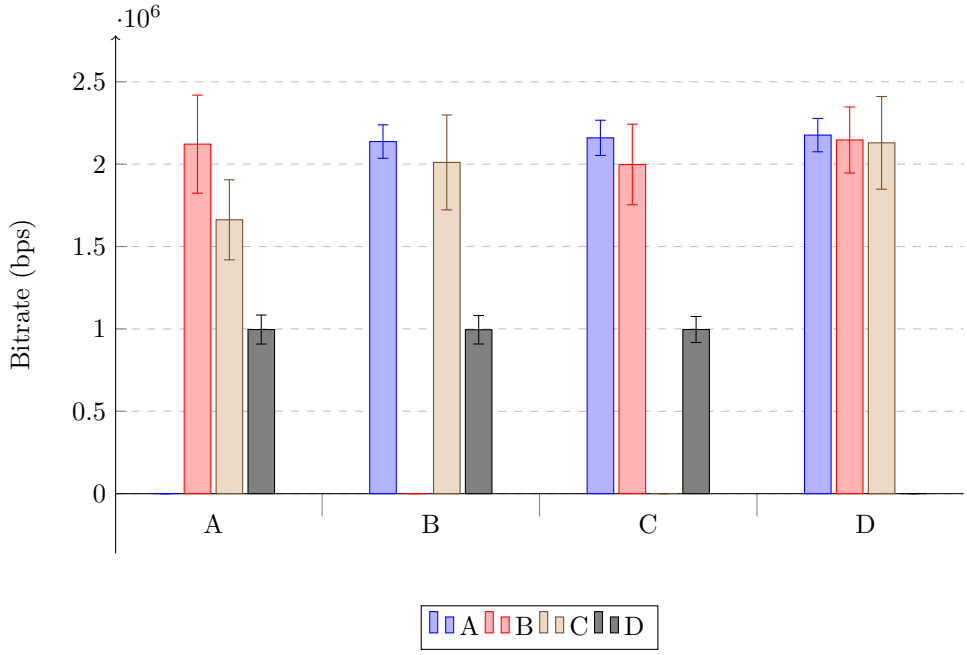
Quick refresher of the “friends” test case; there’s two groups (A–C and D–G), with high latency between the groups, and the following bandwidth limits: A (15/15), B (50/50), C (14/8), D (15/9), E (30/20), F (40/30), and G (9/4).

Figure 4.8 shows that for the most resource-constrained nodes, Firefox – not unexpectedly – completely saturates the links. Both C and G have a fully saturated uplink. G is the only node that also has a saturated downlink, and again we see the effects this has on the latencies in Figure 4.10.

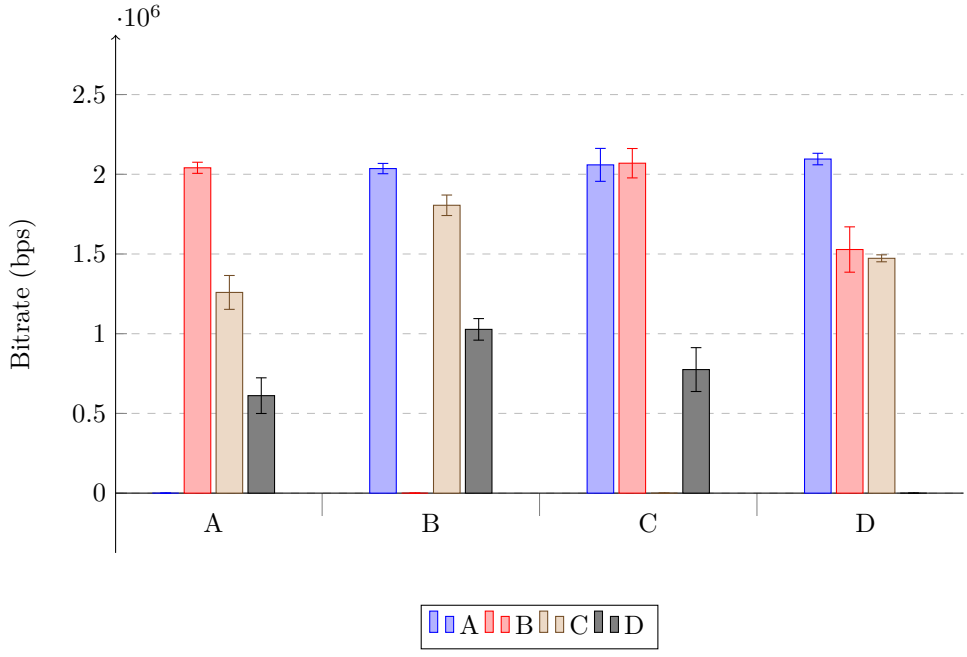
The link utilizations are given in Table 4.5.

Table 4.5: Link utilization in the “friends” test case

Firefox			Chrome		
Node	**Downlink**	**Uplink**	**Node**	**Downlink**	**Uplink**
A	57	83	A	57	81
B	18	25	B	15	24
C	75	99	C	72	83
D	66	100	D	68	84
E	33	49	E	34	56
F	22	41	F	23	37
G	100	99	G	90	75

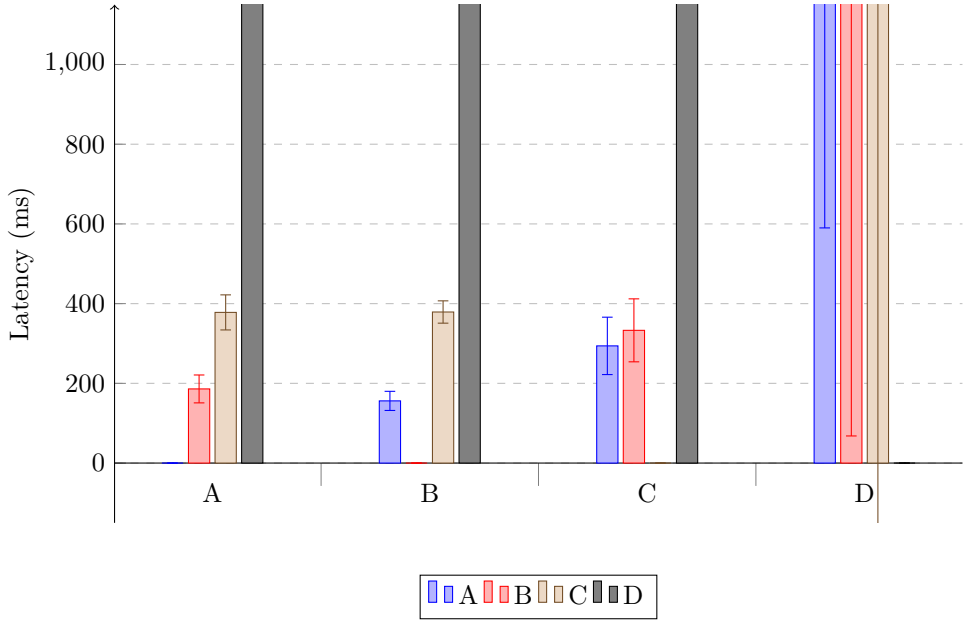


(a) Firefox

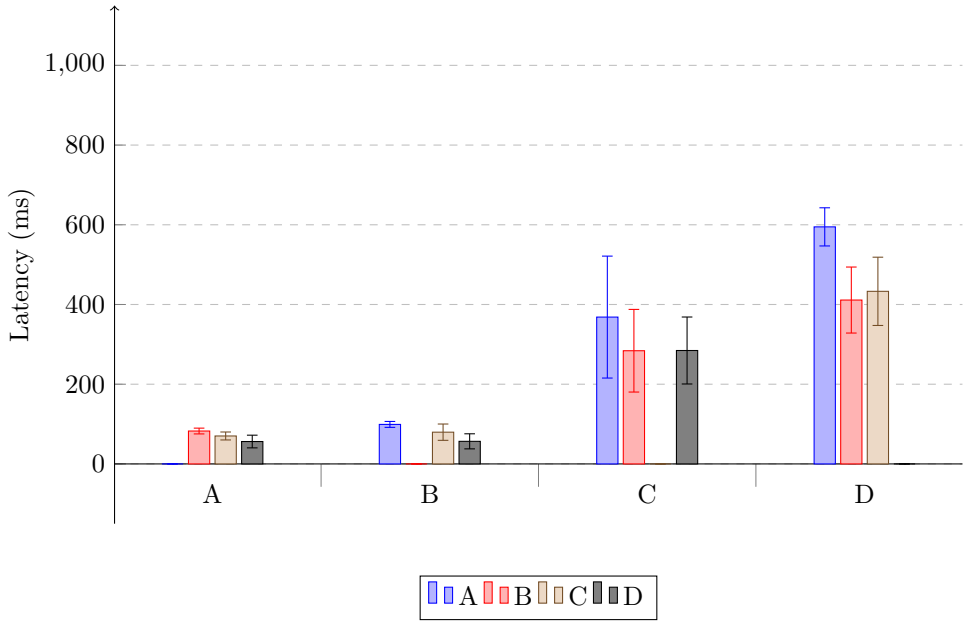


(b) Chrome

Figure 4.6: Bitrates in the “standup” test case.



(a) Firefox



(b) Chrome

Figure 4.7: Observed latencies in the "standup" test case. Firefox out-of-bounds values are, left to right: 4.8 s, 3.5 s, 7.8 s, and > 11 s for everything in to D.

On Chrome, bandwidth is not distributed evenly among its peers. We see this in the data for node G. Of it's 4 Mbps uplink, we see that node E receives a little more than 1 Mbps of this, while F gets around 400 kbps. Node C and D both receive ≈350 kbps, while A gets a full 720 kbps. B are left with the scraps that remain, at ≈130 kbps.

We observe something similar for nodes C and D, with their 8 Mbps and 9 Mbps uplinks, respectively. Node C is well received by most of the nodes in the other group (D, E and F), which all get more than 1.5 Mbps. The local nodes A and B however, get only 1 Mbps and 450 kbps – which shows that having low latencies *does not* seem to significantly influence allocated bitrate. Granted, this dataset is only from one single test run, more data is needed to say anything conclusively about whether this is consistent behavior, but the data seem to imply that even a minute is not enough to reach fairness for Chrome.

Node D repeats much of what we saw in node C, where the remote nodes all get more than the local ones. This might be due to D being the first of the second group into the conversation, establishing connections with the remote nodes before any of the other local nodes are present. Thus when the other nodes in D's group joins, they get to share whatever capacity D has left. How the distribution evolves with time was not studied in this thesis, but might provide insight into how long it would take to reach fairness.

In any case, if it takes more than 10-30 seconds to establish fairness, this author considers it likely that the users will leave the platform and not wait for stuff to smoothen out, at least if video is of any importance in the conversation. Audio will not be hit as hard by uneven distribution, but if your goal as a service provider is to deliver video conversations, video quality and quick connection times will be central to how you're compared to other providers.

Uneven uplink distribution is not only bad for fairness in the conversation, but also for CPU and battery consumption. We can assume node G's video is encoded at least three times, possibly four in this test case[13], even though all of the nodes have spare downlink capacity for sharing one ≈600 kbps stream (4 Mbps/6).

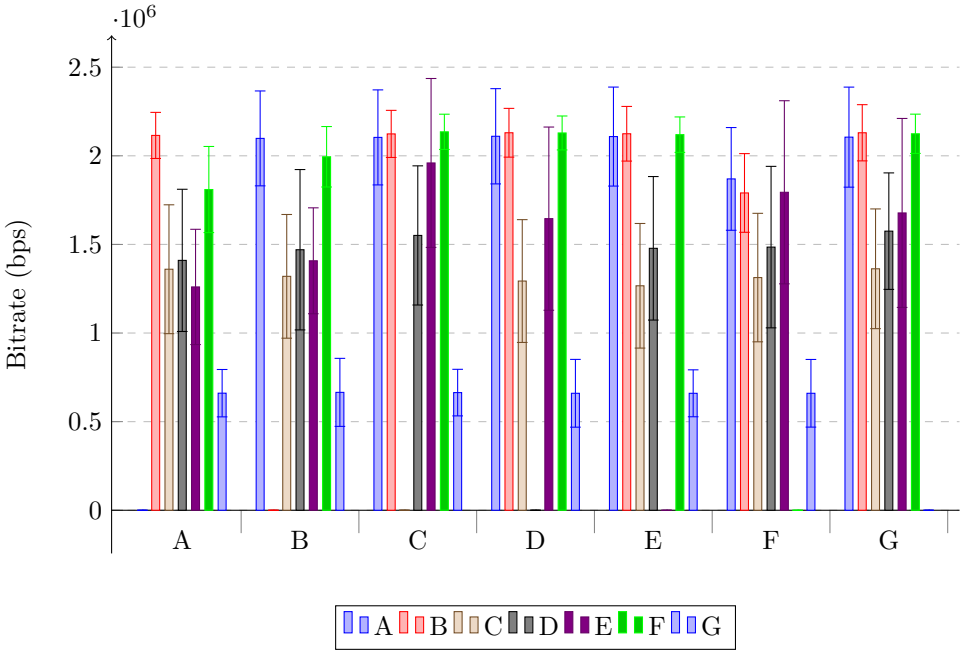
As there's a lot of data points with wildly varying magnitude for Firefox in this case, the latency results have been split in two; one logarithmic view giving a rough overview of how the nodes performed (Figure 4.9), and one cropped view, where only edges with latency less than 500 ms is included (Figure 4.10). As we can see from the linear chart, there's only four nodes that observe latencies below 500 ms, and not even all of those can reasonably be expected to be able to hold a conversation.

[13] A and B could have shared the same stream, C, D and F could have shared a stream, and E has a stream of its own.

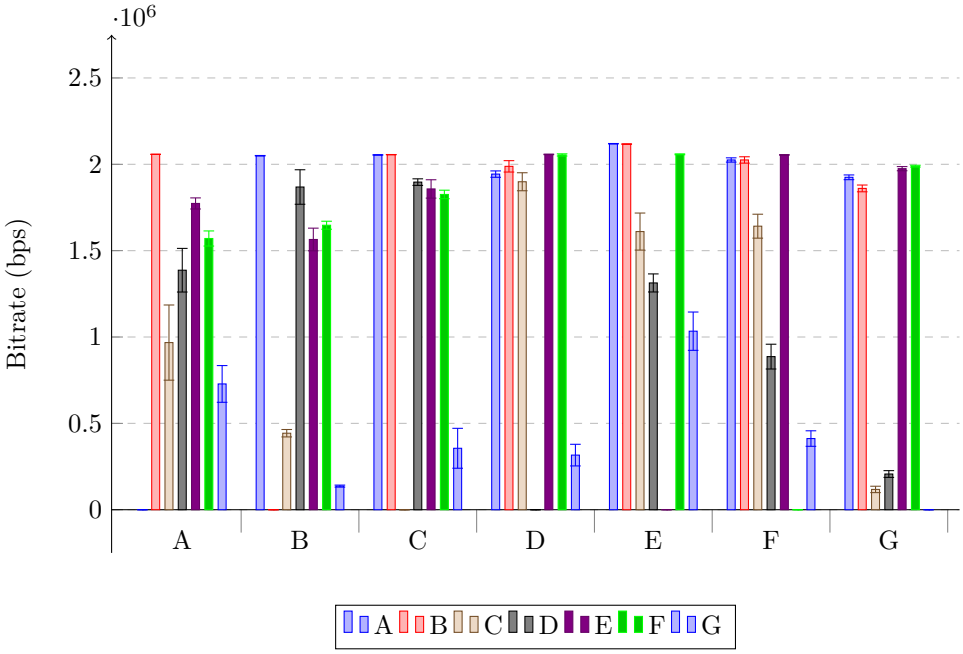
A and B can talk together; A and E can talk, although a little more strained with latencies around 400 ms. B and E can not talk, as B doesn't receive E's stream in reasonable time; B and F however *can* talk together.

To summarize, out of 42 pairs of nodes in the test case, only three of them are able to communicate bidirectionally with Firefox. In practice, this is a conversation all parties abandon immediately.

The latency results on Chrome reinforce the impression we got from the "standup" test case, that nodes with severely constrained connections will also experience much more severe latencies. In this case, nodes C, D and G will all experience noticeable latency. We also note that node C, which favored the remote nodes with it's bandwidth, experiences significantly higher latencies from the two other nodes in its own group than from the remote ones.



(a) Firefox



(b) Chrome

Figure 4.8: Bitrates for test case “friends”

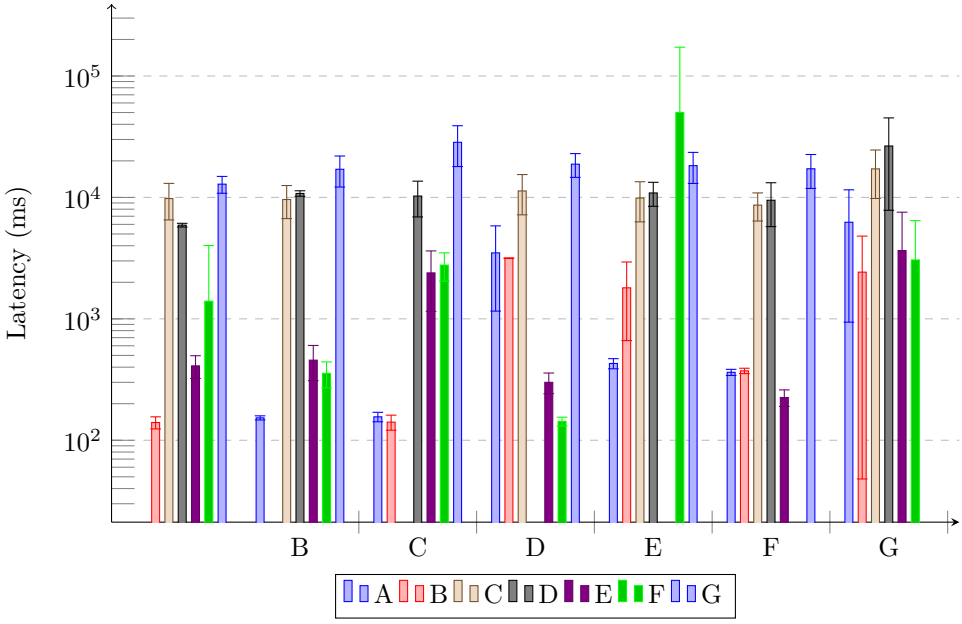
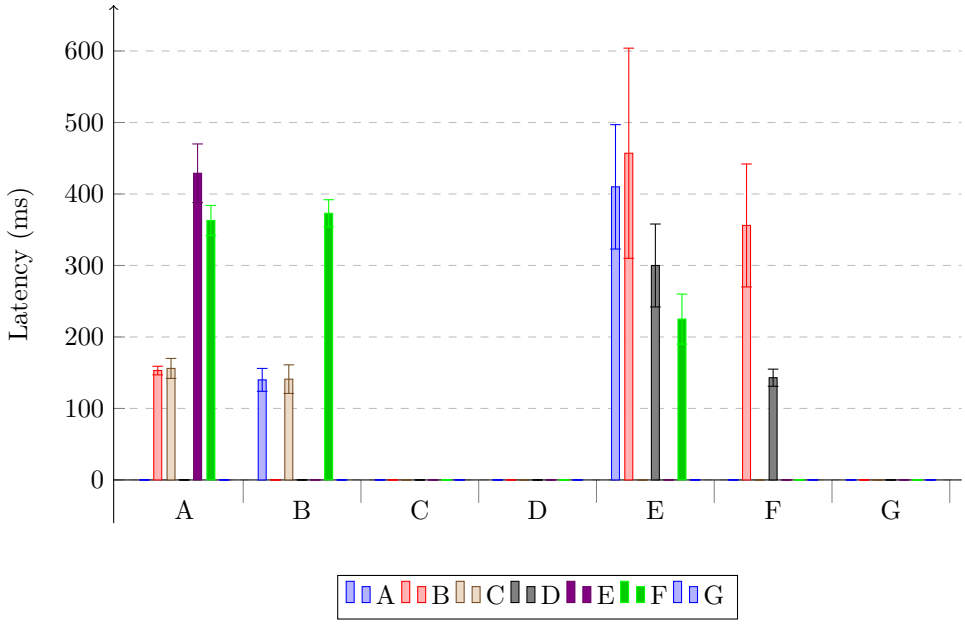
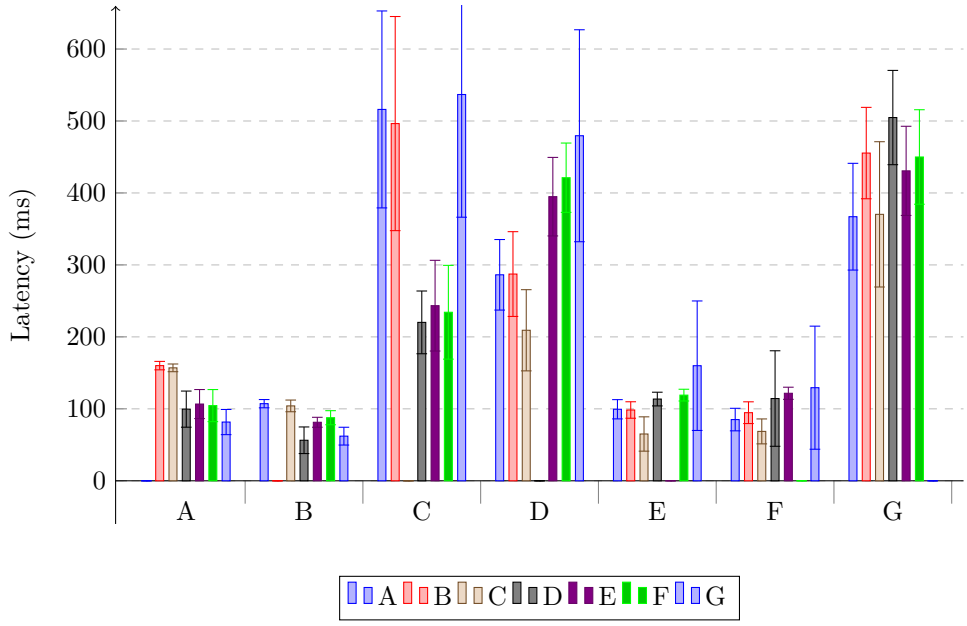


Figure 4.9: Observed latencies for the “friends” test case in Firefox, log scale



(a) Firefox, only sub-500 ms values



(b) Chrome

Figure 4.10: Observed latencies in the “friends” test case

Chapter 5
Analytical Study

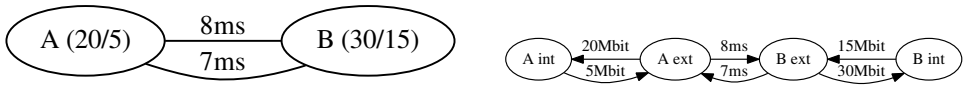
In this chapter we'll walk through how the problem can be modelled as a multi-commodity flow problem and solved using linear programming.

5.1 Modelling

With some prior knowledge of graph algorithms, the first stab at solving the problem might be to look at our sample topologies and see a certain similarity to a max-flow problem – edges have capacities, there's a set of sources, a set of sinks, and we want to maximize *something*. However, there's a couple of things that make it hard to solve directly as a traditional flow problem, particularly that we don't have a single source and a single sink, but lots of them. Trying to model it as a single-source, single-sink problem quickly leads us to discover the limitations of that model, where we realize that if we actually solved it, how would we be able to tell which node is generating flow on a given edge? Clearly, we need a way to distinguish node A's video from node B's video. And we need to make sure that all nodes receive video from every other node, not just maximum bitrate of *any* video.

If we change perspective slightly, we see that this is not a single flow problem, but a *series* of flow problems, sharing an underlying constrained resource. There's the problem of routing video from A to all other nodes, there's the problem of routing video from B to all other nodes, etc. In a conversation with n participants, we now have n separate flow problems, which all share the same resource. However, also this model is too limiting for our use case, as video is sent at a given bitrate, and this stream can neither be split at a given node without incurring a cost, nor does it make sense to add it; two 2 Mbps videos cannot be joined to form a 4 Mbps video. If you put enough restrictions on your nodes and edges it's probable that you might be able to able to prevent this from happening, but there's another way.

This time around, imagine that we have a separate flow problem for each *pair* of nodes; we have one problem routing node A's video to node B, we have one problem



(a) Original problem (b) Problem after node splitting

Figure 5.1: How we can split nodes into external and internal pairs

routing node A's video to node C, etc. This yields a total of $n(n-1)$ problems, which is $O(n^2)$. Modelling the problem this way means we don't have to add any supersinks to connect the targets; each node can act as sink for the video stream – hereafter known as the *commodity* – destined for itself.

One other thing we notice from our initial graph compared to traditional flow problems is that the former have *node constraints*, while flow problems only work with *edge constraints*. To accommodate this, we split each node into two parts, hereafter called the external and internal part, as illustrated in Figure 5.1.

As we now have a well-specified way to go from a given conversation to a graph that can be solved by max-flow algorithms, the entire problem can be solved by joining all the different subproblems under the same resource constraints, and solve as a multi-commodity max-flow problem. This class of problems can be solved with Linear Programming (LP) [AMO88], which can be summarized in canonical form as in Equation 5.1.

$$\begin{aligned} \text{maximize} \quad & \mathbf{c^T x} \\ \text{subject to} \quad & A\mathbf{x} \leq \mathbf{b} \\ \text{and} \quad & \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{5.1}$$

The vector $\mathbf{x}$ is the variables we're trying to find a solution for, each entry in the vector defines a flow on a given edge in the graph for a given commodity. The vector $\mathbf{c}$ scores each flow to determine how much it influences the objective (hereafter known as the *gain*), while the matrix A and the vector $\mathbf{b}$ is the set of constraints put on the system, like staying below bandwidth, flow conservation among non-source and non-sink nodes, and so on.

As LP is a well-known and very general technique that's effective to a vast collection of problems, there are lots of LP-solvers freely available[1]. This simplifies

[1]https://en.wikipedia.org/wiki/Linear_programming#Solvers_and_scripting_.28programming.29_languages

building a solution on top of LP; if you can model your problem as a linear program, it can be efficiently solved by existing, well-tested code. For our experimental solution I chose a solver somewhat arbitrarily among those which had Python bindings, and picked GNU Linear Programming Kit (GLPK).

5.2 Objective

To approach the problem as an LP, we need to have a well-defined objective; the function the LP will try to maximize. To formulate the objective, we need to understand how several performance parameters of the system affect the QoSE.

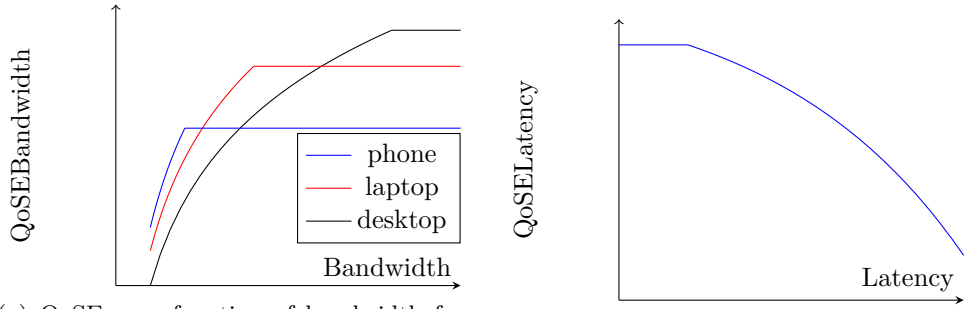
We can start by decomposing the problem, by first looking at how to measure the QoSE from a single node's perspective to one other peer. We then aggregate all of the peers to form the full QoSE as observed by that node, and aggregate that for all nodes in the conversation. Maximizing this value gives a natural even distribution due to the WFL and the logarithmic nature of the QoSE. If the QoSE was a linear function of performance, a small set of nodes could devour all the resources, and the QoSE would have been the same as if the resources were divided evenly. However, the logarithmic shape of the QoSE ensures that there's diminishing returns for resource consumption, thus when one node is receiving 2 Mbps, and one other node is receiving 1 Mbps, increasing the latter from 1 Mbps to 2 Mbps has a greater impact on the QoSE than increasing the former from 2 Mbps to 3 Mbps.

As this thesis only considers bandwidth and latency, the QoSE for a single node to a single peer will be a function of these two variables. No research was found on how these two affect the QoSE for video conversations[2], so we'll make some assumptions here, and if further research is done on this topic, the model can be refined by incorporating actual results at this step.

First, assuming that the latency is fixed, we will adopt a QoSE model from the bandwidth as given in Figure 5.2a. The graph will be tuned to the node's device, as different devices have different saturation points for where there's nothing to gain from sending more data. Devices with larger displays and fast connections have higher expectations to the quality of displayed video, thus the QoSE for a given bandwidth will be higher on smaller devices than larger ones.

As for latency, we assume it follows a function somewhat like the one depicted in Figure 5.2b. Given these two functions for QoSE, we combine them by simply adding them together, with a tunable weighting factor w, which says how heavily latency should count into the QoSE. When considering the QoSE as delivered to a

[2]Much research has been done on QoS in relation to Video-on-Demand, but while the bitstream might be similar, the environment and expectations for real-time communication is entirely different from VOD, and thus not considered directly applicable in this setting



(a) QoSE as a function of bandwidth for different devices

(b) QoSE as a function of latency

Figure 5.2: Assumed QoSE on the signal between a single pair of nodes

single node, we also need to subtract the cost of using its connection, as saturating the connection will have grave impact on queuing delays and packet loss. This cost follows from the queuing delay formula, $\frac{\mu}{\mu-\lambda}$, where μ is the capacity of the link and λ is the offered traffic.

Formally, if we consider our flow network as a graph $G = (V, E)$, a_i is node i's device. $x^c_{i,j}$ is the bandwidth of commodity c on the edge (i, j), $l_{i,j}$ is the latency between nodes i and j. $k_{i,j}$ is the commodity from i to j, C is the set of all commodities, $u(i)$ is the uplink capacity of node i and $d(i)$ is its downlink capacity. r is a constant < 1 that determines how heavily excessive link consumption should be tolled. With $QoSEBandwidth$ and $QoSELatency$ as given in Figure 5.2, we can derive the QoSE for the entire conversation as

$$
\begin{aligned}
ULCost(i) = \frac{u_i}{u_i - r\sum_{c\in C}\sum_{j\in V} x^c_{i,j}}, \qquad DLCost(i) = \frac{d_i}{d_i - r\sum_{c\in C}\sum_{j\in V} x^c_{j,i}} \\
QoSEPair(i,j) = QoSEBandwidth(x^{k_{j,i}}_{j,i}, a_i) + w \times QoSELatency(l_{j,i}) \\
QoSENode(i) = \sum_{j\in V} QoSEPair(i,j) - ULCost(i) - DLCost(i) \\
QoSE = \sum_{i\in V} QoSENode(i) \qquad (5.2)
\end{aligned}
$$

Equation 5.2 is then the QoSE of the entire graph, which becomes the objective function in our LP. Note that this is an assumption, more research should be conducted into finding the actual relationships involved.

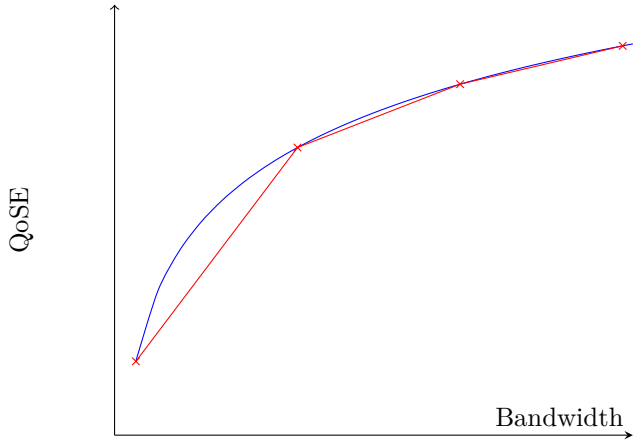


Figure 5.3: QoSE from bandwidth (blue), and a three-part linear approximation (red)

This raises some other issues that has to be handled. The objective function has to be linear to be solved as a linear program, but several of its parameters are non-linear. However, all of these can be approximated arbitrarily close by piecewise linear functions, as illustrated in Figure 5.3. The precision of the approximation can also be parametrized, as either the number of pieces to divide into or as the maximum deviation allowed.

In the following paragraphs we'll discuss how to transform the problem to a linear one that we can solve.

5.2.1 Linear Approximation

Piecewise linear approximations basically try to make a variable subject to different functions in different ranges of its input. As the objective cannot be defined using conditionals or any sort of logic, the approach is a matter of recognizing where a single variable can be replaced by several.

Let's first consider the QoSE from bandwidth. To illustrate how we model this, imagine that instead of node splitting like we did above, we're doing *edge splitting*, replacing a single edge connecting two nodes with a *set* of edges, which can have different capacities and costs. Note that we only have to split the edge coming into the node; there's nothing that affects the objective on the outgoing one. This results in a linear approximation of the actual logarithmic function, as illustrated in Figure 5.3.

Conceptually, we can approach the cost of link utilization in the same way as for

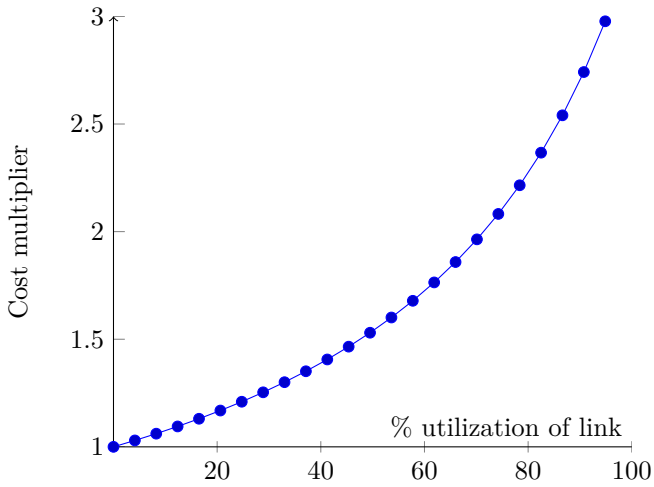


Figure 5.4: How packet delay grows as a function of link utilization for network links. Packet delay for us is equivalent to cost, which thus has to be approximated in an implementation.

bandwidth QoSE by splitting the edges into the node. Thus a 8 Mbps link might be divided into four links of 2 Mbps, with exponentially increasing costs.

As given in the objective function, the link cost can be defined by $\frac{\mu}{(\mu - r\lambda}$, where r is a customizable parameter for how heavily link saturation should be punished. We then partition this function into a small set of piecewise linear intervals, which we can model as parallel edges between two nodes, with different capacities and costs. Keeping this set small limits the number of variables, and thus keeps processing times reasonably low.

Table 5.1: Cost multiplier for link utilization ranges, $r = 0.7$

Link utilization	**Cost multiplier**
0–50%	1.54
51–70%	1.96
71–80%	2.27
81–90%	2.70

To avoid having extra constraints to keep flowing video above a certain minimal threshold, we can replace link capacity with link *slots*. A slot is the size of a minimal unit of video, like 300 kbps. This means that we only need to ensure at least one slot is used between each pair, instead of enforcing more fine-grained constraints.

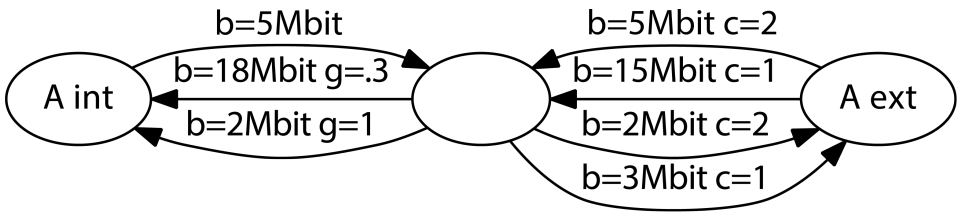


Figure 5.5: An example (20/5) node after two rounds of edge splitting. g is the gain from the bandwidth over an edge, c is the cost. How to split the edges is up to implementations.

Using the partitioning in Figure 5.2.1 as a guide, we can map any number of slots on a physical link into a set of edges E in our graph where $|E| <= 4$. The number of edges to split into determines how close the approximation is, at a cost of more variables in the objective. This can be tuned by implementations.

As there's now two different edge splits on the edge between a node's internal and external parts, we have to insert a temporary node between them to separate the two edge sets. After doing the linear approximations for both edge cost and bandwidth gains, a single node in our flow network ends up like depicted in Figure 5.5.

5.3 Alternative Model

One alternative way to model the problem, is to skip the node splitting step of the previous model, and instead connect nodes directly, but stay below bandwidth by adding constraints for the total sum going out from each node. Which model to choose is – as in every engineering matter – a question of priorities. The first model is a bit harder to comprehend initially, but results in fewer total edges than the latter model does when $n > 3$, as can be seen in Figure 5.6. Edge counts do not matter that much when n is low as finding a solution will occur in trivial time anyway, but the difference might be substantial when n is larger. If performance becomes an issue, more research into different ways to model the problem might yield more efficient solutions.

5.4 Repeaters

This is all well and good, if we solve the maximization problem given earlier we will arrive at an optimal routing of video in any graph. Using this, a browser could immediately establish flows of sustainable bitrate for everyone in the conversation, without the slow creep to steady state that is the case today. However, where

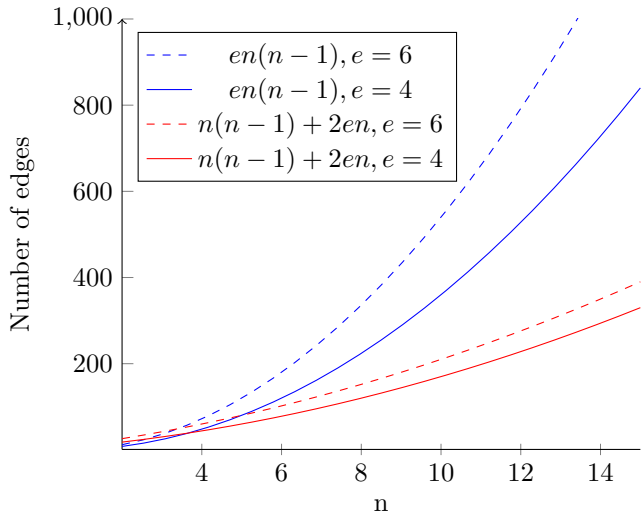


Figure 5.6: How the number of edges in the graph scales for different modelling techniques. The blue graph is the alternative model, red is the suggested one. e is the number of parallel edges to use in the linear approximations, n is the size of the flow network.

today's solutions are topology-bound, we can go beyond those limitations, and use our established technique on slightly modified flow networks.

One thing we note with pure peer-to-peer topologies, is that required bandwidth in and out grows linearly with n, which quickly saturates constrained nodes. Going pure peer-to-peer is usually best in terms of latency, but if the necessary bandwidth is not available, we want to have the option to fall back to relaying some – but not necessarily all – nodes via some unconstrained repeater. This repeater could be provided by the service provider in a data center somewhere, or could be one of the other nodes in the conversation with excess bandwidth available[3]. We have assumed that repeaters are provided in data centers for now, but the constraints can easily be adapted to accommodate nodes as repeaters in a conversation.

Formulating the constraints get a bit weird for repeaters, as they *do not* conserve either bandwidth or commodities. We say that repeaters can *mangle* commodities, which means it can receive the commodity k sent from node A to node B, and can send it as commodity k to B, but it can also send it as commodity m to node C, assuming m is the commodity sent by node A to node C. This is less awkward if the alternative model suggested in section 5.3 is implemented, but there was only time to implement one model for this thesis. This mangling is why we require

[3]These are the nodes Skype calls "supernodes", see [GE14] for a study of supernodes in WebRTC.

each commodity to be received in the objective, it allows receivers to change the commodity to something else than what was sent.

There are some constraints on repeaters though, namely that the incoming amount of one commodity has to be equal to the outgoing amount of all other commodities from the same source. Practically speaking, repeaters can not change the bitrate of incoming video, either up or down. That way, a repeater is a pure IO-bound device, which does not require much else than a fast Internet connection to be operated.

Note that since commodities are routed independently in our flow network, it's possible for a node to use a repeater only for outgoing traffic, but still receive incoming traffic directly from its peers. This will probably often be the case, as many consumer connections are still highly asymmetric in inbound/outbound capacity.

Repeaters are also practical since they're not required to know the contents of the packages it forwards, and thus anyone with spare bandwidth can provide a repeater for anyone to use, without compromising the confidentiality of those conversations.

Deployment wise, repeaters are practically the same as SFUs, thus adding repeaters to a network should not be a challenging task. The biggest difference between repeaters and SFUs is in who controls the sent data; traditional SFUs handle that themselves, negotiating with each peer which stream it should receive based on the peer's capabilities. Repeaters would be instructed by either the sending node or the service provider which streams should be routed where, thus a repeater is a slightly simpler unit.

5.5 Transcoders

Another unit that can be added to the flow network is a transcoder, which is like a repeater in that it has practically unconstrained bandwidth in and out, but also has the capability to transcode incoming video. If node A only has 2 Mbps upload capacity, and node B is a desktop with 10 Mbps downlink, and node C is a phone with only 1 Mbps downlink, A can send its full 2 Mbps video stream to a transcoder, which can then forward the full 2 Mbps stream to node B, but transcode the stream down to a leaner 1 Mbps stream for node C. That way A is able to fully utilize its upload capacity, while the receivers get a stream best utilizing their downlink capacity.

Transcoders are more costly to run than repeaters, due to much heavier compute operations required to transcode video in real-time. They will typically be nodes with specialized hardware for video encoding[4].

[4]The WebM project maintains royalty-free hardware-accelerated designs for encoding VPx: http://www.webmproject.org/hardware/vp9/bige/

Constraint-wise, a transcoder performs the same commodity mangling as repeaters, but can output commodities of arbitrary bitrates less than or equal to the source commodity.

In contrast to repeaters, transcoders have to access the contents of the calls to be able to transcode the signal. This will be the case until a homomorphic cryptosystem is available for video transcoding, which might be in a couple of years or it might be never.

Transcoders are to MCUs what repeaters are to SFUs; slightly simpler since instructions are given on what to do, thus less negotiation with peers.

5.6 Implementation

The sample implementation is in Python and uses GLPK for solving the created linear program. The script makes no attempt to optimize the creation of the linear program, thus it iterates over every single edge dozens of times. Hence, the creation of the linear program is often more expensive than the actual solving of said program. While it's true that there will always be a cost of creating the linear program, since the $O(n^4)$ objective function has to be created along with $O(n^4)$ constraints, the n is usually small enough not to make the total running time unmanageable[5]. If the approach is to be used in situations where n can be large, more work is needed on optimizing this step.

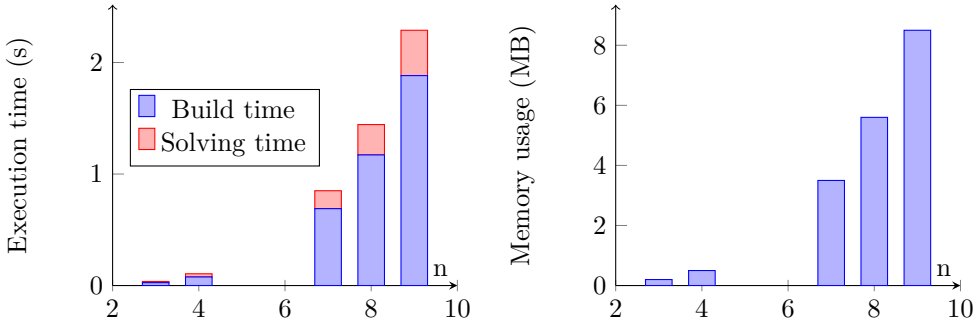
The provided implementation does not fully implement everything discussed here, as time was limited. The script is capable of finding optimal routes for the objective as specified in reasonable time, but does not implement any approximation of the non-linear functions. Thus there's no punishment for maximizing network links, which means all nodes end up with 100% utilization of their uplink, as long as someone has the capacity to receive. The script supports adding repeaters to the mix, routing video through them for nodes that are otherwise incapable of receiving video.

Source code for the implementation is included in Appendix A.

5.7 Scaling

How does the approach scale with the size of the conversation? Our objective function is $O(n^4)$, the same for the number of constraints. Worst-case running time of simplex, the algorithm that powers many LP-solvers, is exponential, but is polynomial in the average case. So how do our cases fare?

[5]Even with 20 nodes, n^4 is only 160,000 operations.



(a) Execution times (b) Memory usage

Figure 5.7: Resource consumption for the sample implementation on desktop computer with Intel i7 860 CPU (four cores, 2.8GHz)

The script was run with a minimal video unit of 400 kbps, which allows even the weakest node in the friends case (node G, with 4 Mbps up) enough slots to stream to all other nodes. While the included script is not feature-complete, it does create test cases close to the correct size, which we can use to give some inclination to expected running time, and how it scales. The tests for 8 and 9 nodes were run by adding one and two extra nodes to the friends test case.

The best fit found for the sample data was a cubic fit,[6] with $R^2 = 0.9995$, indicating that we can expect something close to $O(n^3)$ for this approach. As mentioned earlier, since the n is very small in most of the scenarios, performance is not expected to be an issue. The actual solving step took 0.4 s in the 9 person case, which is not an unreasonable waiting time to establish a connection. The time taken to build the flow network is significantly larger than the solving time in the sample implementation, but since this is highly unoptimized Python it's expected that it can be optimized substantially.

[6]However, the quadratic fit was not much worse, at $R^2 = 0.9974$. The data set is small, more data is needed to say anything conclusively.

Chapter 6
Discussion

In this chapter we'll take a step back, and try to evaluate both the experiments and the suggested solution to see what's necessary to get any further traction.

6.1 Experimental Results

We saw in the experiments that both browsers struggle in heavily constrained environments, but in their own way. Firefox is unable to accommodate constrained nodes at all, while Chrome is very slow at reaching an even distribution of connection resources. Being able to know within seconds of a node joining how best to allocate resources would be of high value.

Our experiments were run in browser-homogeneous environments; all browsers in a conversation were the same. It would be interesting to see how the test cases would have evolved in a more realistic scenario, where some of the users are running Firefox and others are running Chrome. Our "traveller" test case for example, where one node has severely limited bandwidth and higher latencies than the two others, would make an interesting test. How does Firefox act as the limited node, when the two others are Chrome, compared to how Chrome acts as limited, when the two others are Firefox? As we saw in our results, Chrome is much better at negotiating bitrates that don't saturate a node's link, would this also be the case if the sender is Firefox? Results from such a test could indicate where exactly Firefox is failing; whether it's not adhering to signals sent by the limited node, or whether the limited node fails to send to appropriate signals.

These sort of cross-browser tests could also provide tremendous value to the browser implementors if they could be run in an automatic way (like discussed in subsection 4.1.4). When tweaking the implementations, the implementors would be able to run a sanity check towards other browsers and itself in a set of test cases, immediately seeing how the changes affect the performance in different test cases.

6.2 Implementation

Implementing a script demonstrating how the routing works was harder than expected, and not all desired features could be implemented. As every edge in the flow network is a variable, and since almost all will contribute with either cost or gain, you need to keep track of them somehow. The sample script did this with lists within nested maps, which quickly got unwieldy and hard to work with.

Performance did not turn out to be an issue since the data sets in the test cases are quite small. This would also be the case in a production environment, as more than 96% of conversations on appear.in had 5 or fewer participants[1]. Computational overhead is thus largely negligible.

6.3 VAS Fallback

In case a node does not have sufficient bandwidth for receiving $(n-1)$ streams of the minimal bitrate, the routing algorithm will fail. In case this happens, a smart system could fall back to VAS for the most challenged nodes. If this is accomplished, services like appear.in could cover the entire bandwidth spectrum, as was illustrated in Figure 2.1. However, in case the node has sufficient bandwidth for receiving *some* streams, but not all, the user could be presented with an interface that allows prioritizing certain nodes for always being present, while the rest share a VAS-link to the node.

Further fallback is also possible, in the cases where either upload or download is less than the minimal bitrate. If this is the case, the user could be prompted to join the conversation as a voice-only participant, enabling the service to smoothly accommodate all devices with bandwidth larger than 30–40 kbps in both directions. Many services allow the user to opt-in to voice-only, but the service could also detect when video is failing and then automatically switch over, or prompt the user[2]. This step in itself does not require running the full approach, only that the service detects when video fails to deliver at expected quality.

6.4 Finding Network Properties

The proposed approach is heavily reliant on having a fairly accurate representation of the actual capabilities of the network in a given conversation. Some of the properties we're looking for are relatively easy to establish, like latency between the nodes. If the service yields IP-addresses of all the participants when a new user joins, the

[1] Full data in Appendix E
[2] Skype already does this

user can ping those addresses and determine latency to all of them in hundreds of milliseconds.

Bandwidth is notoriously trickier. The biggest problem for measuring bandwidth is that in general it takes a long time, something most users would be very annoyed if they had to do for every conversation. However, the suggested solution does not require you to know your exact bandwidth, and underestimates can work pretty well. Thus a rough test against the provider upon entering a conversation should yield usable results, but this is largely a hypothesis. More research into the effect of accurate bandwidth measurements would be needed to say anything conclusively on the matter.

While tricky, this thesis assumes that efficient ways of gathering the necessary information can be found.

6.5 Who's the Boss

In theory, any node in the conversation could model the problem and solve it to derive where its video should be routed and at what quantity. However, no node can do this without full knowledge of the flow network, which requires everyone to share data with everyone. You'd also need to make sure that everyone in the conversation agrees about the numbers the node came up with, which is hard in a distributed scenario. Since most solutions will be require some central provider to find contacts, it's natural to assume that the service provider can be the negotiator. As the intended application is for WebRTC deployments, this keeps the client light and allows the LP-solver to run where there's most compute capacity available.

This is not a hard requirement however, WebRTC has no dependency on centralization to work. The only thing WebRTC requires is an external channel to communicate session establishment. Practically all WebRTC-based video conversation providers do this over HTTP/WebSockets towards a central server hosted by the provider, but WebRTC could also be utilized without any providers. This could allow the communicating parties to publish connection info over external channels, like SMS, Twitter or mail. The proposed solution could work in such a scenario, perhaps by letting the node with the most compute capacity solve the LP-problem. Such distributed schemes have not been the focus for this thesis, but is an interesting avenue for further research.

6.6 Limitations

The dynamic routing scheme proposed in this thesis is largely independent of underlying technology, but is intended to be built on top of WebRTC. The requirements

to adopt the proposed approach is as follows:

- Clients can discover their own bandwidth
- Clients can discover the latency to each of their peers
- Clients can establish connections with a given bitrate
- Clients can reach each other directly (i.e. known IP addresses)
- Clients can route their video through parties not in the conversation
- Clients can forward received video to other nodes

I believe WebRTC fulfills all of these requirements, but I'm not entirely sure if it's easy to forward video from a client. If that's not possible, the approach still works, but nodes cannot be used as supernodes, thus repeaters have to be provided by the service provider if the limits posed by the peer-to-peer topology is to be overcome.

As long as conversations are modelled as flow networks, any device or network characteristic that can be included in such a model and described as a linear function or approximation can be used to define the objective function. Our goal of maximum bandwidth at minimum latency could also have been minimum latency at minimum server costs, or minimal CPU usage at maximum bandwidth. Any combination is possible.

The proposed approach requires every element in the flow network to be modelled linearly, but I'm not totally certain if that's a hard requirement. I believe it would be possible to solve the problem also in a non-linear way, possibly using algorithms that do not try to be optimal. For the intended application, high precision of the results should not be necessary, thus it might be possible to simplify the model by skipping the linear approximation step in the modelling, and rather find an algorithm that approximates the solution directly.

6.7 Privacy

For privacy-oriented consumers, the suggested solution opens up an interesting opportunity. Many existing solutions like Google Hangouts route every conversation through company-controlled data centers, which in light of the recent NSA revelations[3] is less than stellar from a privacy perspective. Dynamic topologies like discussed in this thesis, combined with a multitude of global VPS providers which provide

[3] I'm referring here to the PRISM program revealed by Edward Snowden [BFGA13]

quick-bootable disk images, enable consumers to provide their own infrastructure that can be used to offload their compute and connectivity to. Combined with WebRTC's mandatory encryption using DTLS, this makes conversations private and not susceptible to surveillance, from companies or governments.

For providers, this could be great news. They would not have to provide expensive infrastructure for call routing, and would rather focus on interfaces for finding friends and other complementary features like text chat, file sharing, call history, contact lists, etc.

This would make the market more volatile, as there would be a lower investment barrier to be able provide a full-blown communications infrastructure. New providers with original ideas could quickly blossom, as users traditionally have not shown a lot of loyalty to most communication platforms[4]. Providing quick-bootable user-controlled server images could be a potential for innovation for companies. It boils down to where the user decides to put their trust, a powerful desktop at home could be used as a relay, or a cloud VPS could provide the same service, granted that you trust the company that provides them – which does not have to be the same company that provides the main service.

6.8 Dynamic Conversations

In our solution, we have so far assumed that most of the properties in the network are static, like the number of nodes, upload/download capacities, latencies, and possibly available CPU, if implemented. In reality however, many of these properties are likely to fluctuate during a conversation, either because the people join and leave conversations, users might be multi-tasking and running other IO-intensive applications on the nodes, or mobile users might have started a conversation over WiFi at home, but started walking to work and thus changed to a cellular connection mid-conversation.

Regularly assessing the state of the conversation should be a natural extension of the system, and should not pose too large of a challenge for implementations. Push-based notifications should also be able to trigger a re-assessment, such as the provider notifying the peers of new nodes joining, to avoid any delay for events that everyone in a conversation should react to immediately. As the algorithm underlying most LP solvers, simplex, is iterative in nature, it might be possible for a solution to a new configuration to start from a previous known good solution, and thus only incorporating the changes to the system to avoid a full re-computation of the

[4]We've seen this several times recently, as users have migrated en masse from SMS to Facebook Messenger to Whatsapp to Snapchat. The only value a user sees in their provider seems to be whether they can reach their friends through them.

topology, but this has not been considered a goal of a system in this first exploration of the idea.

The biggest challenge is ensuring the transitions between topologies become transparent to the user, but as users are more lenient with lagging video than lagging audio, audio could be duplicated on several topologies before video is switched. This could help audio run smoothly during the transition, while video might take a few moments to catch up.

Chapter 7
Conclusions and Future Work

7.1 Concluding Remarks

Traditional peer-to-peer systems require all nodes in a conversation to have upload and download capacity of at least $(n-1) \times b$, where n is the number of users in a conversation, and b the minimal bitrate of a video stream. This requirement grows linearly with the number of users in a conversation, quickly becoming a limiting factor. Other solutions like Google Hangouts route all calls through the service provider's data centers, sacrificing latency for an upload requirement of only b. The approach presented in this thesis merges the best of these properties, allowing users in conversations with sufficient resources to route all their calls in a peer-to-peer to manner, while directing only constrained users through data centers, or through other peers with excess resources.

The approach in this thesis can also improve pure peer-to-peer conversations, by finding an optimal routing of video much faster than what browsers do through trial and error today. Providers adapting this work could instrument the browser directly, allow everyone to establish connections with bandwidth adjusted to fit them from the start without a long period to reach steady state.

Current browsers were shown to not handle constrained environments satisfactorily, either failing entirely or taking a very long time to establish decent connections to new nodes. The approach outlined could enable existing service providers to serve markets that are currently incapable of communicating over peer-to-peer services. Centralized providers could adapt the approach to push high-capacity conversations over peer-to-peer, to reduce the need for servers to manage the traffic.

7.2 Future Work

This thesis only presented a light sample implementation of the approach, which does not conclusively demonstrate that the approach is feasible. Some points below

are listed that remains before this can be established.

7.2.1 Implementation

The system has not been tested in any actual video conversations and does not currently employ all the features described in chapter 5 to achieve a fair routing. Implementing the missing features and testing the solution in actual conversations needs to be done.

As discussed in section 6.8, conversations are unlikely to be static, and will probably in many cases greatly change topology as the conversation progresses. This is inevitable for conversations with more than two participants, as the system will first solve for a $n = 2$ system, and then later have to solve for a $n = 3$ system when the next node joins. The system as proposed in this paper is completely oblivious to the state of a conversation, it simply takes in a configuration of nodes, and outputs an efficient routing of commodities. Adapting the system to support either change-sets to an existing topology, maybe prioritizing not interrupting existing connections when new nodes enter a conversation, or just recomputing from scratch, will require both the provider and the nodes to listen to notifications from others about topology changes. Incorporating this into the system in a transparent way for the user remains to be done in future work.

7.2.2 Other Limiting Factors

Bandwidth and latency were the only limiting factors we included in the sample implementation, but there's a ton of other variables that could influence desired topology, such as CPU, battery, screen size, link packet loss, link jitter, and user profiles ("Only do peer-to-peer, this is a confidential call").

None of these factors were considered for these first tests of a dynamic topology system, but should be considered for future work. Decisions could be based on data, but allow user overrides, such as prompting the user whether to optimize for performance, which will kill the battery in 10 minutes, or slightly degrade performance by routing video through a repeater/transcoder to reduce local CPU consumption to last another 20 minutes. Or just take the decision and silently notify the user. The degree of user autonomy in these cases are left to implementations to decide.

7.2.3 Integrate Test Cases With Browsers

If the entire test suite is made automatic, tests could easily be run by browser vendors to continuously assess how they perform in constrained environments. This also requires that Firefox exposes timing data from the RTCP stream through the `getStats` API, and preferably that the API gets standardized.

References

[AMO88] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. Network flows. Technical report, DTIC Document, 1988.

[BFGA13] Stephen Braun, Anne Flaherty, Jack Gillum, and Matt Apuzzo. Secret to prism program: Even bigger data seizure. *Associated Press*, June 2013. URL: http://bigstory.ap.org/article/secret-prism-success-even-bigger-data-seizure, accessed 2015-06-20.

[Con15] World Wide Web Consortium. *Browser Statistics and Trends*, 2015. URL: http://www.w3schools.com/browsers/browsers_stats.asp, accessed 2015-06-17.

[CS12] Inc. Cisco Systems. *Cisco Video and TelePresence Architecture Design Guide*, March 2012. URL: http://www.cisco.com/c/en/us/td/docs/voice_ip_comm/uc_system/design/guides/videodg/vidguide.pdf.

[GE14] J. Grönberg and Meadows-Jönsson E. *"Tree topology networks in WebRTC: An investigation into the feasibility of supernodes in WebRTC video conferencing"*, 2014.

[Goo] Google. *Start a video call on your computer*. URL: https://support.google.com/hangouts/answer/3110347?hl=en, accessed 2015-06-24.

[Goo12] Dan Goodin. *Skype replaces P2P supernodes with Linux boxes hosted by Microsoft*, March 2012. URL: http://arstechnica.com/business/2012/05/skype-replaces-p2p-supernodes-with-linux-boxes-hosted-by-microsoft/.

[Han14] Philipp Hancke. *How does Hangouts use WebRTC? webrtc-internals analysis*, July 2014. URL: https://webrtchacks.com/hangout-analysis-philipp-hancke/, accessed 2015-06-22.

[LCMP12] Patrick Le Callet, Sebastian Möller, and Andrew (eds) Perkis. *Qualinet White Paper on Definitions of Quality of Experience*. European Network on Quality of Experience in Multimedia Systems and Services (COST Action IC 1003), version 1.2, march 2013 edition, 2012.

[nVi13] nVidia. *NVIDIA Tegra 4 GPU Architecture*, February 2013. URL: http://www.nvidia.com/docs/IO/116757/Tegra_4_GPU_Whitepaper_FINALv2.pdf, accessed 2015-06-22.

[Pro14] The WebM Project. *VP9 Video Codec*, January 2014. URL: http://www.webmproject.org/vp9/, accessed 2015-06-22.

[RESA10] P. Reichl, S. Egger, R. Schatz, and A. Alconzo. The logarithmic nature of qoe and the role of the weber-fechner law in qoe assessment. In *IEEE International Conference on Communications (ICC)*, pages 1–5. IEEE, 2010.

[Sie01] Gerard Sierksma. *Linear and integer programming: theory and practice*. CRC Press, 2001.

[Tho06] Andreas Thomann. *Skype - A Baltic Success Story*, 2006. URL: https://www.credit-suisse.com/no/en/news-and-expertise/news/economy/europe.article.html/article/pwp/news-and-expertise/2006/09/en/skype-a-baltic-success-story.html.

[Vid13] Vidyo. *Vidyo and Google Collaborate to Enhance Video Quality within WebRTC*. August 2013. URL: http://www.vidyo.com/company/news-and-events/press-releases/vidyo-and-google-collaborate-to-enhance-video-quality-within-webrtc/, accessed 2015-06-23.

[Web34] E. H. Weber. *De Pulsu, Resorptione, Auditu Et Tactu. Annotationes Anatomicae Et Physiologicae*. CF Koehler, 1834.

Appendix A
The find_allocations script

This is the Python script[1] introduced in the chapter 5, which computes an efficient video routing for a flow network from a file of cases and a set of parameters.

Necessary dependencies for running the script include the glpk LP solver (available for both Windows and *nix), and the Python bindings provided by the `pulp` library[2].

```
# -*- coding: utf-8 -*-
"""
    Find optimal video routing for a given case.

    :copyright: (c) 2015 by Tarjei Husøy
    :license: MIT, see http://opensource.org/licenses/MIT
"""

from collections import defaultdict, namedtuple
from itertools import chain, product
from pdb import set_trace as trace
from pulp import (LpMinimize, LpMaximize, LpProblem, LpVariable, LpInteger,
    LpSenses, GLPK, value)
import argparse
import io
import logging
import os
import sys
import time
```

[1] Available online at https://github.com/thusoy/hybrid-video-topology/blob/master/tools/find_allocation.py

[2] Install with `pip install pulp`

```
import yaml

logger = logging.getLogger('hytop')
case = None
constraints = []

device_gain = {
    'desktop': 4,
    'laptop': 3,
    'tablet': 2,
    'mobile': 1,
}

class Commodity(int):
    def set_pair(self, sender, receiver):
        self.sender = sender
        self.receiver = receiver


def load_cases(case_file):
    with open(case_file) as fh:
        return yaml.load(fh)

Edge = namedtuple('Edge', ['slots', 'threshold', 'cost'])


def main():
    global case
    default_case_file = os.path.join(os.path.dirname(__file__), 'cases.yml')
    cases = load_cases(default_case_file)
    parser = argparse.ArgumentParser()
    parser.add_argument('-v', '--verbose', help='Logs all constraints added',
        action='store_true', default=False)
    parser.add_argument('-d', '--debug', help='Print debug information',
        action='store_true', default=False)
    parser.add_argument('-s', '--slot-size', default='512kbps',
        help='Set slot size to this size')
    parser.add_argument('-e', '--edges', default=4, type=int,
        help='How many parallell edges to add between each pair of nodes')
    parser.add_argument('-c', '--case', choices=cases.keys(),
        default='traveller')
```

```
    args = parser.parse_args()
    log_level = logging.INFO if args.verbose else logging.WARNING
    logging.basicConfig(level=log_level, stream=sys.stdout)
    case = cases[args.case]
    solve_case(args, number_of_edges=args.edges)


def all_edges(include_self=False):
    for node in nodes():
        for other_node in nodes():
            if other_node != node or include_self:
                yield (node + 'proxy', other_node + 'proxy')
        yield (node, node + 'proxy')
        yield (node + 'proxy', node)
    for proxy in proxies():
        for repeater in repeaters():
            yield (proxy, repeater)
            yield (repeater, proxy)

def get_edges(number_of_slots, number_of_edges=4):
    cutoffs = get_cutoffs(number_of_edges)
    print '%d slots:' % number_of_slots,
    edges = [] # (slots, treshold, cost) tuples
    utilization = 0.0
    utilization_step = 1.0/number_of_slots
    for cutoff in cutoffs:
        slots = int(number_of_slots*(cutoff - utilization))
        utilization += utilization_step*slots
        edges.append(Edge(slots, utilization, cost(utilization)))
    utilized_slots = sum(edge.slots for edge in edges)
    edges.append(Edge(number_of_slots - utilized_slots, 1, cost(1)))
    edges = [edge for edge in edges if edge.slots]
    return edges

def cost(utilization, punishment_factor=0.9):
     # The closer the punishment_factor is to 1, the heavier the punishment
     # for saturating links (balance against cost factor of delay)
    return 1/(1-punishment_factor*utilization)

def get_cutoffs(partitions=4):
    """ Find the cut-off points for partitioning something where each part is
```

```
    twice as large as the previous.

    >>> get_cutoffs(2)
    [0.66]
    """
    number_of_segments = sum(2**i for i in range(partitions))
    segment_size = 1.0/number_of_segments

    # Create an array like [0.066667, 0.1333, 0.26667, 0.53] for partitions=4
    segs = [segment_size*2**i for i in range(partitions)]

    # Reverse the array and make each element a cumulative sum
    cutoffs = [sum(segs[-1:-1-i:-1]) for i in range(1, partitions)]
    return cutoffs

def node_pairs():
    for node in nodes():
        for other_node in nodes():
            if node != other_node:
                yield (node, other_node)


def nodes():
    for node in sorted(case['nodes'].keys()):
        yield node


def commodities():
    commodity_number = 0
    for node in nodes():
        for other_node in nodes():
            if node != other_node:
                commodity = Commodity(commodity_number)
                commodity.set_pair(node, other_node)
                yield commodity
                commodity_number += 1


def commodity_from_nodes(sender, receiver):
    number = 0
    for node in nodes():
```

```
143         for other_node in nodes():
144             if node != other_node:
145                 if node == sender and other_node == receiver:
146                     commodity = Commodity(number)
147                     commodity.set_pair(node, other_node)
148                     return commodity
149                 number += 1
150 
151 
152 def proxies():
153     for node in nodes():
154         yield node + 'proxy'
155 
156 
157 def repeaters():
158     for repeater in case.get('repeaters', {}):
159         yield repeater
160 
161 def add_constraint(constraint, label=None):
162     text = '%s constraint' % label if label else 'Constraint'
163     logger.info('%s: %s', text, constraint)
164     constraints.append(constraint)
165 
166 
167 _debug_vars = []
168 _debug_index = 0
169 def debug():
170     global _debug_index
171     _debug_index += 1
172     _debug_vars.append(LpVariable('__debug__x%d' % _debug_index, lowBound=0))
173     _debug_vars.append(LpVariable('__debug__y%d' % _debug_index, lowBound=0))
174     return _debug_vars[-1] - _debug_vars[-2]
175 
176 def dump_nonzero_variables(prob):
177     print '\n'.join('%s = %s' % (v.name, v.varValue) for v in prob.variables()
178         if v.varValue)
179 
180 
181 def get_edge_latency(node, other_node):
182     """ Get latency of edge between node and other_node. If that is not
183     specified in the case spec, latency from other_node to node will be
```

```
    returned if present.
    """
    if node.startswith('rep'):
        edge_latency = case['repeaters'][node][other_node[0]].split()[0]
    elif other_node.startswith('rep'):
        edge_latency = case['repeaters'][other_node][node[0]].split()[0]
    elif 'proxy' in node and 'proxy' in other_node:
        try:
            edge_latency = case['nodes'][node[0]][other_node[0]].split()[0]
        except:
            # A -> B not defined, lookup B -> A
            edge_latency = case['nodes'][other_node[0]][node[0]].split()[0]
    else:
        # TODO: Add edge cost for parallell edges between proxies and their
        # nodes
        edge_latency = '0ms'
    return int(edge_latency.strip('ms'))


def get_objective(variables, number_of_edges):
    objective = 0
    for node, other_node in node_pairs():
        commodity = commodity_from_nodes(node, other_node)
        other_proxy = other_node + 'proxy'
        edges = get_edges(number_of_edges)
        for edge_num, edge in enumerate(edges):
            # Add bandwidth-gains to objective
            device_class = case['nodes'][other_node]['class']
            gain = device_gain[device_class]
            edge_var = variables[other_proxy][other_node][commodity][edge_num]
            gainconst = 10
            objective += gainconst * gain * edge_var
            # objective -= edge.cost * edge_var
            # TODO: Subtract incoming flow to the source node from objective?


    for commodity, (node, other_node) in product(commodities(), all_edges()):
        # Subtract edge cost from objective
        edge_latency = get_edge_latency(node, other_node)
        for edge in variables[node][other_node][commodity]:
            objective -= edge_latency*edge
```

```
225
226     return objective
227
228
229 def initialize_variables(number_of_edges):
230     variables = defaultdict(lambda: defaultdict(list))
231     # Initialize all edge variables
232     for (node, other_node), commodity in product(all_edges(), commodities()):
233         bandwidth_limited = ('proxy' in node and node[0] == other_node) or (
234             'proxy' in other_node and other_node[0] == node)
235         parallell_edges = number_of_edges if bandwidth_limited else 1
236         for edge in range(parallell_edges):
237             # Assume nothing exceeds gigabit speeds, not even backbone links
238             if edge == 0:
239                 variables[node][other_node].append([])
240             variables[node][other_node][-1].append(LpVariable('%sto%sK%dC%d' %
241                 (node, other_node, commodity, edge), lowBound=0, upBound=1000,
242                 cat=LpInteger))
243     return variables
244
245 def parse_bandwidth_into_slots(bandwidth):
246     slot_size = 400000
247     bandwidth = bandwidth.strip('bit')
248     unit = bandwidth[-1]
249     multipliers = {
250         'G': 10**9,
251         'M': 10**6,
252         'k': 10**3,
253     }
254     multiplier = multipliers[unit]
255     return int(bandwidth[:-1])*multiplier/slot_size
256
257
258 def add_bandwidth_conservation(variables):
259     # Stay below bandwidth (capacities)
260     for node in nodes():
261         downlink_raw = case['nodes'][node]['downlink']
262         downlink_capacity = parse_bandwidth_into_slots(downlink_raw)
263         add_constraint(sum(sum(variables[node+'proxy'][node][commodity]) for
264             commodity in commodities()) <= downlink_capacity)
265         uplink_raw = case['nodes'][node]['uplink']
```

```
266         uplink_capacity = parse_bandwidth_into_slots(uplink_raw)
267         add_constraint(sum(sum(variables[node][node+'proxy'][commodity]) for
268             commodity in commodities()) <= uplink_capacity)
269
270
271 def add_all_commodities_must_be_sent_and_received_constraint(variables):
272     # All commodities must be sent by the correct parties
273     # Note: Node should not need to send a commodity if a repeater does, and
274     # the repeater receives another commodity from this node
275     for commodity in commodities():
276         proxy = commodity.sender + 'proxy'
277         commodity_sources = sum(variables[commodity.sender][proxy][commodity])
278         for repeater, proxy in product(repeaters(), proxies()):
279             commodity_sources += sum(variables[repeater][proxy][commodity])
280         add_constraint(commodity_sources >= 1)
281         node_ext = commodity.receiver + 'proxy'
282         node = commodity.receiver
283         all_incoming_edges = variables[node_ext][node][commodity]
284         add_constraint(sum(all_incoming_edges) >= 1)
285
286
287 def add_proxy_flow_conservation(variables):
288     for commodity, node in product(commodities(), nodes()):
289         # Add flow conservation for proxies, as per an all-to-all topology
290         proxy = node + 'proxy'
291         in_to_proxy = 0
292         out_of_proxy = 0
293         for other_node in chain(proxies(), repeaters()):
294             if proxy == other_node:
295                 continue
296             in_to_proxy += sum(variables[other_node][proxy][commodity])
297             out_of_proxy += sum(variables[proxy][other_node][commodity])
298         add_constraint(in_to_proxy == variables[proxy][node][commodity])
299         add_constraint(out_of_proxy == variables[node][proxy][commodity])
300
301
302 def add_node_flow_conservation(variables):
303     for commodity, node in product(commodities(), nodes()):
304         # Add flow conservation for nodes, make sure they can only be origin
305         # for their own commodity, and does not terminate their own
306         # commodities
```

```
307         # TODO: Does not allow nodes to re-encode data for now
308         proxy = node + 'proxy'
309         if node == commodity.sender:
310             received = sum(variables[proxy][node][commodity])
311             add_constraint(received == 0, 'Node')
312         elif node != commodity.receiver:
313             sent = sum(variables[node][proxy][commodity])
314             received = sum(variables[proxy][node][commodity])
315             add_constraint(sent == received, 'Node')
316 
317 def add_repeater_flow_conservation(variables):
318     # Repeaters repeat incoming commodities out to all proxies, mangled to
319     # their desired commodity
320     for repeater, node in product(repeaters(), nodes()):
321         proxy = node + 'proxy'
322         left_side = 0
323         right_side = []
324         for commodity in commodities():
325             if commodity.sender == node:
326                 left_side += variables[proxy][repeater][commodity][0]
327                 recv_ext = commodity.receiver + 'proxy'
328                 right_side.append(variables[repeater][recv_ext][commodity][0])
329 
330                 # Never send traffic back to source (hopefully not needed)
331                 sender_ext = commodity.sender + 'proxy'
332                 edge_var = variables[repeater][sender_ext][commodity][0]
333                 add_constraint(edge_var == 0, 'Repeater flow')
334 
335         for outgoing in right_side:
336             add_constraint(left_side == outgoing, 'Repeaterflow')
337 
338 
339 
340 def get_constraints(variables):
341     constaint_sources = (
342         add_bandwidth_conservation,
343         add_all_commodities_must_be_sent_and_received_constraint,
344         add_proxy_flow_conservation,
345         add_node_flow_conservation,
346         add_repeater_flow_conservation,
347     )
```

```
348     for source in constaint_sources:
349         source(variables)
350     return constraints
351 
352 def print_problem_constraints(prob):
353     print 'Problem constraints:'
354     for v in prob.variables():
355         if v.varValue and v.name.startswith('__debug__'):
356             for c in prob.constraints.values():
357                 if ' %s ' % (v.name,) in str(c):
358                     print '\t%s' % c
359 
360 def find_path_between_nodes(variables, node, other_node):
361     commodity = commodity_from_nodes(node, other_node)
362     destination = node + 'proxy'
363     path = [other_node + 'proxy']
364     while path[-1] != destination:
365         # print path
366         path_hops, commodity = find_next_path_hops_and_commodity(variables,
367             path[-1], destination, commodity)
368         path.extend(path_hops)
369     else:
370         # Found path between nodes
371         path = list(reversed(path))
372         return path
373 
374 def find_next_path_hops_and_commodity(variables, origin, destination,
375     commodity):
376     incoming_paths = find_nodes_who_sends_commodity(variables, origin,
377         commodity)
378     if len(incoming_paths) == 2:
379         # There's a cycle (we go through another node), add the cycle and
380         # the actual exit to the path
381         return get_exit_path_from_proxy(variables, origin,
382             incoming_paths, commodity), commodity
383     elif len(incoming_paths) == 1:
384         return incoming_paths, commodity
385 
386     else:
387         if not origin.startswith('rep'):
388             raise ValueError('Commodity K%d not found in to %s' % (commodity,
```

```
389                 destination))
390         # Trace a repeater changing commodity type
391         commodity = find_mangled_commodity(variables, destination,
392             commodity.sender)
393         return find_path(variables, origin, destination, commodity), commodity
394
395 def get_exit_path_from_proxy(variables, origin, incoming_paths, commodity):
396     path = []
397     # Check for cycle
398     for incoming_path in incoming_paths:
399         if incoming_path == origin:
400             continue
401         # Do we send to that node as well as receive -> cycle.
402         if any(edge.varValue for edge in
403             variables[origin][incoming_path][commodity]):
404                 path.append(incoming_path)
405                 path.append(origin)
406                 break
407
408     # Add non-cycle path
409     for incoming_path in incoming_paths:
410         if incoming_path not in path:
411             # The other non-cycle path
412             path.append(incoming_path)
413             break
414     return path
415
416
417 def find_mangled_commodity(variables, repeater, sender):
418     all_node_commodities = [c for c in commodities() if c.sender == sender]
419     for c in all_node_commodities:
420         for sending_node, variable in variables.iteritems():
421             if repeater in variable and any(edge.varValue for edge in \
422                                             variable[repeater][c]):
423                 return c
424     raise ValueError('Mangled commodity not found.')
425
426
427 def find_path(variables, origin, destination, commodity):
428     path = []
429     last_node_found = origin
```

```
430     while last_node_found != destination:
431         sending_nodes = find_nodes_who_sends_commodity(variables,
432             last_node_found, commodity)
433         if sending_nodes:
434             last_node_found = sending_nodes[0]
435         else:
436             raise ValueError('No path found after repeater change, '
437                 'path: %s, origin: %s, dest: %s, c: %s' % (path, origin,
438                 destination, commodity))
439         path.append(last_node_found)
440     return path
441
442 def find_nodes_who_sends_commodity(variables, destination, commodity):
443     sending_nodes = []
444     for search_node, variable in variables.iteritems():
445         if destination in variable:
446             edges = variable[destination][commodity]
447             any_traffic = any(edge.varValue for edge in edges)
448             if any_traffic:
449                 sending_nodes.append(search_node)
450     return sending_nodes
451
452
453 def get_path_cost(variables, path, commodity):
454     cost = 0
455     for index, edge in enumerate(path[1:], 1):
456         sender, receiver = path[index-1], path[index]
457         cost += get_edge_latency(sender, receiver)
458     return cost
459
460 def print_solution(variables):
461     # Print the solution
462     for node, other_node in node_pairs():
463         commodity = commodity_from_nodes(node, other_node)
464         path = find_path_between_nodes(variables, node, other_node)
465         cost = get_path_cost(variables, path, commodity)
466         flow = sum(edge.varValue for edge in
467             variables[path[-2]][path[-1]][commodity])
468         path_as_str = ' -> '.join(path)
469         print '%s til %s (K%d): %s, flow: %d, cost: %dms' % (node, other_node,
470             commodity, path_as_str, flow, cost)
```

```
471
472 def print_bandwidth_usage(variables):
473     for node in nodes():
474         downlink_raw = case['nodes'][node]['downlink']
475         downlink_capacity = parse_bandwidth_into_slots(downlink_raw)
476         uplink_raw = case['nodes'][node]['uplink']
477         uplink_capacity = parse_bandwidth_into_slots(uplink_raw)
478         proxy = node + 'proxy'
479         downlink_used = sum(sum(edge.varValue for edge in
480             variables[proxy][node][commodity]) for commodity in commodities())
481         uplink_used = sum(sum(edge.varValue for edge in
482             variables[node][proxy][commodity]) for commodity in commodities())
483         downlink_percentage = float(downlink_used)/downlink_capacity
484         uplink_percentage = float(uplink_used)/uplink_capacity
485         print '%s downlink: %.1f, uplink: %.1f' % (node, downlink_percentage,
486             uplink_percentage)
487
488 def solve_case(args, number_of_edges):
489     global_start_time = time.time()
490     variables = initialize_variables(number_of_edges)
491
492     # TODO: Subtract repeater/re-encoder costs
493
494     constraints = get_constraints(variables)
495
496     if args.debug:
497         for i in range(len(constraints)):
498             constraints[i] += debug()
499
500     prob_type = LpMinimize if args.debug else LpMaximize
501     prob = LpProblem("interkontinental-asymmetric", prob_type)
502     objective = get_objective(variables, number_of_edges) if not args.debug \
503         else sum(_debug_vars)
504     logger.info('Objective: %s %s', LpSenses[prob.sense], objective)
505     prob += objective
506
507     for constraint in constraints:
508         prob += constraint
509
510     starttime = time.time()
511     pipe = io.StringIO()
```

```
512     res = GLPK(pipe=pipe).solve(prob)
513     endtime = time.time()
514 
515     output = pipe.getvalue()
516     memstart = output.find('Memory used')
517 
518 
519     for commodity in commodities():
520         print 'K%d: %s -> %s' % (commodity, commodity.sender,
521             commodity.receiver)
522 
523     if res < 0:
524         print 'Unsolvable!'
525         sys.exit(1)
526     else:
527         if args.debug:
528             print_problem_constraints(prob)
529 
530         dump_nonzero_variables(prob)
531         print
532 
533         print_solution(variables)
534         print_bandwidth_usage(variables)
535 
536         print "Score =", value(prob.objective)
537         print 'Found solution in %.3fs' % (endtime - starttime)
538         with open('results.csv', 'a') as fh:
539             memory_segment = ''.join(output[memstart:memstart+20])
540             memory_used = float(memory_segment.split()[2])*10**6
541             solve_time = endtime - starttime
542             build_time = starttime - global_start_time
543             retrace_time = time.time() - endtime
544             fh.write('%.3f,%.3f,%.3f,%d' % (solve_time, build_time,
545                 retrace_time, memory_used) + '\n')
546 
547 
548 if __name__ == '__main__':
549     main()
```

Appendix B
capture.sh

The `capture.sh`[1] script was executed on each of the machines in the cluster to start taking regular screenshots, and print the current local time to the terminal. This allowed calculating the end-to-end latency as the difference between the local current time (which was kept in sync between the nodes using a common nearby NTP server) and the time sent by each of the nodes in the conversation. The observed times was later manually interpreted and noted down in a Comma-Separated Values (CSV) document, which was then used for further analysis. Manual interpretation was unavoidable since the timestamps was often overlayed with previous timestamps, due to several components in the end-to-end chain having refresh rates far larger than the millisecond accuracy we'd prefer.

The script also starts `tcpdump`, to capture how much traffic is sent and received from the different hosts, for further analysis.

```
1  #!/bin/bash
2
3  # Starts the actual capture of screenshots, and prints the current time
4  # offset and local clock
5  session_id=$(hostname)-$(date +"%s")
6
7  # Print which role we're configured as
8  # grep $(hostname) apply-case.py
9  my_ip=$(curl -s canhazip.com)
10 iface=$(ifconfig | grep "inet addr:$my_ip" -B1 | head -1 | cut -d" " -f1)
11
12 # Start sniffing traffic to reconstruct bandwidth usage between peers,
```

[1] Available online at https://github.com/thusoy/hybrid-video-topology/blob/master/tools/capture.sh

```
13 # excluding some common spammy services (ssh, spotify, name lookups, ssdp)
14 sudo tcpdump -i $iface -nNqtts68 "port not 3271 and port not 17500 and port \
15     not 137 and port not 138 and port not 1900 and (udp or tcp)" \
16     -w /tmp/$session_id.pcap 2> /dev/null &
17 tcpdump_pid=$!
18
19 python clock.py &
20 clock_pid=$!
21 trap cleanup INT
22
23 function cleanup () {
24     echo -e "\n$session_id/$my_ip"
25     kill -9 $clock_pid
26     sudo kill -INT $tcpdump_pid
27     exit $?
28 }
29
30 while :; do
31     gm import -window root /tmp/screen-$session_id-$(date +"%s").png
32     sleep 10
33 done
```

Appendix C
apply-case.py script

This is the script[1] used to limit available bandwidth and set up inter-node latencies in the test cluster. The script loads it's configuration from the same `cases.yml` case definition file as some of the other scripts. It then calls out to the Linux command `tc`, which instructs the kernel to assign the given parameters to outgoing traffic.

Incoming traffic is only restricted in bandwidth, no latencies are being applied.

The script automatically detects from the IP of the host running it which role it should be assigned in the case, which makes it fast and easy to automate execution of the script across several machines. This is based on another YAML-file `rolemap.yml`, which holds a simple mapping from role names to their hostname. This enables the script to use the hostname of the current machine to find the role, and to use DNS to find the IP of the other machines, which is necessary as parameters to `tc`.

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  """
4      Applies network restrictions according to a desired case.
5
6      :copyright: (c) 2015 by Tarjei Husøy
7      :license: MIT, see http://opensource.org/licenses/MIT
8  """
9
10 import argparse
11 import requests
12 import socket
```

[1] Available online at https://github.com/thusoy/hybrid-video-topology/blob/master/tools/apply-case.py

```
13 import os
14 import re
15 import yaml
16 from subprocess import call as _call, check_output
17
18
19 TC=["sudo", "tc"]
20 SSH_PORT = 3271
21
22
23 def load_cases(case_file):
24     with open(case_file) as fh:
25         cases = yaml.load(fh)
26     for case in cases.values():
27         for role in case['nodes']:
28             for other in case['nodes']:
29                 if role == other:
30                     continue
31                 if other not in case['nodes'][role]:
32                     case['nodes'][role][other] = case['nodes'][other][role]
33     return cases
34
35 def load_role_map(map_file):
36     with open(map_file) as fh:
37         return yaml.load(fh)
38
39
40 def main():
41     default_case_file = os.path.join(os.path.dirname(__file__), 'cases.yml')
42     default_role_map = os.path.join(os.path.dirname(__file__), 'rolemap.yml')
43     cases = load_cases(default_case_file)
44     role_map = load_role_map(default_role_map)
45     parser = argparse.ArgumentParser()
46     parser.add_argument('-r', '--role', help='Which role should be '
47         'activated. Defaults to checking whether any role is associated '
48         'with your IP address')
49     parser.add_argument('-c', '--case', help='Which case to load',
50         default='traveller', choices=cases.keys())
51     parser.add_argument('-C', '--clear', action='store_true',
52         help="Clear all existing rules without applying a new case")
53     args = parser.parse_args()
```

```
54      if args.clear:
55          print('Clearing any existing rules...')
56          clear_all_rules()
57          return
58      case = cases[args.case]
59      ipify_role_map(role_map, case['nodes'].keys())
60      role = args.role or get_role_from_ip(role_map)
61      activate_role(role, role_map, case)
62
63
64  def clear_all_rules():
65      device = get_interface_device()
66      call(TC + ['qdisc', 'del', 'dev', device, 'ingress'], silent=True)
67      call(TC + ['qdisc', 'del', 'dev', device, 'root'], silent=True)
68
69
70  def ipify_role_map(role_map, roles_in_use):
71      """ Needed since a DNS lookup usually will only return IPv4 addresses,
72      while the WebRTC-discovery protocol will also find the node's IPv6
73      addresses. We thus SSH to each node and query its interface list for
74      all IPs it'll answer on.
75      """
76      ip_regex = re.compile(r'[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}')
77      if_cmd = ("ifconfig"
78          " | grep -o 'inet addr:[^ ]*'"
79          " | cut -d: -f2"
80          " | grep -v 127.0.0.1")
81
82      if6_cmd = ("ifconfig"
83          " | grep -o \"inet6 addr: [^ ]*\""
84          " | cut -d' ' -f3"
85          " | grep -v '^fe80'"
86          " | grep -v '^::1' || exit 0")
87      for role in roles_in_use:
88          hostname_or_ip = role_map[role]
89          if not ip_regex.match(hostname_or_ip):
90              # Not an IP in the role map, let's find the IPs
91              ips = []
92              ipv4_addr_out = check_output([
93                  'ssh',
94                  '-o StrictHostKeyChecking=no',
```

```
95                  hostname_or_ip,
96                  '-p %d' % SSH_PORT,
97                  if_cmd])
98              ips += ipv4_addr_out.strip().split('\n')
99              ipv6_out = check_output([
100                 'ssh',
101                 '-o StrictHostKeyChecking=no',
102                 hostname_or_ip,
103                 '-p %d' % SSH_PORT,
104                 if6_cmd])
105             if ipv6_out:
106                 ips += ipv6_out.strip().split('\n')
107             # Probably a hostname, resolve it and use the IP in the role map
108             role_map[role] = ips
109
110
111 def get_role_from_ip(rolemap):
112     my_ip = get_my_ip()
113     for role, ips in rolemap.items():
114         if my_ip in ips:
115             return role
116     raise ValueError("Role not found for ip %s" % my_ip)
117
118
119 def get_my_ip():
120     """ Get the IP of the box running this code. """
121     # This function is memoizable
122     return requests.get('http://httpbin.org/ip').json()['origin']
123
124
125 def activate_role(role, role_map, case):
126     print 'Activating role %s' % role
127     clear_all_rules()
128     uplink = case['nodes'][role]['uplink']
129     downlink = case['nodes'][role]['downlink']
130     add_roots(downlink, uplink)
131     add_role_rules(role, role_map, case)
132
133
134 def add_roots(downlink, uplink):
135     device = get_interface_device()
```

```
136     # Limit uplink
137     call(TC + ['qdisc', 'add', 'dev', device, 'root', 'handle', '1:', 'tbf',
138         'rate', uplink, 'buffer', '20000', 'limit', '30000'])
139     call(TC + ['qdisc', 'add', 'dev', device, 'parent', '1:', 'handle', '2:',
140         'htb'])
141 
142     # Limit downlink. This effectively limits UDP to the given downlink
143     # bandwidth, but TCP will have lower performance, because of negative
144     # effects of the window size and the large delays. This doesn't matter in
145     # this case, as our traffic is UDP-based, but it's worth keeping in mind.
146     call(TC + ['qdisc', 'add', 'dev', device, 'handle', 'ffff:', 'ingress'])
147     call(TC + ['filter', 'add', 'dev', device, 'parent', 'ffff:', 'protocol',
148         'ip', 'prio', '50', 'u32', 'match', 'ip', 'src', '0.0.0.0/0',
149         'police', 'rate', downlink, 'burst', downlink, 'flowid', ':1'])
150 
151 
152 def add_role_rules(role, role_map, case):
153     device = get_interface_device()
154     ipv4_regex = re.compile(r'[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}')
155     for role_num, (other_role, delay_config) in enumerate(
156             sorted(case['nodes'][role].items(), key=lambda t: t[0]), 1):
157         if other_role in ('uplink', 'downlink', 'class'):
158             continue
159         class_id = role_num*10
160         handle_id = str(class_id) + '1'
161         delay_config_as_list = delay_config.split()
162         if not other_role in role_map:
163             raise ValueError('Role does not have a specified target in the '
164                 'role_map: %s' % other_role)
165         call(TC + ['class', 'add', 'dev', device, 'parent', '2:', 'classid',
166             '2:%d' % class_id, 'htb', 'rate', case['nodes'][role]['uplink']])
167         call(TC + ['qdisc', 'add', 'dev', device, 'parent', '2:%d' % class_id,
168             'handle', '%s:' % handle_id, 'netem', 'delay'] +
169             delay_config_as_list)
170         for ip in role_map[other_role]:
171             if ipv4_regex.match(ip):
172                 call(TC + ['filter', 'add', 'dev', device, 'protocol', 'ip',
173                     'parent', '2:0', 'prio', '3', 'u32', 'match', 'ip', 'dst',
174                     ip, 'flowid', '2:%d' % class_id])
175             else:
176                 call(TC + ['filter', 'add', 'dev', device, 'protocol', 'ipv6',
```

```
177                     'parent', '2:0', 'prio', '4', 'u32', 'match', 'ip6',
178                     'dst', ip, 'flowid', '2:%d' % class_id])
179
180
181 def get_interface_device():
182     # This function is memoizable
183     my_ip = get_my_ip()
184     all_interfaces = []
185     ip_links_output = check_output(['ip', 'link', 'list'])
186     for line in ip_links_output.split('\n'):
187         if not line.startswith('%s:' % (len(all_interfaces) + 1)):
188             continue
189         interface = line.split(':')[1].strip()
190         all_interfaces.append(interface)
191     for interface in all_interfaces:
192         ip_addr_output = check_output(['ip', 'addr', 'show', 'dev',
193             interface])
194         if my_ip in ip_addr_output:
195             return interface
196     raise ValueError('Failed to find correct interface to use!')
197
198
199 def call(args, silent=False, **kwargs):
200     devnull = open(os.devnull, 'wb')
201     if silent:
202         kwargs['stdout'] = devnull
203         kwargs['stderr'] = devnull
204     else:
205         print 'Running cmd: %s' % ' '.join(args)
206     _call(args, **kwargs)
207
208
209 if __name__ == '__main__':
210     main()
```

Appendix D
Extracting `getStats` data

Data extraction from the browsers while running the tests consisted of two parts: A JavaScript file that was injected into the browser extracting the actual data, and a webserver running outside the test cluster storing the captured data for analysis.

Since the data directly from `getStats` is overly verbose and a bit hard to navigate, some cleanup is performed in the browser before shipping it off to storage, to have as small an impact on the test bandwidth as possible. If no influence of analytics is desired, a local web server could have been spawned on each node running in the test, preventing any external data usage during the duration of the test. The data could then be collected later. This was not considered necessary for these tests, as the overhead of relatively small JSON posted over HTTP is very small compared to streaming live video.

The JavaScript was injected into Chrome using the extension "Custom JavaScript for websites"[1]. The script was configured to dump statistics to the external server every second.

The client-side script is very compact and carries negligible overhead on page load. Minified and gzipped it weighs in at 1.3kB, with no external dependencies.

Since the returned data from the `getStats` API differs between browsers, it has to be cleaned up before it can be presented for analysis. This could be done either on the client side, or on the server side. I'd argue that this is most sensibly done on the server side, since if keeps the client-side script lighter, prevents it from having to incorporate changes as the browsers change, and reduces the likelihood of crashes. If parsing is done here, the script will have to know about every change done by every browser, which will make it huge and failure-prone when the browsers change the data format. If the script submits the user agent string with the results, the server

[1] Available here: https://chrome.google.com/webstore/detail/custom-javascript-for-web/poakhlngfciodnhlhhgnaaelnpjljija

can collect them all, and when new data appears, it can add handlers for that, and build up support for ever more user agents in the backend.

However, there's a balance to be found. The `getStats` API is very verbose, returning lots of data of connection candidates and key pairs and lots of data that is generally not that interesting for analytics. To reduce impact on bandwidth, it's of interest to the client to not have to submit a lot of junk that will be discarded by the backend anyway. Thus the script used here tries to find the middle way; it detects the interesting parts (audio in/out, video in/out and current connection details) and discards the rest, sending the interesting parts off to the server without any further parsing of the contents.

Support for Firefox is only half-way there, reports are fetched from the API but the aforementioned detection of report types is not compatible with Firefox. Extending the script to support this should be easy.

```
/*
Loosely based on getStats.js by Muaz Khan (available at
github.com/muaz-khan/getStats)

(c) 2015 by Tarjei Husøy, MIT license
*/

(function () {
    function preprocessGoogleGetStats(reports, keysToSkip, callback) {
        var result = {
            audio: {},
            video: {},
            timestamp: new Date().getTime(),
        };

        for (var i = 0; i < reports.length; i++) {
            var report = reports[i],
                isIncomingAudio = report.audioOutputLevel !== undefined,
                isOutgoingAudio = report.audioInputLevel !== undefined,
                isIncomingVideo = report.googFrameRateReceived !== undefined,
                isOutgoingVideo = report.googFrameRateSent !== undefined,
                // TODO: googActiveConnection is only true between two
                // Chrome browsers, find a better way.
                isConnectionInUse = report.type == 'googCandidatePair' &&
                    report.googActiveConnection == 'true',
```

```
26                  isBandwidthEstimation = report.type == 'VideoBwe';
27
28              if (isIncomingAudio) {
29                  result.audio.incoming = parseReport(report,
30                      'audio.incoming.', keysToSkip);
31              } else if (isOutgoingAudio) {
32                  result.audio.outgoing = parseReport(report,
33                      'audio.outgoing.', keysToSkip);
34              } else if (isIncomingVideo) {
35                  result.video.incoming = parseReport(report,
36                      'video.incoming.', keysToSkip);
37              } else if (isOutgoingVideo) {
38                  result.video.outgoing = parseReport(report,
39                      'video.outgoing.', keysToSkip);
40              } else if (isBandwidthEstimation) {
41                  result.video.bandwidth = parseReport(report,
42                      'video.bandwidth.', keysToSkip);
43              } else if (isConnectionInUse) {
44                  result.connection = parseConnectionReport(report);
45              }
46          }
47
48          if (result.connection !== undefined) {
49              callback(result);
50          } else {
51              console.log("Failed to find active connection, try again...");
52              console.log(reports);
53          }
54      }
55
56      function getPrivateStats(peer, callback, keysToSkip) {
57          _getStats(peer, function (reports) {
58              preprocessGoogleGetStats(reports, keysToSkip, callback);
59          });
60      }
61
62      function parseReport(report, prefix, keysToSkip) {
63          var parsedReport = {};
64          for(var key in report) {
65              var qualifiedKeyName = prefix + key;
66              var shouldSkip = keysToSkip.indexOf(qualifiedKeyName) != -1;
```

```
67              if (report.hasOwnProperty(key) && !shouldSkip) {
68                  parsedReport[key] = report[key];
69              }
70          }
71          return parsedReport;
72      }
73
74      function parseConnectionReport(report) {
75          return {
76              local: {
77                  candidateType: report.googLocalCandidateType,
78                  ipAddress: report.googLocalAddress
79              },
80              remote: {
81                  candidateType: report.googRemoteCandidateType,
82                  ipAddress: report.googRemoteAddress
83              },
84              transport: report.googTransportType
85          };
86      }
87
88
89      // a wrapper around getStats which hides the differences (where possible)
90      // following code-snippet is taken from somewhere on github
91      function _getStats(peer, callback) {
92          if (navigator.mozGetUserMedia) {
93              // Running on Firefox, Firefox requires the stream to fetch
94              // stats for as an argument to getStats.
95              var localStreams = peer.getLocalStreams()[0];
96              var tracks = localStreams.getTracks();
97              tracks.forEach(function (track) {
98                  peer.getStats(track).then(function (res) {
99                          var items = [];
100                         res.forEach(function (result) {
101                             items.push(result);
102                         });
103                         callback(items);
104             }, function (reason) {
105                 console.log("getStats failed. Reasons: " + reason);
106             });
107         });
```

```
108         } else {
109             peer.getStats(function (res) {
110                 var items = [];
111                 res.result().forEach(function (result) {
112                     var item = {};
113                     result.names().forEach(function (name) {
114                         item[name] = result.stat(name);
115                     });
116                     item.id = result.id;
117                     item.type = result.type;
118                     item.timestamp = result.timestamp;
119                     items.push(item);
120                 });
121                 callback(items);
122             });
123         }
124     };
125 
126     window.getStats = function (peer, callback, keysToSkip) {
127         keysToSkip = keysToSkip || [];
128         getPrivateStats(peer, callback, keysToSkip);
129     }
130 })();
131 
132 // appear.in-specific code starts here
133 (function () {
134     function ajax(url, config) {
135         // $.ajax-like wrapper around XHR, without any jQuery-dependencies
136         var method = config.type || 'GET';
137         var xhr = new XMLHttpRequest();
138         xhr.onreadystatechange = function () {
139             if (xhr.readyState = XMLHttpRequest.DONE) {
140                 if (xhr.status >= 200 && xhr.status < 300) {
141                     if (config.success) {
142                         config.success(xhr, xhr.status);
143                     }
144                 } else if (xhr.status >= 400 && xhr.status < 600) {
145                     // Client or server error
146                     if (config.error) {
147                         config.error(xhr, xhr.status);
148                     }
```

```
149                 }
150             }
151         }
152         if (config.error) {
153             xhr.onerror = function (xhrStatusEvent) {
154                 config.error(xhr, 'Non-HTTP failure. Could be connection ' +
155                     'related, CORS, etc. Check console for details.');
156             };
157         }
158         xhr.open(method, url);
159         if (config.contentType) {
160             xhr.setRequestHeader('Content-Type', config.contentType);
161         }
162         var payload = config.data || '';
163         xhr.send(payload);
164     }
165 
166     function shipReports(reports) {
167         ajax('https://collect.thusoy.com/collect', {
168             type: 'POST',
169             data: JSON.stringify(reports),
170             contentType: "application/json",
171             error: function (xhr, status) {
172                 console.log("Posting stats to collector failed: " + status);
173             }
174         });
175     }
176 
177     function createReportAggregator() {
178         // Bundles all reports from the same time into a list that's shipped
179         // of to the collector at the same time. Uses a list of remotes
180         // currently collected, and ships off when the same remote appears
181         // again.
182         var remoteIps = [];
183         var currentReports = [];
184 
185         return function (result) {
186             var remoteIp = result.connection.remote.ipAddress;
187             if (remoteIps.indexOf(remoteIp) != -1) {
188                 // New set of connections coming, flush
189                 shipReports(currentReports);
```

```
190                 currentReports = [];
191                 remoteIps = [];
192             }
193             remoteIps.push(remoteIp);
194             currentReports.push(result);
195         }
196     }
197
198     // TODO: ideal API for getStats: getStats(funcThatGetsConnections,
199     //       resultFunc, interval)
200     // This would put aggregation within the API, instead of outside. If
201     // interval is missing, only call once.
202     function printStats() {
203         var ngInjector = angular.element(document.body).injector();
204         var rtcmanager = ngInjector.get('RTCManager');
205         var peerConnections = rtcmanager.getPeerConnections();
206         // TODO: Allow skipping entire categories by 'audio.*', or the same
207         // field in all categories, like '*.ssrc'
208         var skipList = [
209             'audio.incoming.ssrc',
210             'audio.outgoing.ssrc',
211             'video.incoming.ssrc',
212             'video.outgoing.ssrc',
213         ];
214         for (var i = 0; i < peerConnections.length; i++) {
215             var peerConnection = peerConnections[i];
216             getStats(peerConnection, reportAggregator, skipList);
217         }
218         setTimeout(printStats, 1000);
219     }
220     var reportAggregator = createReportAggregator();
221
222     console.log("Starting stats collection in 5s");
223     setTimeout(printStats, 5000);
224 })();
```

The web server code storing the incomng data is quite simple. It performs some very basic sanity checking of the incoming data, and appends it to a file in a customizable location. It uses the Flask python framework[2].

[2]Available here: http://flask.pocoo.org/

```python
# -*- coding: utf-8 -*-
"""
    Webserver that appends incoming getStats data to file.

    :copyright: (c) 2015 by Tarjei Husøy
    :license: MIT, see http://opensource.org/licenses/MIT
"""

import flask
import argparse
import json

app = flask.Flask(__name__)

@app.route('/collect', methods=['POST'])
def collect():
    data = flask.request.get_json()
    if sanity_check_data(data):
        with open(app.config['TARGET'], 'a') as target:
            target.write(request.data + '\n')
        return 'Thank you'
    else:
        return 'Missing mandatory arguments', 400


def sanity_check_data(data):
    required_properties = (
        'audio',
        'video',
        'connection',
        'timestamp',
    )
    if data and isinstance(data, list):
        for report in data:
            if not all(key in report for key in required_properties):
                return False
        return True
    return False


```

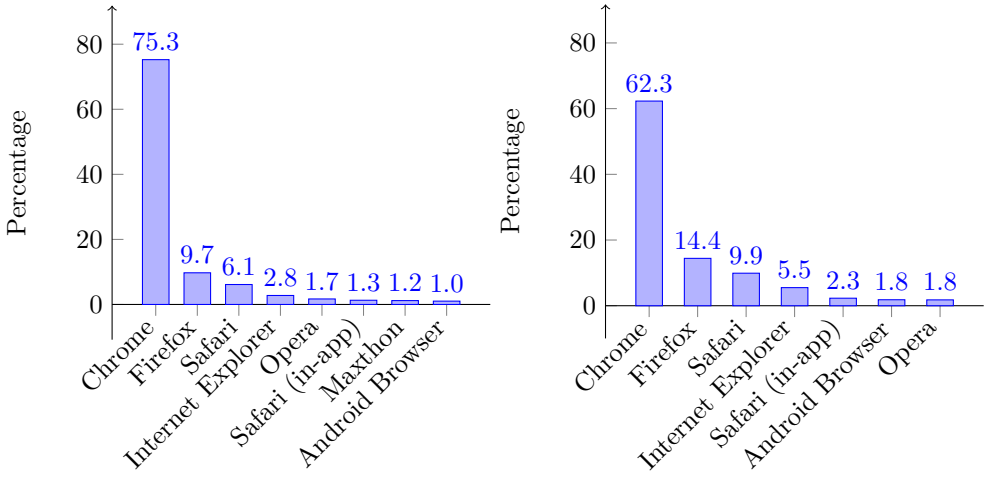
```
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-t', '--target', help='Where to store incoming data',
        default='data.dat')
    parser.add_argument('-d', '--debug', default=False, action='store_true')
    args = parser.parse_args()
    app.config['TARGET'] = args.target
    app.run(host='0.0.0.0', port=9000, debug=args.debug)
```

Appendix E
appear.in Usage Data

For this thesis, appear.in has been generous enough to lend some insight into their user data. The dataset is quite interesting; take for example the difference between browser popularity for new users and total sessions in Figure E.1 – Chrome has a substantially bigger portion of the latter. I consider it likely that it's driven by Chrome's superior performance to Firefox in the tests performed in chapter 4, which yields other questions for browser vendors. Are users so fluid in terms of browser choice that they pick whichever is best suited for the task at hand? I suspect that might be the case, but for now possibly only among the early adopters that find and use services like appear.in.

Although, it could also be the other way around – that users on other browsers are less likely to return given their browser's inability to deliver adequate service. Whichever the actual cause is, browsers with poor WebRTC performance are loosing shares to those who can deliver, and everyone would gain from support being more widespread. This is based on the reasoning that Chrome users will also be more satisfied with the experience if their non-Chrome friends also have a better experience.

Browser breakdown, by total sessions and new sessions, is given in Figure E.1. An overview of what devices users on appear.in use, is given in Figure E.2, and conversation sizes are broken down in Figure E.3.



(a) Breakdown of sessions on appear.in, browsers with more than 1% from mid May to mid June 2015

(b) Breakdown of new users on appear.in, browsers with more than 1% from mid May to mid June 2015

Figure E.1: New users compared to all sessions on appear.in, mid-May to mid-June 2015

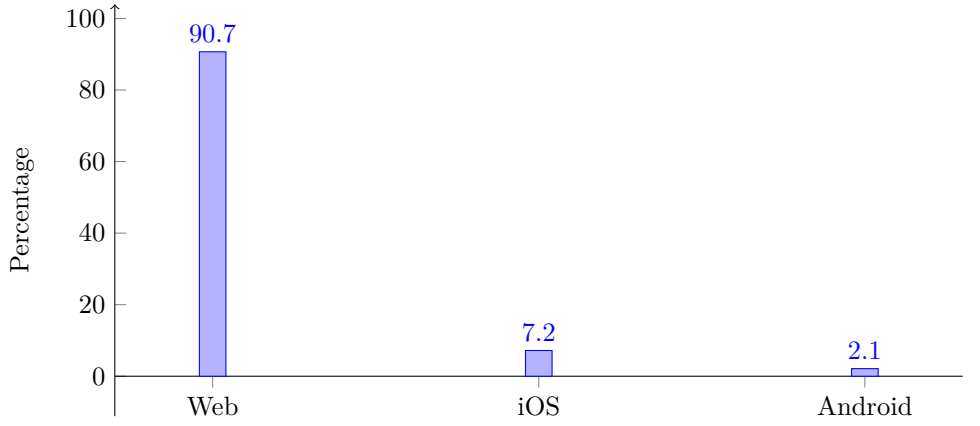
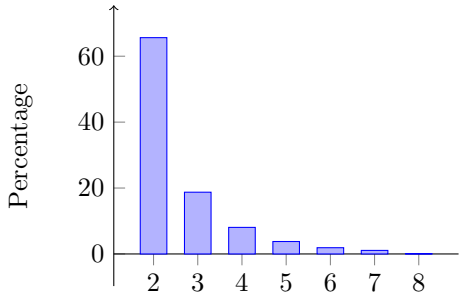
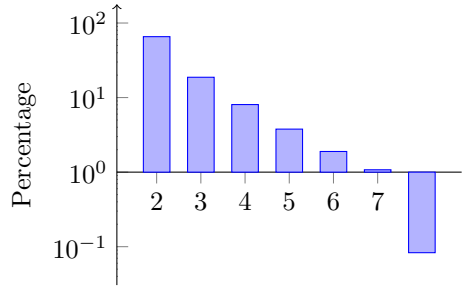


Figure E.2: Devices used on appear.in, mid-May to mid-June 2015



(a) #Users in a conversation, linear scale



(b) #Users in a conversation, log scale

Figure E.3: How many users are present in conversations on appear.in, mid-May to mid-June 2015

Appendix F
Comparison of Sampling Methods

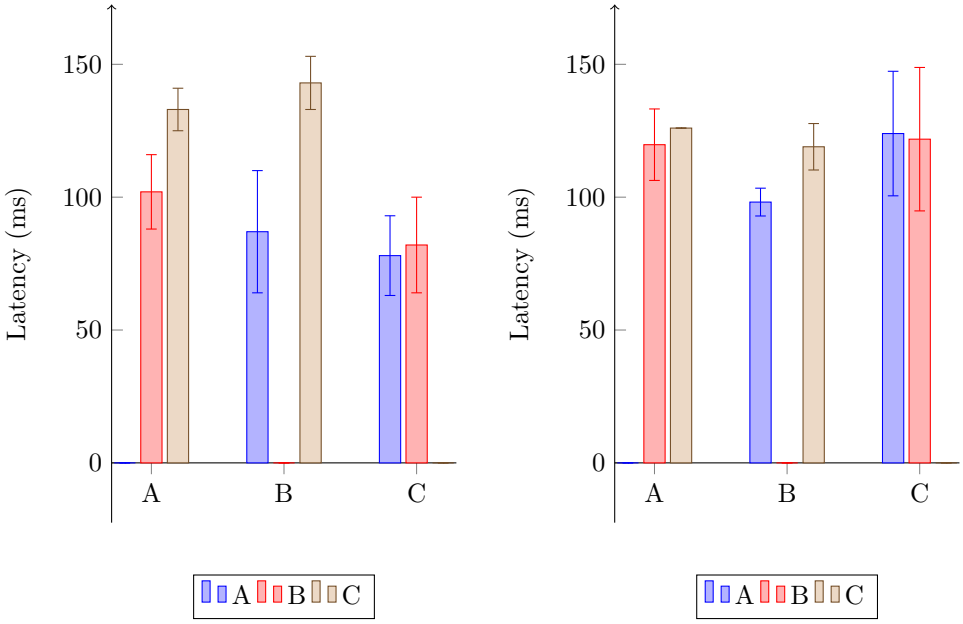
While the direct comparison between Firefox and Chrome in chapter 4 is interesting, any discrepancy observed there could also be a result of different performance between the browsers, not just the sampling method used.

To directly compare the methods, a subset of the tests were run with both methods on Chrome, so that the results could be compared. Note that like the other tests, since the method that films the timer is very labor intensive, only a single test run was performed, thus the sample size is too small to say anything conclusive. However, we do have an indication that the methods are both fairly accurate.

Then there's the bitrate sampling methods, `tcpdump` vs. `getStats`, which are compared in Figure F.3 and Figure F.4. They seem to be very well aligned, but `getStats` seems to underreport actual traffic a little bit. I suspect this might be due to the RTP Control Protocol (RTCP) stream not being accounted for, RTCP can take up to 5% of the session bandwidth. If it's left out of the `getStats` reports, we can assume the full session bandwidth is 5.3% higher than what we can see in the graphs, which means they're about 108 kbps lower than the actual numbers. Taking this into account, they're practically equal.[1]

Figure F.4 shows a bit higher discrepancy for bitrate from node B to node C. This might be because node B is closer to saturating it's outbound link, which increases the odds of queued and dropped packets, and could explain why the effective bitrate seen by the application is lower than what's received by the interface. This is not thought to have any significant effect on conclusions made from this data.

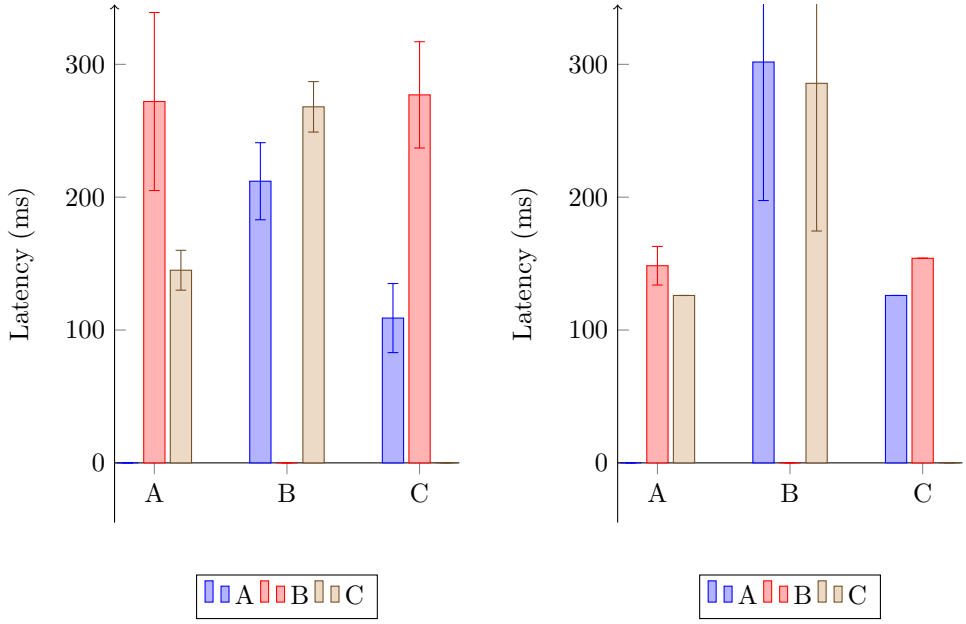
[1] This could be verified by checking the dump files to see what ports were used, but `libjingle` multiplexes RTP and RTCP over the same port, so there's no way to extract the RTCP overhead without doing deep packet inspection, which is hard since it's all DTLS.



(a) Latencies measured by filming a timer

(b) Latencies reported by `getStats`

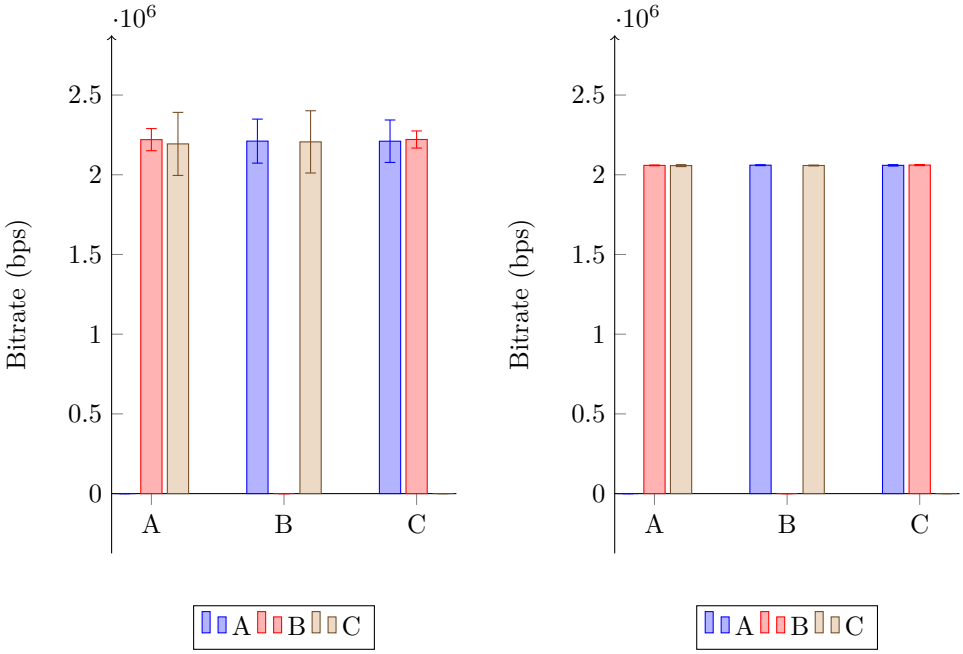
Figure F.1: Timer broadcasting and getStats compared for three nodes without traffic shaping. Note that the timer broadcasting has only 6 samples, while `getStats` has 80.



(a) Latencies measured by filming a timer

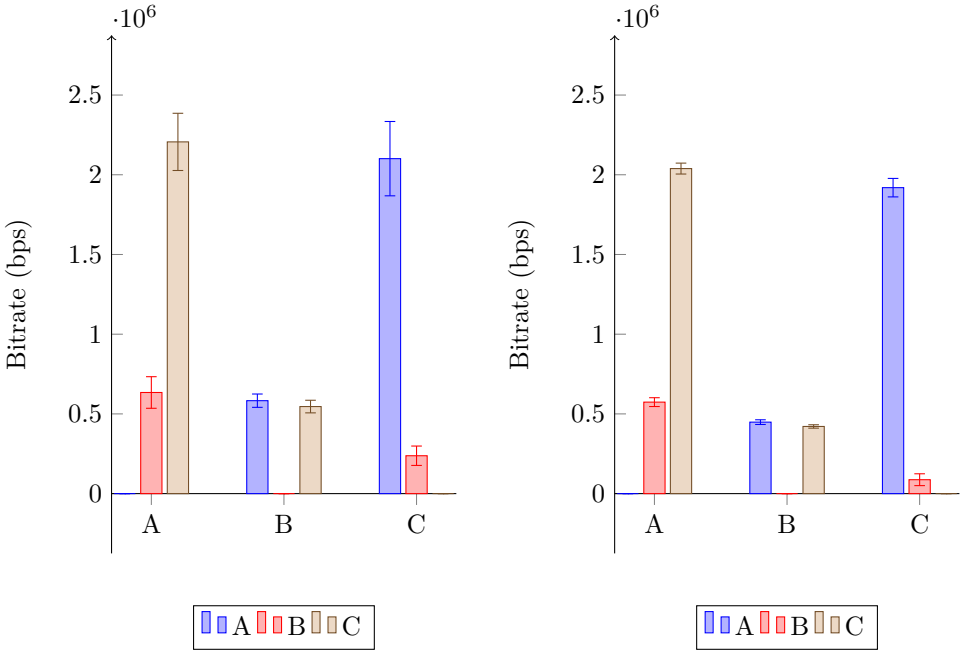
(b) Latencies reported by `getStats`

Figure F.2: Timer broadcasting and getStats compared for the traveller test case. Note that the timer broadcasting has only 6 samples, while `getStats` has 80.



(a) `tcpdump` (b) `getStats`

Figure F.3: Bitrates reported by tcpdump and getStats compared, no traffic shaping. Sample size was 120 for both methods.



(a) `tcpdump` (b) `getStats`

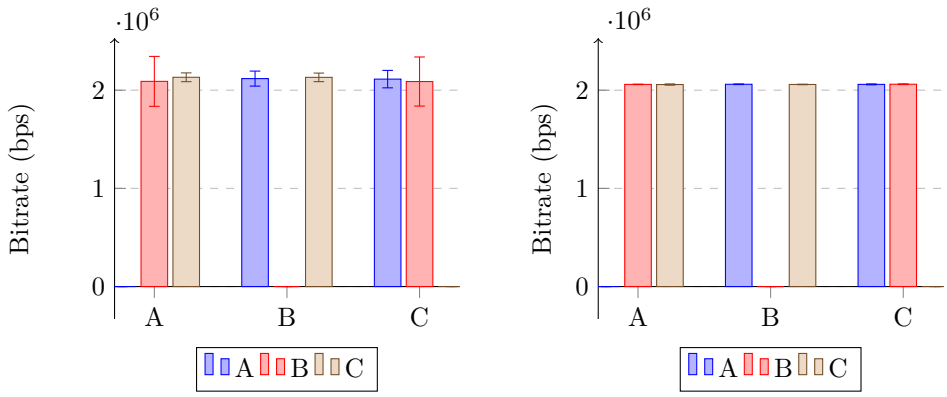
Figure F.4: Bitrates reported by tcpdump and getStats compared, "traveller" test case. Sample size was 120 for both methods.

Appendix G
No Traffic Shaping Comparison

The full data set for the browsers compared, without any traffic shaping applied, follows.

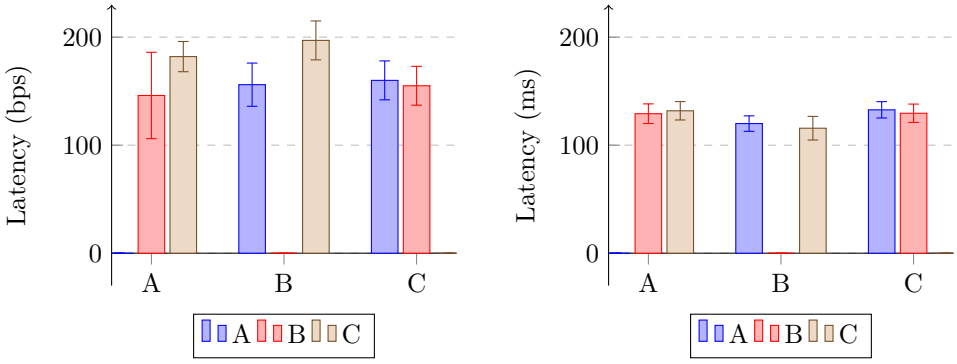
The only big outlier here is node E on the seven person test case. For some reason, it sends consistently less video out, and with higher latencies than all the other nodes in the test. It's unknown why this is, but could be because of more traffic on the subnet that node was placed on. These sort of random perturbations of network links should be expected on the Internet, thus applications need to be resilient to these events. Note that the Chrome tests were run a couple of days later and was probably not affected by the same outburst, thus yielding Firefox in a slightly worse light here than what is reasonable.

The full data set can be found in the project repository, at https://github.com/thusoy/hybrid-video-topology.



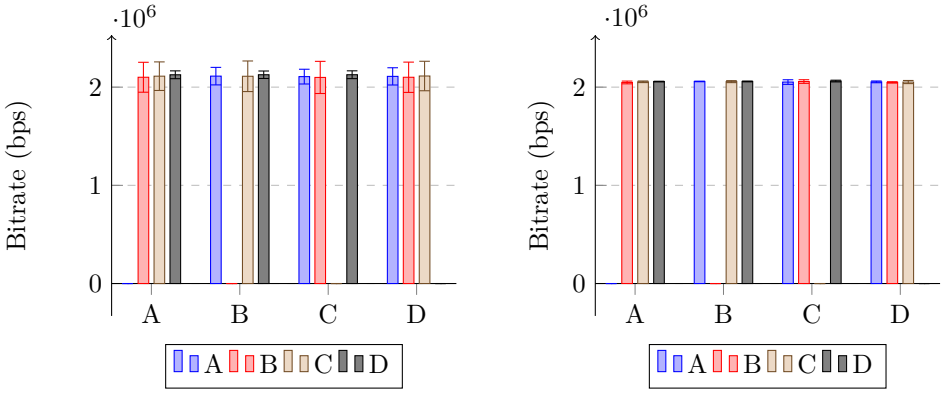
(a) Firefox (b) Chrome

Figure G.1: Observed bitrates with three people, no traffic shaping



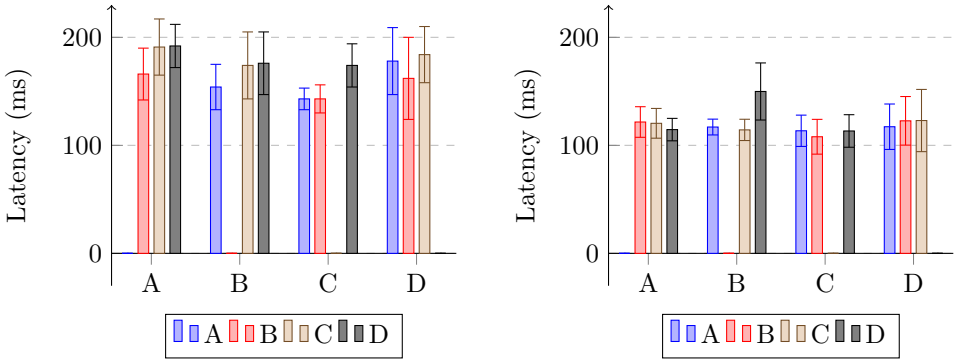
(a) Firefox (b) Chrome

Figure G.2: Observed latencies with three people, no traffic shaping



(a) Firefox (b) Chrome

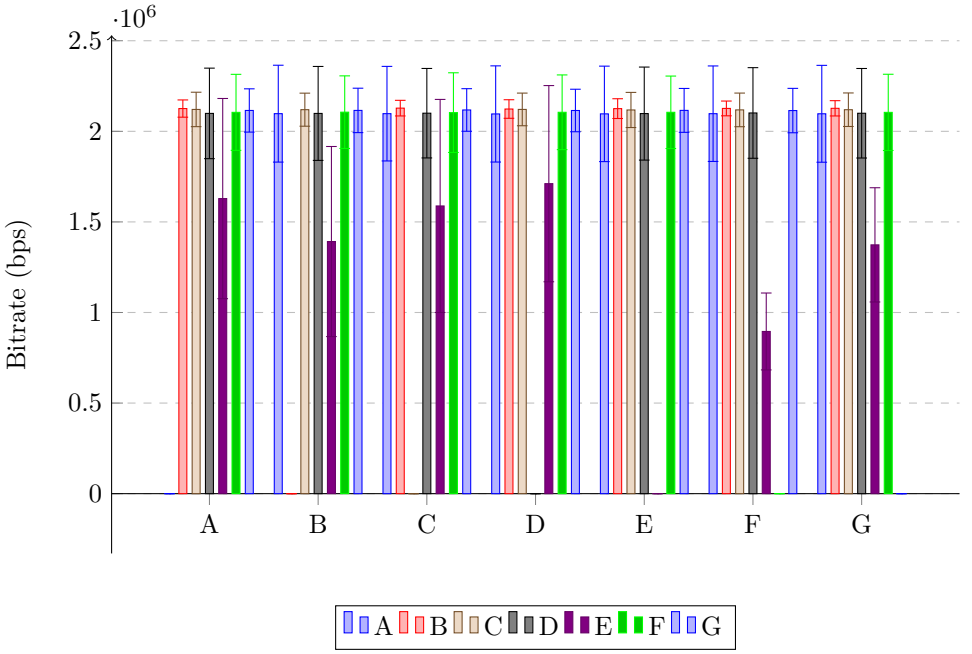
Figure G.3: Observed bitrates with four people, no traffic shaping



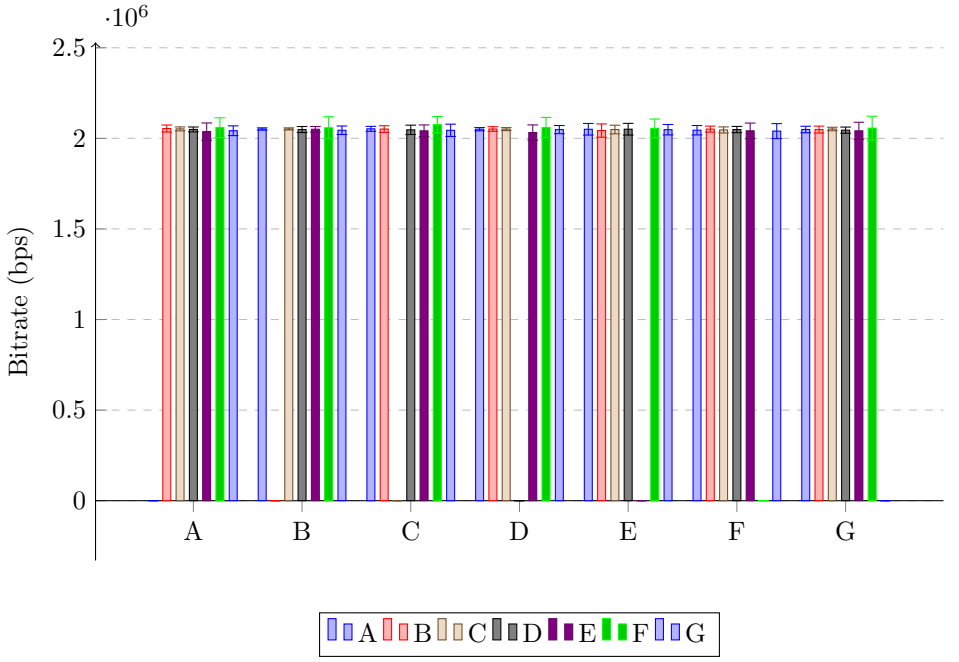
(a) Firefox

(b) Chrome

Figure G.4: Observed latencies with four people, no traffic shaping

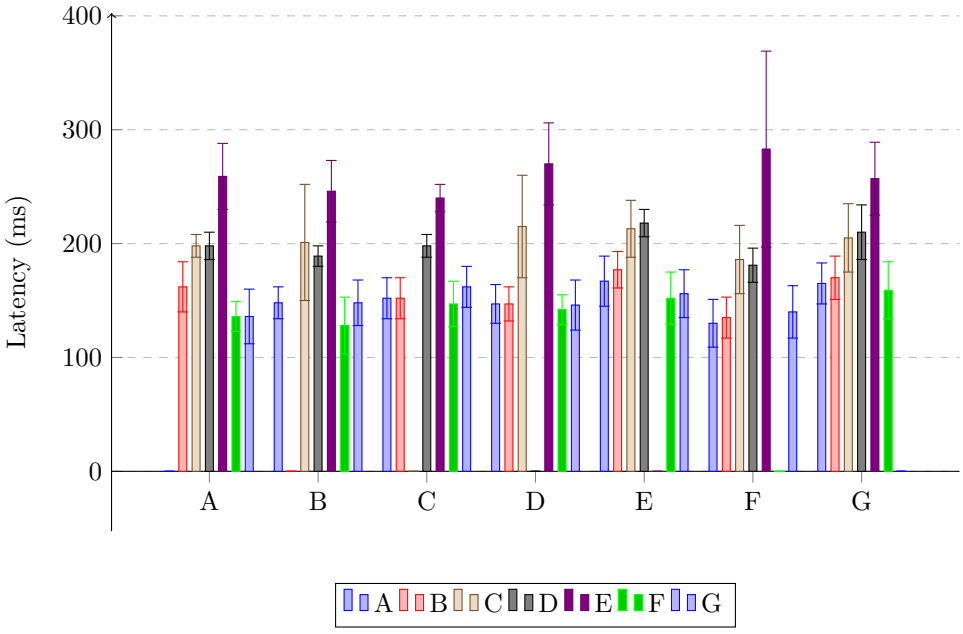


(a) Firefox

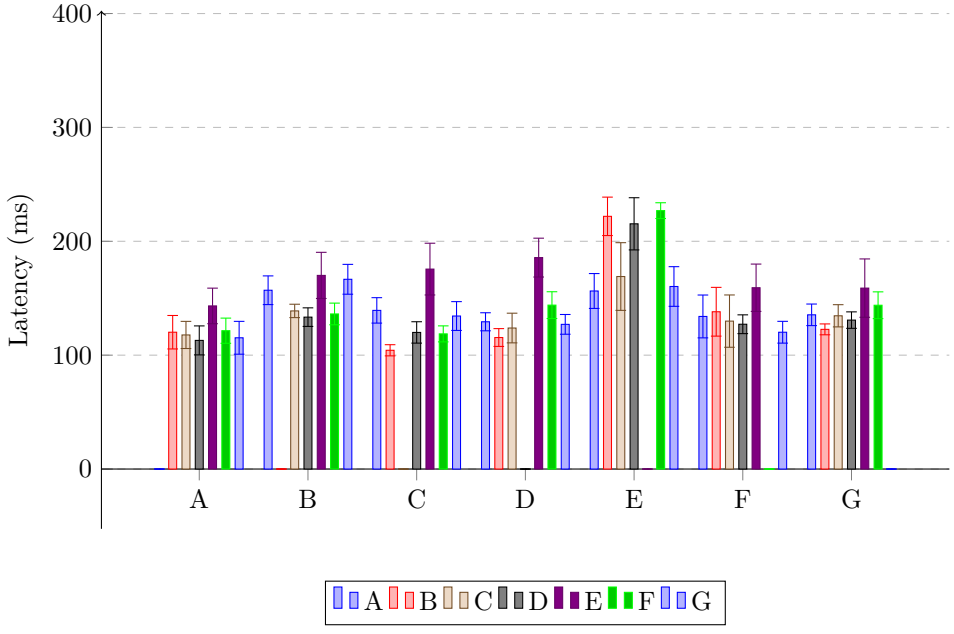


(b) Chrome

Figure G.5: Observed bitrates with seven people, no traffic shaping



(a) Firefox



(b) Chrome

Figure G.6: Observed latencies with seven people, no traffic shaping