Performance analysis of adaptive streaming algorithms for a low-latency environment

ALBERT SJÖLUND & ANTONIO KEVO

MASTER´S THESIS

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY

FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Performance analysis of adaptive streaming algorithms for a low-latency environment

Albert Sjölund, Antonio Kevo
`al5518sj-s@student.lu.se`
`an8006ke-s@student.lu.se`

Department of Electrical and Information Technology
Lund University

Supervisor: Johan Sternerup, David Svensson Fors, Emma Fitzgerald

Examiner: Maria Kihl

June 17, 2022

© 2022
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Streaming video over the internet can face issues when met with poor and varying network conditions, which can be especially noticeable when streaming live video such as security footage or video calling. To handle this, there exists congestion control algorithms that monitor network conditions based on feedback from the receiver and adapt the video output. This thesis aims to implement and compare three different algorithms NADA, SCReAM and GCC as part of Axis' video streaming system and IoT camera, and to perform a performative analysis in both environments.

To perform the analysis, a set of realistic situations are simulated inside the same framework that is used in production, as well as implementing and testing out a *Proof on Concept* directly on camera hardware. The algorithms are implemented into Axis own WebRTC system, and run inside a container system with networking tools to monitor performance. A similar networking performance test is run directly on camera hardware to test out how it behaves on real hardware.

This thesis concludes that the NADA algorithm is an ideal choice when there is not a lot of latency present, having great utilization of the available link. However, in the presence of non-constant network delay it, together with SCReAM fails to utilize the link and only GCC can maintain a proper sending rate; GCC is shown to be a very good general-purpose algorithm. SCReAM shows a constant lower utilization of the available link, matching its target use of mobile networking and with the best measure of round-trip time in low-latency tests it is a good fit for remote-controlled devices.

Popular Science Summary

Congestion, be it traffic from cars or from computers, is an undesirable thing. It causes slowdowns in the network and reduces the quality of service, which can compromise performance. In order to avoid this, the network can be monitored in order to look for possible signs of congestion, and regulate the data. This is especially important for real-time media, such as a live stream, as it both sends large amounts of data and when congested can show visible lag. For real-time media, congestion control algorithms can monitor the congestion present in the network, and adjust how much data may travel through it.

This thesis aims to test out how viable congestion control algorithms are inside **WebRTC**, a web standard for real-time communication, to perform dynamic changes to video quality. These tests are evaluated at Axis Communications, both as a simulation and on camera hardware.

In order to evaluate the performance of these congestion control algorithms, a simulation system is created inside Axis' in-house real-time platform WebRTC, to see how they perform in different network scenarios. These scenarios test a complete loss of network, a drop in available network and test where the latency is high. This system is used to evaluate the congestion algorithms called **GCC**, **SCReAM** and **NADA**. All three algorithms are used to monitor the network and estimate how high quality the video stream can be without causing congestion. These simulations are complemented with a test running on an Axis surveillance camera, to test the performance on real video hardware. For the camera, the NADA algorithm is tested with live video generated from the camera.

The results showed how GCC is able to keep a reliable video even in tests where high latency is introduced. NADA, however, is a well performing algorithm during changes in the network capacity when no additional latency is added, having a very fast response to these drops. The tests run very well on the camera, being able to properly react to changes in the network. However, this thesis seeks only to perform an initial evaluation, and optimization is the next goal for congestion control in Axis' cameras.

Acknowledgements

Many thanks to Vladimir Karadzic, David Svensson Fors, and Johan Sternerup for supervising the thesis work, assisting us with the implementation of the algorithms by sharing knowledge about how WebRTC and GStreamer operate, and guiding us throughout the thesis work through weekly meetings. We would also like to acknowledge the help provided by the Core Technologies team, where they assisted us with allowing the algorithms to directly change the bitrate of the hardware encoder. Lastly, I'd like to mention Maria Kihl and Emma Fitzgerald from the Department of Electrical and Information Technology at LTH, for assisting us with feedback and partial guidance with the thesis writing.

Table of Contents

List of Figures

List of Tables

Definitions

- **Protocol -** Set of common rules used to unify communication over a link, given that both sides apply the same rules
- **Packet -** A segment of a larger message sent as one unit over the network
- **Header -** The initial part of any data, commonly used to clarify what comes next
- **Payload -** The actual data part of a packet, excluding headers and metadata
- **Octet -** 8 bits of data, used as alternative phrasing when byte is ambiguous
- **Padding -** Bits of data added at the end of a certain segment, used to align the larger data segment to a certain format
- **Bit -** A singular 1 or 0
- **Byte -** A group of 8 bits
- **Bitrate -** Bitrate is the number of bits transferred, usually over the span of a second. For instance, a bitrate of 800 kBps would mean 800 thousand bits are transferred each second.
- **Thread -** A distinct execution of code, running in parallel to other threads.

Chapter 1 Introduction

Network congestion is a term describing a reduction in the Quality of Service (QoS) which can typically occur when there is too much traffic being handled, and it exceeds the capacity of a node [1]. A similar analogy can be made to traffic congestion in highways, where a highway is congested when it has to manage more vehicles than its capacity [1]. This leads to traffic jams where the traffic moves very slowly, leading to longer trip times and increased vehicular queuing as the traffic on the road builds up. Similar effects occur in networks such as queuing delay, packet loss, and blocking of new connections [1]. Furthermore, if the data continues to be pushed into a node or link during a congestion, it can prevent or limit useful communication. This is known as congestion collapse [2]. One of the ways of handling network congestion is through the use of congestion control algorithms, which control how much data may travel through the network while avoiding congestion collapses [3]. Congestion control for real-time multimedia continuously monitor the stream and use information derived from it to adapt the streaming properties.

1.1 Problem Statement

Congestion control is required for all data transported across the Internet in order to promote fair usage [3]. However, the requirements for real-time multimedia are different compared to the requirements for the bulk transfer such as FTP or bursty transfers like web pages [3]. The algorithms are required to transmit data continuously within a very limited time window, while still providing a useful amount of bitrate [3]. They must be resilient to the effects of events, such as limitation of bandwidth or introduction of delay [3]. They must be able to react quickly to handle both local and remote interface changes (e.g. WLAN to 3G data) where the bandwidth available can change radically [3]. The algorithms should also be able to adapt the rate of media applications depending on the available bandwidth, and be *fair* with the other traffic flows, for example TCP [3]. As a result of an increasing amount of RTP-based real-time media traffic on the Internet, it is now more important than ever to ensure that this type of traffic is properly congestion controlled [3]. An example of this are surveillance cameras that makes use of Web Real-Time Communication (WebRTC) to transfer data between itself and a server. Depending on the network setup, the network can be subject to various network constraints, leading to congestion. These constraints vary from setup and setup, making it difficult to predict how a data transmission will behave. This is especially the case when data is transmitted outside a local network, where the data can be transmitted through the network in numerous ways. However, a surveillance camera is required to continuously transmit data through the network despite the existence of network constraints. Thus, the use of congestion control in surveillance cameras is essential.

1.2 Goal

The goal of this thesis is to evaluate how various congestion control algorithms operate within WebRTC, to analyze usability of algorithms for both regular cameras and cameras which impose a requirement of high responsiveness and low latency e.g. PTZ (Pan-Tilt-Zoom) cameras. This includes investigating how the algorithms operate when subjugated to various network conditions such as bandwidth limit, delay, and jitter. The thesis work took place at Axis Communication AB ("Axis") in Lund, Sweden. Axis is a leading manufacturer of security solutions, including Internet of Things (IoT) surveillance cameras [4]. The company was founded in Lund 1984 and today has over 3800 employees across 50 countries [4]. The algorithms are also implemented into an Axis surveillance camera that makes use of WebRTC, as a *Proof of Concept.*

We have formulated the following research questions based on the goals above:

- RQ_1 - Which congestion control algorithm is best suited for a WebRTC environment with regard to utilization, latency and responsiveness?
- RQ_2 - Can the algorithms be implemented and used in an Axis surveillance camera?

1.3 Limitations

In this thesis, three algorithms are evaluated. While there are many other methods of performing congestion control in real-time media, including other algorithms, these three were chosen both to keep the scope reasonable and for the criteria that they meet. Also, methods excluding congestion control algorithms such as Forward Error Correction or RTP Retransmission (RTX) are among some other methods that are not evaluated in this thesis.

1.4 Contributions

The results of this thesis will build upon work on the topic of congestion control for real-time media with a focus on improving not just bandwidth but latency for the real-time requirements of security streaming. The thesis shall offer insight into how adaptive streaming behaves under various network conditions and measure the efficiency of different implementations that make use of WebRTC. Differing from other implementations is the requirements induced by the application, be it an IoT surveillance camera, security monitoring or similar; ensuring optimizations not only prioritize bandwidth but also minimizing latency simultaneously.

1.5 Thesis Overview

The thesis paper is structured into following chapters, in order:

1. **Related Work** - Introduces previous work related to this thesis in order to provide context of what was done prior to this thesis.
2. **Technical Background** - Includes a background into protocols, algorithms, and tools used during the thesis.
3. **Approach** - Provides context into how the problems were approached, how the algorithms were implemented, as well as the thought process behind the decisions done throughout the thesis.
4. **Evaluation** - Describes the experimental setup, the results reached with the setup, and the implication of the said results.
5. **Conclusion** - Concludes the thesis by answering the stated research questions and provides future work that builds upon the content in this thesis to explore further research topics.

Chapter 2

Related Work

This chapter mentions and details work related to this thesis, how they differ and what the papers cover. Section 2.1 will discuss the various research papers regarding the congestion control algorithms for real-time media. Section 2.2 will focus on the concept of induced demand and its relevance to this thesis. Finally, Section 2.3 will provide an overview of existing studies regarding congestion control unrelated to adaptive bitrating and WebRTC. Compared to previous work done regarding congestion control for real-time media, most research studies utilize a simulated network environment while this thesis utilizes a production build by Axis. This shows how the algorithms would behave when applied to an existing application used by consumers. Furthermore, there is a general lack of surveys that focuses on congestion control for real-time media due to it being a relatively new area.

2.1 Congestion Control Algorithms

There are many papers that focus on congestion control algorithms for real-time media, often comparing them in a simulated network environment. These algorithms are commonly the ones used in this paper, NADA, SCReAM, GCC. The paper [5] analyzes the performance of Google Congestion Control when subject to high round-trip times and possible packet losses, but focuses only on this algorithm. It also showcases the benefits of different video codecs, which is something relevant but not explored in this thesis. The design paper [6] mentions how Google Congestion Control was intended for use in WebRTC specifically. A similar study [7] to this thesis compares the three algorithms and comes to conclusions how each of them behave. Like [5] and [6], it uses a simulated testbed, using the NS Network Simulator as the framework to evaluate the different algorithms. A paper [8] also compares congestion control algorithms, in this case only NADA and GCC. However, it instead uses the Chromium network simulator to evaluate the algorithms inside a web-browser, rather than the NS network simulator framework. Furthermore, a thorough evaluation of GCC [9] was performed over a testbed with up to 4 real computers via the WebRTC framework. It thoroughly analyzed the performance of GCC but does not perform a comparison to other algorithms. A paper [10] presents an improvement to the GCC algorithm to avoid unnecessary reduction in bitrate. This thesis analyzes algorithms with little internal configuration to the algorithms themselves (changing parameters), instead compares their individual performance. However, [10] finds issues with how GCC behaves in a low-latency environment, which is relevant to the analysis of this thesis. Furthermore, [10] runs its analyses in a web-browser (Chromium) rather than a network simulator. One paper [11] analyzes the performance of multiple SCReAM streams in order to operate remote machinery over LTE. Rather than a simulated network testbed it is tested on a Raspberry Pi mounted on a vehicle. This thesis instead compares multiple algorithms on an emulated production testbed, rather than a single one on a moving vehicle.

There are other algorithms used to perform adaptive streaming via congestion control, such as FRACTaL [12]. It is compared against SCReAM and shows it performs better in terms of fair use and in lossy links. The FRACTaL algorithm is not used in this thesis, although SCReAM is used as a part of the approach.

A paper [13] analyzes the performance of coupling a congestion control algorithm (NADA) with a controller for Stream Control Transmission Protocol (SCTP) [14] to help with the issues derived from SCTP. A follow-up article pairs and analyzes the aforementioned FSE-NG controller for SCTP with SCReAM [15], likewise with the goal of minimizing end-to-end delay to enable real-time communication. However, for both of these articles, SCTP is a separate internet protocol from UDP with different properties and is not specifically relevant to this thesis. Though, optimizing end-to-end delay is an important factor for this thesis' analysis.

As there is a complicated relationship between input variables from the network and the output from the algorithms, one paper [16] discusses the challenges of using Machine Learning as a base for calculating congestion windows as well as other parameters. The tested algorithms in this paper are built using control theory and similar areas rather than being based on Machine Learning.

2.2 Induced Demand

A separate area of interest for congestion control algorithms is the concept of induced demand. Induced demand is related to traffic as a broad concept, generalizing over both physical ideas of traffic such as road traffic while being applicable to more theoretical applications, e.g. network traffic. Traffic on roads tends to maintain their equilibrium, implying that an increased capacity only provides a short-term solution to congestion [17]. Therefore, it shows how increased capacity isn't always enough to handle congestion; which, relatedly, showcases how results in congestion control can be applicable even with technological improvements. A study [18] measured the rollout of LTE networks in South Korea and showed empirical data of how the positive feedback from an increase in capacity showed a gain in the used consumer capacity, above the theorized increase. This difference in demand was pointed to the positive feedback caused by increased capacity. Therefore, the area of congestion control studied by this thesis has relevance even with improvements to the capacity and utility of network infrastructure.

A lot of analysis has been done in recent years comparing different types of congestion control algorithms, with varying configurations. However, these are either specific algorithms, a smaller scope and/or a different type of testbed. Furthermore, this thesis aims to test out algorithms ideal for a low-latency environment, for instance a controllable camera. This is relevant for Axis infrastructure as well as an important comparison between congestion control algorithms.

2.3 Congestion Control outside WebRTC

There is also a great deal of research being conducted on congestion control outside WebRTC and adaptive bitrating in general. A survey [19] focused on comparing various congestion control mechanisms used in Wireless Sensor Networks (WSNs) in terms of congestion detection, congestion notification and congestion mitigation. These mechanisms were then classified into four different categories. The survey also introduced directions on researches and works that could be done in the future. [20] performed a survey on congestion control for Constrained Application Protocol (CoAP) over User Datagram Protocol (UDP). They found that CoAP itself has to support congestion control when running over UDP for WSNs. The basic mechanism for congestion control in CoAP is to measure the retransmission timeout without estimating round-trip-times. Thus, the majority of the efforts to improve congestion control have focused on measuring round-trip-time more accurately and make retransmission timeout more adaptive to network conditions. [21] performed a survey for congestion control for delay and disruption tolerant networks (DTNs), as the networks are often subjected to high latency caused by very long propagation delays and/or intermittent connectivity. They managed to draw conclusions from the survey e.g. many of them tried to increase successful message delivery while decreasing delivery delay and keep network utilization high without congesting it, and attempted to resolve congestion that may result from replication-based message forwarding.

While there exists a lot of surveys on congestion control that have been done for over a decade, there is a general lack of it that focuses on real-time media such as WebRTC. This is due to congestion control for real-time media being a relatively new area, that has just recently started to being used in existing products and applications which utilize real-time media.

Chapter 3

Technical Background

In this chapter, the technical background and concepts required to understand the approach and evaluation of this thesis are explained. Section 3.1 will provide an overview of the various network protocols used in this thesis. Section 3.2 gives a rundown of the congestion control algorithms used in the thesis, while Section 3.3 focuses on defining latency and jitter, and how they compare to each other. Finally, Section 3.4 provides an overview of the software tools used throughout the thesis.

3.1 Network Protocols

This section will focus on providing an overview of the various network protocols, and how they are used in this thesis.

3.1.1 UDP

User Datagram Protocol (UDP) is a transaction-oriented transport-layer protocol that provides a procedure for applications to send messages, also known as *datagrams*, to other applications with a minimum amount of protocol mechanisms [22]. In this thesis, UDP packets are used as wrappers when transferring RTP/RTCP packets. UDP provides best-effort delivery [23] meaning that it does not provide any guarantees that data is delivered or that delivery meets any QoS. This also means that there are no special features that recover lost packets or restore corrupted ones [24].The model does not have any requirements in maintaining any logical relationship among multiple data units [25], which can lead to packets not arriving in a predetermined order.

3.1.2 RTP

Real-time transport protocol (RTP) is a network protocol designed to provide end-to-end delivery services for data with real-time characteristics, such as interactive audio and video [26]. These services include payload type identification, sequence numbering, timestamping and delivery monitoring [26]. It is the primary network protocol to transfer video in WebRTC. RTP does not provide any mechanisms to ensure timely delivery, nor does it guarantee QoS. Instead, it depends on the lower-layer networks to provide these mechanisms [26]. It does not guarantee delivery or prevent out-of-order delivery, nor does it assume that the underlying network is reliable and delivers packets in sequence [26]. Because of this, the RTP protocol contains timing information and a sequence number to reconstruct the sender's packet sequence [26]. RTP is designed to be flexible in order to provide the information needed by a particular application and will often be integrated into the application processing rather than being implemented as a separate layer [26]. It is also intended to be tailored for specific tasks through modifications and/or additions to the header as needed [26].

3.1.2.1 RTP Header Extension

A header extension is provided in order to allow individual implementations to experiment with new payload-format-independent functions that require further information to be carried to the RTP packet header [26]. This feature is designed so that the header extension may be ignored by other interoperating implementations that have not been extended to support the extension [26]. The header extension is formed as a sequence of extension elements with possible padding. Each extension element consists of a local identifier, a length and data. There are two types of header extensions: one-byte header, and two-byte header, as seen in Figures 3.1 and 3.2 respectively. The two types are defined by the patterns `0xBEDE` and `0x1000` respectively in the *defined by profile* field. Depending on the header extension type, the size of extension elements can vary [27]. For one-byte headers, the extension elements are one octet long with the identifier and length field being 4 bits each. For two-byte headers, the elements are 2 octets long, with the identifier and length fields being twice the size. 355.65944pt

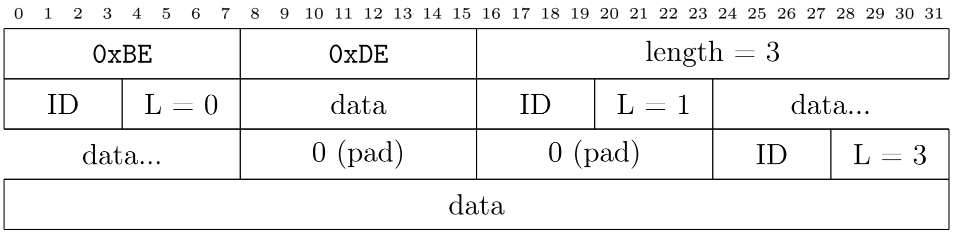


Figure 3.1: Example of one-byte header extension, with three extension elements and some padding [27].

3.1.3 RTCP

RTP control protocol (RTCP) is a protocol that works alongside RTP to monitor data delivery in networks [28]. While RTP is responsible for transmitting data packets, RTCP is responsible for monitoring QoS and periodically transmitting control packets to all participants in an RTP session using the same distribution mechanism as the data packets [26]. RTCP performs four functions:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

0x10	0x00	length = 3	
ID	L = 0	ID	L = 1
data	0 (pad)	ID	L = 4
data			

Figure 3.2: Example of two-byte header extension, with three extension elements and some padding [27].

1. The primary function is to provide feedback on the quality of the data distribution. This is an essential part of the RTP's role as a transport protocol and is related to the flow and congestion control functions of other transport protocols. [26]
2. RTCP carries a persistent transport-level identifier for an RTP source called the canonical name or CNAME which is used in case the SSRC identifier is changed due to conflicts or program restarts. Receivers may also require the CNAME to associate multiple data streams from a given participant in a set of related RTP sessions, for example to synchronize audio and video. [26]
3. Because the first two functions require that all users send RTP packets, the rate has to be controlled in order for RTP to scale with the number of users. Thus, each user sends its own control packet to each other in order to independently identify the number of users in the session, which is later used to calculate the packet rate. [26]
4. An optional function that conveys the minimal session control information. It is useful in *loosely controlled* sessions where participants can enter and leave without membership control or parameter negotiation. [26]

It is recommended that functions 1-3 are used in all network environments [26].

3.1.4 Transport-wide Congestion Control

Transport-wide Congestion Control (TWCC) is an RTP header extension and an RTCP feedback message used in congestion control algorithms for RTP-based media flows [29]. The RTP header extension provides a transport-wide sequence number while the RTCP feedback message feeds back arrival times and sequence numbers of the packets received on a connection [29].

3.1.4.1 RTP Header Extension Format

The RTP header extension is added onto the transport layer, and uses the same counter for all packets sent over the same connection [29]. This is a better fit for congestion control since the congestion controller does not have to operate on media streams, but on packet flows. In addition, it allows for earlier packet loss detection (and recovery) since a loss in stream A can be detected when a packet from stream B is received. Thus, there is no need to wait until the next packet of stream A is received. As seen in Figure 3.3, the header extension is a one-byte header defined by the 16-bit header pattern `0xBEDE` [27]. Since the transport-wide sequence number is 16-bits, L is set to one. The sequence number is attached to every packet sent is incremented by one for each packet being sent over the same socket.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

0xBE		0xDE	length = 1	
ID	L = 1	transport-wide sequence number		0 (pad)

Figure 3.3: TWCC RTP header extension [29].

3.1.4.2 Transport-wide RTCP Feedback Message

In order to allow as much freedom as possible to the sender, it requires information about each packet that got delivered successfully [29]. The simplest way to achieve this is to have the receiver send back messages containing an arrival timestamp and a packet identifier for each packet received [29]. This makes the receiver dumb as it is only responsible for recording arrival timestamps (A) of each packet [29]. The sender keeps a map of in-flight packets, and upon feedback arrival, it looks up the on-wire timestamp (S) of the corresponding packet [29].

Since the sender receives feedback about each packet sent, it will be set to better assess the cost of sending bursts of packets compared to aiming at sending at a constant rate decided by the receiver [29]. This does create a few downsides such as not being able to differentiate between lost feedback on the downlink and lost packets on the uplink [29]. It also increases the feedback rate on the reverse direction. Similarly to how lost RTCP receiver reports are handled, lost feedback messages are handled by ignoring packets that would have been reported as lost or received in the lost feedback messages [29]. Usually feedback messages should be sent for every frame received, but in cases of low uplink bandwidth it is acceptable to send them less frequently such as once per round-trip-time, in order to reduce overhead [29].

Message Format Figure 3.4 shows message format for the RTCP message. The first 8 octets are similar to other RTCP messages, with payload type (PT) 205 identifying it as a transport layer feedback message. The message contains two 32-bit SSRCs, one for the packet sender and one for the media source. This is followed by two 16-bit fields which contains the transport-wide sequence number of the first packet in this feedback (base sequence number), and the number of packets this feedback contains status for (packet status count), starting with the packet identified by the base sequence number. Reference time, a 24-bit signed integer, indicates an absolute reference time in some (unknown) time base chosen by the sender of the feedback packets. The value has to be interpreted in multiples of 64 ms [29]. The feedback packet count is an 8-bit field that counts the number of feedback packets sent, and is used to detect lost feedback packets [29]. Afterwards is a list of packet status chunks, each 16-bits long, that indicates the status of a number of packets starting with the one identified by base sequence number [29]. Finally, there is a list of receive delta fields, one for each packet that got the *Packet received* status (see details below).

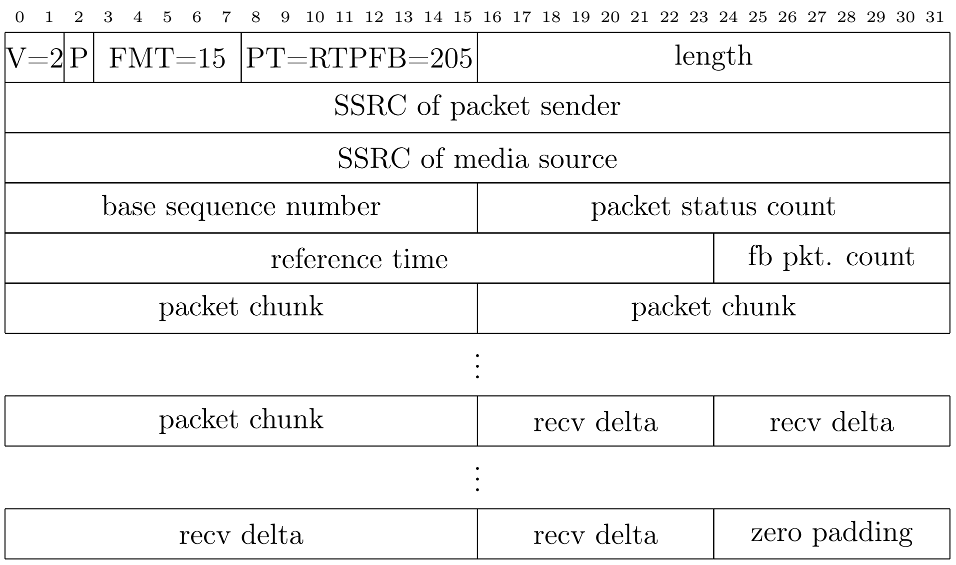


Figure 3.4: TWCC RTCP feedback packet [29].

Packet Status Symbols The status of a packet is described using 2-bit symbols:

- `00` - Packet not received
- `01` - Packet received with small delta
- `10` - Packet received with large or negative delta

Note that packets with `00`-status should not necessarily be interpreted as lost [29]. They might just not have arrived yet. For each received packet, a receive delta block is appended to the feedback message if the packet delta, to the previous received packet, is within ±8191.75 ms [29]. Deltas are represented as multiples of 250 μs [29]. If the `01`-symbol has been appended to the status list, an 8-bit unsigned receive delta will be appended to the receive delta list, representing a delta in the range [0, 63.75] ms [29]. For the `10`-symbol, a 16-bit signed receive delta will be appended instead, representing a delta between [-8192.0, 8191.75] ms [29].

3.1.4.3 Packet Status Chunks

Packet status' are bundled together in a packet chunk of 16-bits in length [29]. There are two kinds of packet chunks: run length chunk, and status vector chunk [29], as seen in Figures 3.5 and 3.6. The first bit (T) is used to determine the type of the chunk [29]. In run length, this is followed by a 2-bit field (S) determining the packet status, and finally, a 13-bit field containing the run length [29]. Run length determines the number of packets with the determined packet status (S) [29]. For the status vector, S is only 1-bit, followed by a 14-bit symbol list [29]. S is used to determine the symbol size [29]. If set to zero, the status vector can only contain *packet received* (0) and *packet not received* (1), essentially ignoring the receiving deltas [29]. This means that each status symbol is compressed down to one bit, allowing the symbol list to contain up to 14 packet status' [29]. A one means the status vector contains the normal 2-bit status symbols, allowing the list to have up to 7 packet status' [29].

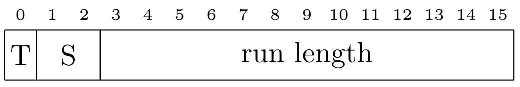


Figure 3.5: Run length chunk packet format [29].

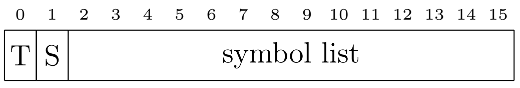


Figure 3.6: Status vector chunk packet format [29].

3.2 Congestion Control Algorithms

This section covers the congestion control algorithms that are being evaluated in this thesis, how they operate, and the properties they exhibit.

3.2.1 Google Congestion Control

Google Congestion Control (GCC) is a congestion control algorithm compliant with the WebRTC framework [6]. It was designed to work with RTP/RTCP protocols and infers congestion by using delay gradient [6]. The algorithm uses two components: a loss-based controller, and a delay-based controller. The delay-based controller is responsible for computing the bitrate A_r that is fed back to the sender with the aim of containing the delay [6]. The loss-based controller is responsible for receiving A_r and compute A_s based on packet loss such that it does not exceed A_r [6]. The loss-based controller runs on the sender-side while the delay-based controller runs on the receiver-side. However, there is no requirement for the delay-based controller to run on the receiver-side, thus the delay-based controller may run on the sender-side along with the loss-based controller [30].

Figure 3.7 shows the general architecture employed by the algorithm. The encoder is responsible for compressing digital video to a bitrate as close as possible to the target bitrate A [6]. The pacer acts like a queue which sends a group of packets to the network within a set interval [30]. Finally, the padder is used to correct frames that have a bitrate lower than A. Note that the receiver can also send feedback using TWCC feedback messages rather than RTCP feedback messages containing Receiver Estimated Maximum Bitrate (REMB) [30].

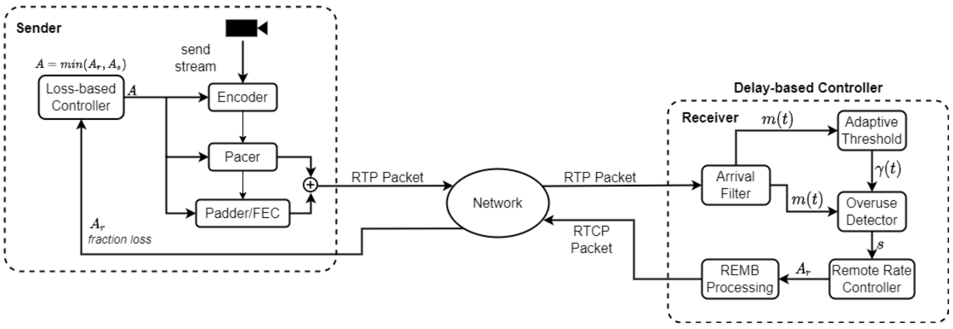


Figure 3.7: Google Congestion Control architecture [6].

3.2.1.1 Delay-based Controller

On the receiver-side, the delay-based control algorithm is responsible for computing A_r using the following equation [6]:

$$A_r = \begin{cases} \eta A_r(t_{i-1}) & \sigma = \text{Increase} \\ \alpha R_r(t_i) & \sigma = \text{Decrease} \\ A_r(t_{i-1}) & \sigma = \text{Hold} \end{cases} \tag{3.1}$$

t_i denotes the time that video frame i is received, $\eta = 1.05$, $\alpha = 0.85$, σ represents the current state in the finite state machine (FSM), and $R_r(t_i)$ is the receiving rate measured in the last 500ms. The delay-based control algorithm can be further decomposed into several subcomponents: a pre-filtering, an arrival-time filter, an adaptive threshold, an over-use detector, a remote rate controller, and a REMB-processing.

Pre-filtering In GCC, Pre-filtering is responsible for managing transient delay caused by channel outages [30]. During an outage, packets are queued into network buffers, for reasons unrelated to congestion, and are delivered in a burst when the outage ends [30]. Pre-filtering groups packets together that arrive in a burst if one of the following conditions holds [30]:

- A sequence of packets sent within a burst time interval constitute a group. [30]

- A packet that has an inter-arrival time less than the burst time as well as an inter-group delay variation $d(i)$ less than zero is considered being part of the current group of packets. [30]

Arrival-time Filter The goal of this block is to produce an estimate of $m(t_i)$ of the one way delay gradient [6]. This is done through the use of a Kalman filter, with the measured one way delay gradient $d_m(t_i)$ as its input [6]. It is computed as follows (see Figure 3.8):

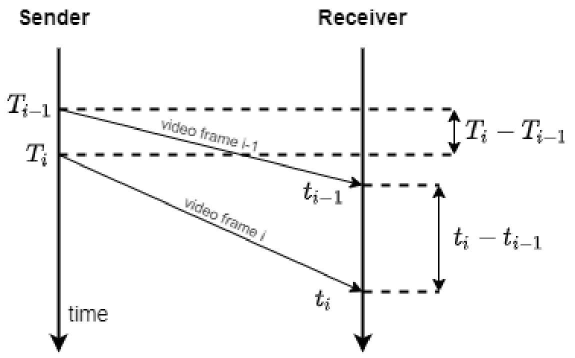


Figure 3.8: One way delay gradient measurement [6].

$$d_m(t_i) = (t_i - t_{i-1}) - (T_i - T_{i-1}) \tag{3.2}$$

T_i is the time at which the first packet of video frame i has been sent and t_i is the time at which the last packet that forms the video frame has been received. Figure 3.9 shows how the estimate of $m(t_i)$ is computed: at each step, $z(t_i)$ is multiplied by the Kalman gain $K(t_i)$ which provides the correction to the estimation $m(t_i)$ according to:

$$m(t_i) = (1 - K(t_i)) \cdot m(t_{i-1}) + K(t_i) \cdot (d_m(t_i)) \tag{3.3}$$

Adaptive Threshold This subcomponent is responsible for adapting the algorithm sensitivity to delay variations based on network conditions [6]. The threshold $\gamma(t_i)$ has to be adaptive otherwise two issues can occur:

1. The delay-based control action may have no effect when the size of the bottleneck queue along the path is not sufficiently large. [6]
2. The GCC flow may be starved by a concurrent loss-based TCP flow. [6]

The adaptive threshold $\gamma(t_i)$ is computed as follows:

$$\gamma(t_i) = \gamma(t_{i-1}) + \Delta T \cdot k_\gamma(t_i)(|m(t_i)| - \gamma(t_{i-1})) \tag{3.4}$$

t_i is the time instant that video frame i is received, and $\Delta T = t_i - t_{i-1}$. The gain $k_\gamma(t_i)$ is defined as follows:

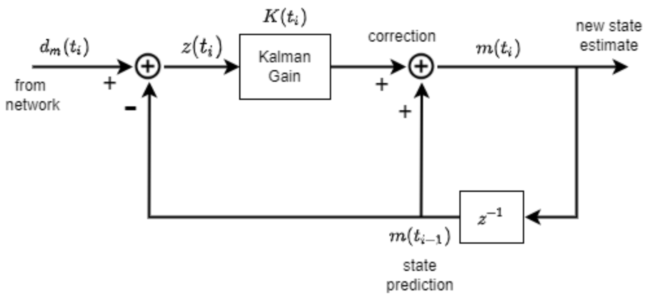


Figure 3.9: State estimation with a Kalman filter [6].

$$k_\gamma(t_i) = \begin{cases} k_d & |m(t_i)| < \gamma(t_{i-1}) \\ k_u & \text{otherwise} \end{cases} \tag{3.5}$$

Here, k_u and k_d determine the speed at which the threshold is increased or decreased, respectively [6].

Over-use Detector Every time t_i a video frame is received, the over-use detector produces a signal s based on $m(t_i)$ and the threshold $\gamma(t_i)$, which is used to drive the FSM that determines the state σ [6]. When $m(t_i) > \gamma(t_i)$, the algorithm starts to track the time spent in this condition and if it is greater than 100ms the *overuse* signal is generated [6]. On the other hand, if $m(t_i) < -\gamma(t_i)$, an *underuse* signal is generated, whereas a *normal* signal is generated if $-\gamma(t_i) \leq m(t_i) \leq \gamma(t_i)$ [6].

Remote Rate Controller The remote rate controller computes A_r based on (3.1) using the signal s that drives the FSM [6]. The FSM is used to minimize the queuing delays in the buffers along the end-to-end path [6]. Figure 3.10 shows how the FSM changes based on the s and the current state σ. Assuming the FSM starts on the *hold* state, when the bottleneck buffers start to build up, the estimated one way delay gradient $m(t_i)$ becomes positive [6]. The over-use detector detects this and triggers an *overuse* signal, setting the FSM to a *decrease* state [6]. As a result, the sending rate is reduced and the bottleneck buffers starts to be drained, up to the point that $m(t_i)$ becomes negative [6]. An *underuse* signal is detected and FSM is set back to *hold* state until the bottleneck buffer is emptied [6]. When this occurs, $m(t_i)$ is nearly zero and the over-use detector generates a *normal* signal, which drives the machine into the *increase* state [6].

REMB Processing This block notifies the sender with the computed rate A_r through RTCP feedback messages. The feedback messages are either sent every second, or when A_r has decreased by more than 3% [6].

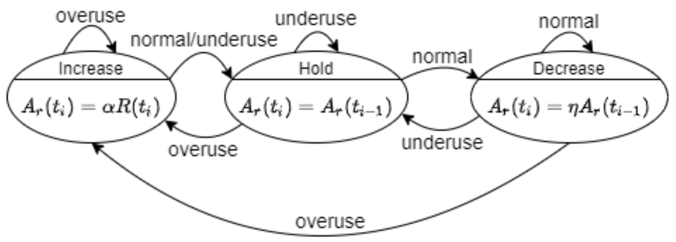


Figure 3.10: Remote rate controller finite state machine [6].

3.2.1.2 Loss-based Controller

The loss-based controller complements the delay-based controller whenever losses are detected [6]. This occurs every time t_k that the RTCP message k carrying the bitrate A_r is received [6]. From the RTCP message we compute the fraction of lost packets $f_l(t_k)$ and use it to compute the sending rate $A_s(t_k)$ according to the following equation [6]:

$$A_s(t_k) = \begin{cases} A_s(t_{k-1})(1 - 0.5f_l(t_k)) & f_l(t_k) > 0.1 \\ 1.05(A_s(t_{k-1})) & f_l(t_k) < 0.02 \\ A_s(t_{k-1}) & \text{otherwise} \end{cases} \tag{3.6}$$

When the fraction is considered small ($0.02 \leq f_l(t_k) \leq 0.1$), A_s remains constant [6]. If a high fraction lost is estimated ($f_l(t_k) > 0.1$), A_s decreases multiplicative, whereas a low fraction lost ($f_l(t_k) < 0.02$) makes A_s increase multiplicative due to being considered negligible [6].

3.2.1.3 Sending Rate Actuation

After computing A_s, the target bitrate A is computed as the minimum between A_s and A_r and fed to both the encoder and the pacer as shown in Figure 3.7. The encoder aims to produce a bitrate as close as possible to A [6]. However, the encoder cannot change the rate as frequently as pacer's rate as it can cause video quality flickering [6]. Thus, the encoder may not produce a bitrate that precisely matches A [6]. If the encoder produces a bitrate larger than A, the pacer is allowed to drain its queue at a rate of $f \cdot A$, where f is the pacing factor of 1.5 [6]. This way, the pacer can quickly empty its queue in order to avoid queuing delays at the sender [6]. On the other hand, whenever the pacer produces a bitrate lower than A, padding or forward error correction (FEC) might be added [6]. As a result, an average sending bitrate equal to A is produced [6]. In addition, FEC can be used with am RTP retransmission stream as a bitrate stabilizer by sending additional data to determine if additional link capacity exists [31].

3.2.2 Network-Assisted Dynamic Adaptation

Network-Assisted Dynamic Adaptation (NADA) is a novel congestion control scheme designed by CISCO [32]. NADA is designed to act based on the existence of implicit congestion control signals (delay and/or loss) but can incorporate explicit markings as well, for instance Explicit Congestion Notification (ECN) [32]. Furthermore, it is a rate-based algorithm and acts on information received from the aggregate congestion signal [33, 32]. Similarly to other congestion control algorithms, it works based on monitoring RTP packets sent from the sender as well the RTCP reports returned from the client to dynamically estimate information regarding available bandwidth in the network [32].

NADA operates in two modes based on the aggregate congestion signal. (The collection of all input signals from the RTP/RTCP system)

- **Accelerated ramp up** - If the link is deemed underutilized the rate is multiplicative increased based on a factor γ; however, round trip time and feedback interval is taken into account to avoid causing queue delays. (R-mode 0) [32]

- **Gradual rate update** - When the aggregate congestion signal is non-zero the rate is adjusted based on two factors, the current value as well the difference in value. (R-mode 1) [32]

When receiving a report from the client, collect current system time as well as the input data from the report, including the receivers timestamp [32]. Afterwards, the feedback delay

$$\delta_{fb} = t_{curr} - t_{last}$$

is calculated, alongside an estimation of the round-trip time (RTT) [32]. Now, depending on the mode it updates the reference value via either a multiplicative calculation or a difference in value [32]. Lastly, the reference rate value is clipped between the upper and lower bounds [32].

NADA specifies the algorithm to perform on the receiver side as well [32]. As with similar sender-based algorithms the receiver is treated as dumb and is mostly responsible for generating relevant information to send to the server via feedback reports [32]. The statistics monitored include delay, loss and/or ECN markings which make up the aggregate congestion signal [32]. However, to simplify development NADA has been bridged together with the feedback from Transport Wide Congestion Control (TWCC) which generates and sends feedback reports to the sender, creating an aggregate congestion signal from these reports [32]. To facilitate rapid testing, an instance of NADA derived from Firefox (Gecko) [34] has been implemented as a plugin usable without any requirements on the client other than TWCC.

3.2.3 Self-Clocked Rate Adaptation for Multimedia

Self-Clocked Rate Adaptation for Multimedia (SCReAM) is a congestion control algorithm designed for RTP streams [33]. While it was originally devised for WebRTC, the algorithm can be used by any application where congestion control of RTP streams is necessary [33]. The algorithm consists of three parts: network congestion control, sender transmission control, and media rate control. These parts all reside on the sender-side while the receiver-side is rather simple in comparison, as it is only responsible for generating RTCP feedback packets containing acknowledgements of received RTP packets and an Explicit Congestion Notification (ECN) count [33].

3.2.3.1 SCReAM Sender

The sender-side of the algorithm, also known as SCReAM sender is responsible for sending RTP packets to the receiver-side (SCReAM receiver), receiving RTCP packets from the SCReAM receiver and adjust the bitrate accordingly using the media rate controller [33]. As we can see in Figure 3.11, the video frames are encoded and forwarded to the RTP queue (1), where the media rate controller adapts to the size of the RTP queue (2) and provides a target rate for the media encoder (3) [33].

The RTP packets from the queue are forwarded to a sender transmission controller (4) which limits the frames so that the number of bytes in flight is less than the congestion window [33]. Finally, the RTP packets are forwarded to a UDP Socket and sent to the SCReAM receiver (5) [33]. The RTCP packets are received (6) and the information about the bytes in flight and congestion window is exchanged between the network congestion control and the sender transmission control (7) [33].

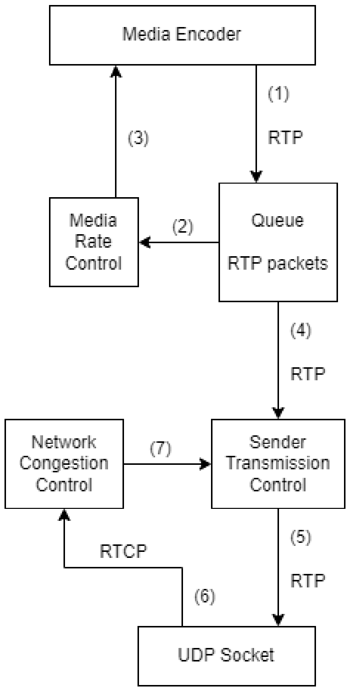


Figure 3.11: Internal Structure of the SCReAM sender [33].

3.2.3.2 Network Congestion Control

The network congestion control is responsible for setting an upper limit on how much data can be on the network (bytes in flight) [33]. This upper limit is also known as a congestion window and is used later in the sender transmission control [33]. It uses techniques similar to LEDBAT [35] to measure the queuing delay [33]. The congestion window is calculated based on the queuing delay, received from the SCReAM receiver [33]. The congestion window is allowed to increase if the queuing delay is below the predefined queuing delay target, set between 50-100 ms [33]. This ensures that the queuing delay is always kept low. The reaction to loss or ECN events also leads to an instant reduction of the congestion window.

The network congestion control also responsible for calculating the send window which is used to determine whether RTP packets are allowed to be transmitted [33]. The send window is adjusted depending on the queuing delay, its relation to the queuing delay target, and the relation between the congestion window and the number of bytes in flight [33]. When the queuing delay is higher than the queuing delay target, a stricter rule is applied in order to avoid further queue buildup in the network [33]. For cases where the queuing delay is lower than the queuing delay target, a more relaxed rule is applied [33]. This allows the bitrate to increase quickly when no congestion is detected while still being able to exhibit stable behavior in congested situations [33].

3.2.3.3 Sender Transmission Control

The sender transmission control limits the output of data, given by the relation between the congestion window and the number of bytes in flight [33]. It uses packet pacing to mitigate coalescing, i.e., when packets are transmitted in bursts, with the risks of increased jitter and potentially increased packet loss [33]. Packet pacing limits the packet transmission rate given by the estimated link throughput [33]. Even if the send window allows for the transmission of a number of packet, they are not transmitted immediately [33]. Instead, they are transmitted in intervals given by the packet size and the estimated link throughput [33].

3.2.3.4 Media Rate Control

The media rate control serves to strike a reasonable balance between a low amount of queuing in the RTP queue(s) and a sufficient amount of data to send in order to keep the data path busy [33]. It is executed at regular intervals except during a prompt reaction to loss events [33]. The media rate control operates based on the size of the RTP packet send queue and observed loss events [33]. The queuing delay trend is also considered in order to reduce the amount of induced network jitter [33]. Setting the media rate control too cautiously leads to possible underutilization of network capacity and can cause the flow to become starved out by other more opportunistic traffic [33]. On the other hand, setting it too aggressively leads to increased jitter [33].

3.2.3.5 SCReAM Receiver

The SCReAM receiver is responsible for feeding back acknowledgements of received RTP packets and a total ECN count to the SCReAM sender [33]. In addition, the SCReAM receiver echoes back the reception time of the RTP packet with the highest sequence number [33]. Upon receiving an RTP packet, the receiver must maintain enough information to send the aforementioned values to the SCReAM sender using a RTCP transport-wide feedback message [33]. This is the case for each received RTP packet. The SCReAM implementation used for this thesis, makes use of the RTCP Congestion Control Feedback Report as the default RTCP feedback message. The frequency that determines how often a feedback message can be sent depends on the available RTCP bandwidth [33].

3.2.4 Comparison between NADA, SCReAM & GCC

Besides SCReAM, the initial designs of GCC and NADA perform most of their essential computations on the receiver-side and feeds back the information to the sender-side which performs the necessary operations such as limiting the bitrate. However, all parts of GCC and NADA may be implemented on the sender-side, while keeping the receiver dumb and solely responsible for sending back feedback messages, which SCReAM does by default [30, 32]. Both SCReAM and NADA use an approach similar to LEDBAT [35] to compute their queuing delay [33, 32]. They also possess rate adaptation modes which determine how fast the bitrate is increased [33, 32], while GCC determines the bitrate by the state of the FSM and the fraction of packets lost [6]. Besides limiting the bitrate, all three algorithms also limit the packet sending rate. GCC does this through the use of a pacer queue which sends a group of packets every burst time (5 ms) [30]. NADA computes a sending rate based on the updated parameters [32]. SCReAM uses a sender transmission control which computes a send window based on the relation between the number of bytes in flight and the congestion window [33].

3.3 Jitter & Latency

Latency is something familiar in networking and is simply a measurement of the time it takes for data to travel between two points [36]; however, usually measured as a round-trip delay (the time it takes for a message to travel back and forth); this can cause confusion as it can mean two related but different concepts. This implies latency, when viewed as a round-trip delay, is the time it takes for a message to be sent as well as acknowledged by both parties. To generalize these concepts is delay, which is, broadly speaking, the time it takes for any data to travel between two mediums. For instance, it can be viewed as the delay when trying to create a multimedia packet to when you are able to actually transmit it due to pacing constraints.

Packets travelling over the network can take different routes to reach various routing hardware that process' them. Inherently to this variant comes variation in the aforementioned latency, that is, based on certain conditions the time it takes to transmit packets is generally not fixed. This is known as jitter. [36] Jitter is the fluctuation of latency and can cause issues when it gets very high. Jitter can be the possible cause of the following issues when measuring QoS.

- **Packet loss** - jitter can cause inconsistent packet delivery and ultimately cause packet loss.
- **Network congestion** - A two-way issue. Buffering and congestion can cause delay in packets; however, observing possible jitter in the link can help monitor the congestion and, by acting on it, allow prevention of both issues.

3.4 Software

This section focuses on providing an overview of the software tools used during the implementation and experimentation of this thesis.

3.4.1 Containerization

Containerization is technique to isolate certain properties of an application from other programs running on the same operating system. It is roughly equivalent to a lightweight virtual machine [37] (VM). It offers great separation of resources while reducing the overhead that comes from virtualizing software. Other than the efficiency provided, a benefit of containerization is reproducibility, recreating an identical environment ideal for testing complex applications with a singular command.

3.4.1.1 Docker

Docker is a software stack used for many things pertaining to container deployment, but primarily allows the easy creation of containers from images which contains everything needed to run the application. Docker allows automated container creation, container versioning, shared libraries among containers and image (container base) versioning [38]. The three primary parts of Docker containers are [39]

- **Builder** - set of tools to build containers, e.g. `docker build`.
- **Engine** - runtime libraries to run containers, e.g. `dockerd`.
- **Orchestration** - management of many containers at once, useful for large scale deployment (i.e. not simply replicating a test environment)

3.4.2 Linux TC - Traffic Control

Traffic Control is a software utility available for Linux to regulate and manipulate traffic control settings of network traffic handled by the Linux kernel. [40] It deals with four methods of traffic policing that can be used to regulate traffic as well as three underlying objects to deal with them. To process egress (outgoing) traffic, traffic control uses shaping and scheduling.

- **Shaping** - Regulates the rate of transmission for egress traffic. Used primarily to regulate bandwidth but also to smooth out bursts in network traffic.
- **Scheduling** - Handles packet scheduling,

Traffic Control implements these methods using qdiscs, or queuing disciplines. [40] The Linux Kernel enqueues packets on to the qdisc, and attempts to send as many packets as possible to the underlying driver. Qdiscs exists either as classless or classful, with the former allowing no qdisc children while classful qdiscs allow you to pass along data to other qdiscs. Since classless qdiscs do not contain other qdiscs they can only be applied at the root level or as children of classful qdiscs, by the specifying the handle (the i.d.) of the parent qdisc. An example of a classless qdisc is netem, or `Network Emulation`. This qdisc handles rate limiting, latency emulation with jitter, packet loss as well as duplication on egress traffic on the targeted interface; netem allows multiple regulators at once.

An example command to apply netem with rate control and latency emulation can be the following, with # symbolizing elevated privileges required to execute the command.

```
# tc qdisc add dev eth0 root netem rate 1mbit latency 400ms 20ms
```

This would set a maximum rate of 1Mbit per second, with an added latency of 400ms with a jitter of maximum 20ms with a default uniform distribution, and applying this to network interface `eth0`. Additional options for a netem command includes packet loss with a stochastic behavior, packet corruption, packet duplication, packet reordering as well as configuring multiple slots with different properties in them such as applying a delay to packets in that specific slot.

3.4.3 WebRTC

WebRTC (Web Real-Time Communication) is a web technology with a goal in aiding to capture as well as to stream different types of media between parties [41]. The aim is to provide a set of standards to simplify relaying media between any party while ensuring minimum incompatibility (e.g. without the need for plugins). This is done by providing a clear set of standards and APIs (Application Programming Interfaces) for the developer giving the assurance that it is very likely that both sides will support the data being sent. It can be used for simple voice calling, although it supports large-scale group calling as well as real-time video streaming, including sending arbitrary data e.g. chat messages or statistics over data channels. While sending data between computers at an application level is simple, having packets clear firewalls and router configurations is not a simple task. Therefore, WebRTC incorporates the following streaming concepts and technologies to aid in connecting computers behind different configurations [42].

- **ICE | Interactive Connectivity Establishment** - ICE takes care of abstracting away the connection protocols used to bind clients together regardless of the underlying network topology.
- **STUN | Session Traversal Utilities for NAT** - STUN is a protocol used to send data to a computer behind a NAT (Network Address Translator).

NAT can be problematic to properly send data through and the reason why STUN is useful in aiding negotiations between clients. Unlike TURN, STUN only aids in creating a connection between clients and does not relay any actual media data.

- **TURN | Traversal Using Relays around NAT** - TURN is a protocol similar to STUN aiding in sending data between clients. However, TURN servers act as a relay forwarding all data sent to it to the relevant client.

Further technologies used by WebRTC include the low-level TCP and UDP, encryption via TLS/DTLS, RTP/SRTP, which can be seen in Figure 3.12.

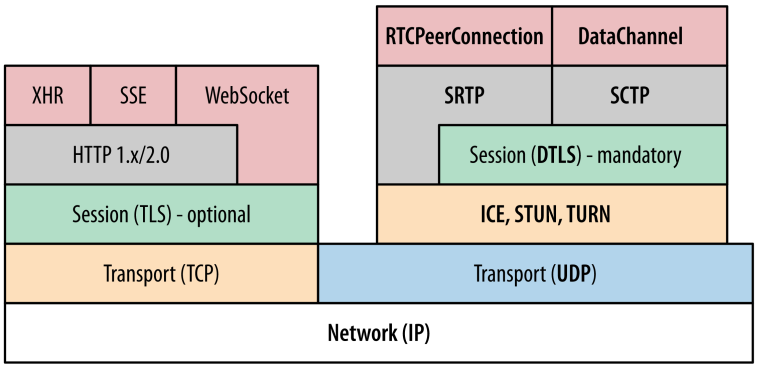


Figure 3.12: Overview of WebRTC technologies [42] (CC BY-NC-ND 4.0).

3.4.4 GStreamer

GStreamer is a multimedia framework intended for developing streaming media applications. It is composed of different elements creating a media pipeline that can handle various kinds of video formats as well as video consumers [43]. Beyond the core functionality of GStreamer, it supports a powerful plugin system to extend the functionality. However, most features are already available in the common repositories.

A GStreamer element is an object taking an arbitrary number of inputs and producing an arbitrary number of outputs, performing calculation on the input data before forwarding it down the pipeline. For instance, a file on the computer produces data while a video player consumes video data to show it on screen. These elements build together a pipeline that can process many types of video data in a platform-independent manner, by supporting programmatic pipelines as well as textual pipelines parsed by GStreamer. For development there are multiple ways to communicate between and with GStreamer elements. They are represented internally via Buffers, Events, Messages and Queries, shown in 3.13. Furthermore, elements can expose GLib signals which notify event listeners that an event has occurred, as well as typed properties that can be read. The types used by GStreamer are the following.

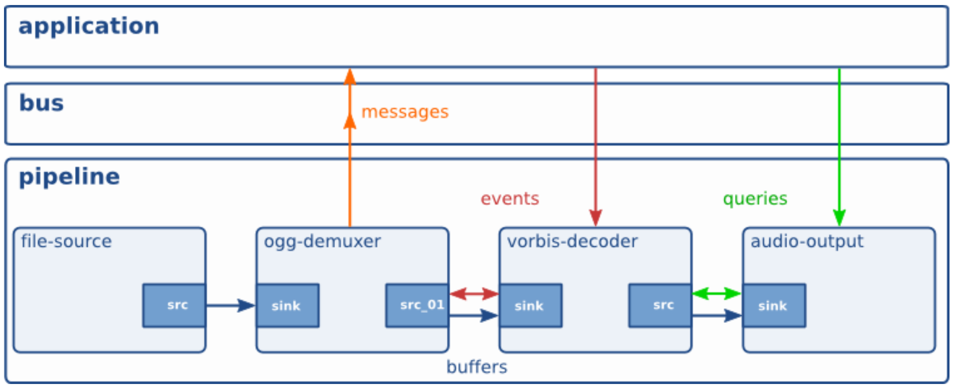


Figure 3.13: A small example pipeline, showing direction of communication [44] (CC-SA 4.0).

- **Buffers** - Primary means of communication. Contains encoded data from source to sink pads.
- **Events** - Events can be sent in both directions, and from application to elements or vice versa.
- **Messages** - Messages are posted from the elements to a message bus, which can indicate a state change in an element, an error or something that needs processing.
- **Queries** - Queries are used to query data from elements, which can be information related to the current media e.g. playtime.

For the developer the internals of the element is not of interest, rather, a developer acts on the pads, properties and signals exposed by the element. A pad is the most important part of an element, where the relevant data is sent and pushed. The type of data varies from raw byte data to compressed video frame, among other types; the pad type can be exposed via its capabilities, which expose a set of properties including what type of data it sends and/or receives. Pads come in two forms, sink and source, where the former receives buffers and the latter produces them. An element can have either one or both, and create more on demand via request pads. These can be useful when the number of inputs and/our outputs are not known.

Chapter 4

Approach

This chapter describes how the framework required to evaluate the algorithms in Axis' system is built. It also details the individual components needed to set up the entire software stack, from how the algorithms are implemented in Axis' system as well as the parts needed to collect results in Docker and directly on the camera.

4.1 Method and Implementation

This thesis evaluates the algorithms that fulfil the criteria mentioned in the requirements for congestion control [3], specifically NADA, SCReAM and GCC. To ensure strong compatibility with varying receivers of media, TWCC is implemented and added as part of the evaluation.

To implement the algorithms, plugins for GStreamer have been implemented which can sit between the video producer and the WebRTC media transmitter, being an intermediary of RTP and RTCP data. While the algorithms themselves exist as open-source code, the code needed to create the plugins and to evaluate them within Axis' WebRTC framework has been written. For SCReAM and NADA in particular, there exists open source code which were implemented by the original authors [45, 34]. Furthermore, the SCReAM implementation is wrapped in as a GStreamer plugin, allowing direct inclusion into the WebRTC GStreamer pipeline. While each algorithm required separate implementations, the boilerplate code to create a GStreamer plugin is derived from Ericsson's GStreamer plugin for SCReAM, reducing the amount of code needing to be written.

Other than adding support for the algorithms themselves, the glue code needed to bind together the WebRTC system with the algorithms has been written. This includes individual GStreamer pipelines, that although similar have to ensure the correct algorithm is chosen based on environment variables. Furthermore, statistics collections have been written with support for the kind of multithreaded program that GStreamer is. Lastly, the simulation suite has been written which emulates a real WebRTC system, to allow for results similar to that of a production environment.

4.2 Algorithm manager in WebRTC

In order to simplify the testbed for different algorithms as well as configuring different parameters an algorithm manager was implemented into Axis' WebRTC system. It is responsible for configuring the correct pipelines, setting up algorithms as well as setting up the correct pads for RTP/RTCP messaging between GStreamer elements. Furthermore, it is the endpoint for statistics collection which it reads from the relevant properties in the algorithm elements, at a configurable 50ms interval. As compilation times are quite large, especially compiling directly for the camera, environment variables are used to configure the algorithms, therefore eliminating the need to recompile between tests.

Name	Description
MIN	The minimum algorithm output (kbps)
MAX	The maximum algorithm output (kbps)
ALGO	Which algorithm to configure
LOCAL	If true, running in a simulated environment.

Table 4.1: List of environment variables.

Configuring these parameters before each test allowed simplified setup of a larger, scripted test environment.

4.3 Pipelines in GStreamer

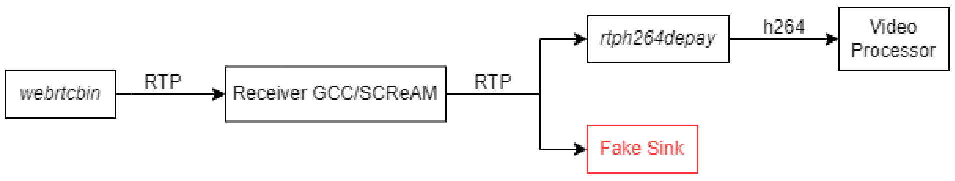
Figures 4.1 and 4.2 shows the GStreamer pipelines used for testing. Since NADA is used in conjunction with TWCC, there is no external element added to the receiver pipeline, and it remains unchanged. However, both sides instantiate TWCC as part of the RTP payloader/depayloader, with example code available in the appendix.

This ensures that transport feedback is emitted by the video payloader and that RTCP feedback gets sent by the client and analyzed by the target. Furthermore, the server listens to the signal `on-feedback-rtcp` emitted by the RTCP pipeline. This signal contains the FCI (Feedback Control Information) containing all the necessary timestamps. However, this signal is emitted multiple times per packet meaning identical input has to be filtered out.

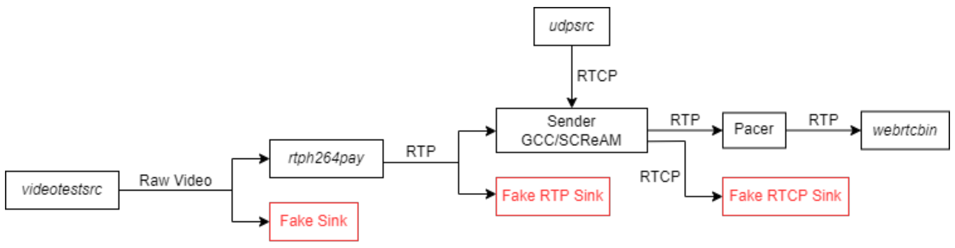
4.3.1 Rust FFI

As a consequence of implementing the plugins in the Rust programming language, it was necessary to bridge Rust code with algorithms implemented in C. The FFI (Foreign Function Interface) for NADA and GCC is shown in the appendix.

This allows writing the necessary bridge between camera/fake video source and the algorithm in Rust while not having to rewrite the algorithm implementation in Rust, keeping the original C code but allowing a fast and safer implementation.



(a) Receiver pipeline.



(b) Sender pipeline.

Figure 4.1: Scream and GCC Pipelines.

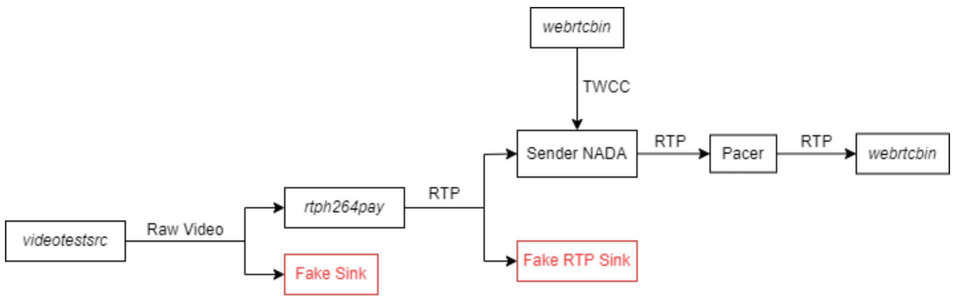


Figure 4.2: Nada Sender Pipeline.

4.4 NADA

To implement NADA as a GStreamer plugin, the following parts are required to implement a complete algorithm.

- **Feedback input** - Ensuring that the feedback from TWCC is properly delivered to NADA, to allow it to react to changes in the network.
- **Statistics** - Off-thread and on-thread data collection for analysis. This was done for all algorithms.
- **Timestamping** - Ensuring all packets are timestamped with send time, arrival time (client clock) as well as receive time on sender, for NADA.
- **TWCC Parsing** - As raw feedback is inputted, implement a custom parser for TWCC data.

- **Rate-shaping buffer** - A buffer that acts as a pacer queue, to offset difference between video output and sending rate.

Handling feedback input was handled via a GStreamer signal, relaying RTCP data to the NADA plugin. This data was parsed via the custom TWCC parser and offloaded to another thread for processing and for forwarding to the algorithm implementation.

NADA also specifies the existence of a rate-shaping buffer [32] but does not declare its implementation; instead, declares what its purpose is. To implement this, an off-thread queue was added that consumes RTP packets at a rate defined by NADA. The remaining time, if any, the thread sleeps waiting for the next opportunity to send packets. Figure 4.3 shows a high-level overview of the rate-shaping buffer implementation.

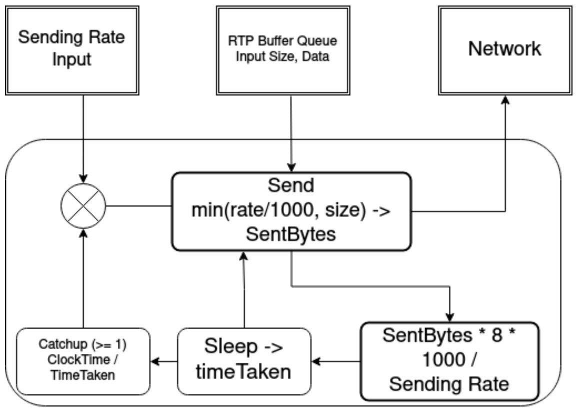

Sending Rate Input
RTP Buffer Queue Input Size, Data
Network
Send min(rate/1000, size) -> SentBytes
SentBytes * 8 * 1000 / Sending Rate
Sleep -> timeTaken
Catchup (>= 1) ClockTime / TimeTaken

Figure 4.3: Rate shaping buffer flow.

When there is available data, the sending rate is computed from the sending rate input from the algorithm multiplied by possible catch up (should the time it takes to execute the code segment exceed sleep interval the sending rate is temporarily increased). From the RTP Buffer Queue packets are dequeued until the amount of bytes sent would exceed 10ms of sleep, or if the buffer is empty. By queuing more packets at once rather than one at a time (to exceed 10ms of sleep), it offsets the time it takes to actually run the function, as sending individual packets would stall the thread. These packets are then timestamped and sent over the network. Afterwards, the time to sleep is computed as $\frac{SentBytes*8*1000}{SendingRate}$, e.g. if 1000 bytes is sent over the network with a sending rate of 100 kbps the thread would sleep 8ms, minus the time it took to run the loop.

4.5 GCC

To implement GCC, a freestanding GCC implementation [46] is used, which is also used in a previous paper as a GCC base for the RMCAT testing [7]. To implement GCC as a GStreamer plugin required that all parts of the algorithm are called at the right time, as well as having the callbacks set up correctly.

The two necessary implementations to enable GCC involved first tagging RTP packets with a header extension of 3 bytes, which includes the sender-side timestamp of when the packet was sent, to be made available for the receiver side. While this has an overhead it enables the client to be aware of the time the packet is sent on the sender. Second, unlike SCReAM and NADA, GCC does not expose round-trip time metrics. To implement round-trip time metrics for GCC a separate callback is used whenever a packet feedback is received. The timestamp is stored in a map used as a sliding window 4.4, calculating the round-trip time over all buffers with a timestamp less than 1000ms relative the current measured time. This is an approximation of the round-trip time, as round-trip time is affected by the interval at which the client sends feedback to the sender.

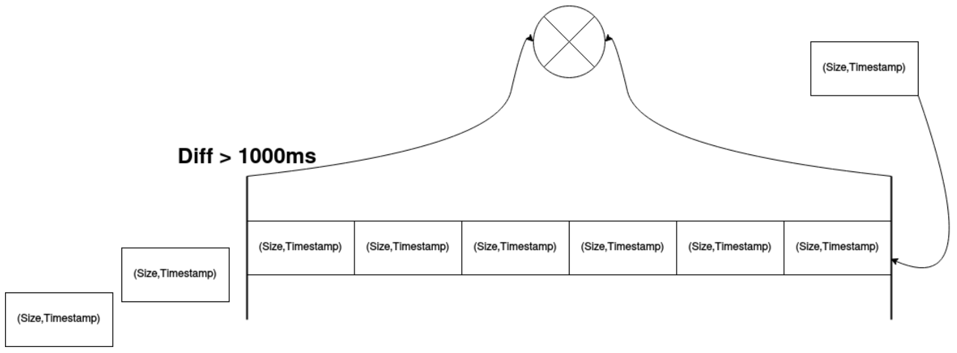


Figure 4.4: A sliding window, summing over new (1000ms) values. Old values are eventually dropped.

4.6 SCReAM

SCReAM is a reference implementation made by Ericsson available as a GStreamer plugin [45]. It contained all the necessary code to be added to an existing GStreamer pipeline, including all FFI code as well as logging information. To add SCReAM to the Axis pipeline for testing the pads of the plugin has to be wired up to the proper sources and sinks as well as extracting relevant performance metrics from the algorithm internals. No other methods are written for SCReAM. Similarly to GCC, to simplify testing a separate channel was used to send RTCP traffic rather than attempt to reconfigure the transport layer of WebRTCBin in GStreamer.

4.7 Statistics

To collect statistics data was outputted to a file available outside the docker container. A timer was set up to run every 50ms to collect current data from the system. This is printed as a comma-separated list, one line per 50ms. The available data is actual measured data sent over wire, algorithm output, video output, the parameters from the simulation (to be used for plotting) which includes currently simulated bandwidth, latency and jitter. Lastly, there is optionally TWCC stats available from the underlying GStreamer internal TWCC statistics. Statistics dealing with bandwidth and round-trip time are calculated using a sliding window 4.4 as statistics such as bytes sent over network are not instantaneous measurements.

Round-trip time is measured via feedback received from the client, taken as an average of the smallest current round-trip times in the buffer as it is the time for a client to acknowledge a packet and return feedback on it, calculated as the current timestamp when receiving the feedback minus the send timestamp from the pacer queue. (stored as a buffer of minimum round trip time values) However, this is then dependent on how often the client sends feedback and therefore is not an exact measure of network latency (unlike a ping-pong measurement) but a measure of feedback response time.

4.8 Hardware implementation

To perform an initial evaluation on the camera the algorithm is compiled for the `armv7hf` architecture and added to the WebRTC package running on the camera, using cross-compilation for the ARM target. As it is the same framework running on both targets, Axis' WebRTC system, it allows communication between the camera and the computer running the same program built for the desktop. To simplify the communication part only NADA is tested as it integrates with the built-in TWCC implementation and thus is not reliant on a separate RTCP channel; this allowed a simple integration in the camera as it avoided extra network negotiation. The camera is connected to the testing computer via a switch and to test out the performance, the `tc` command is running on the camera to regulate network in the same manner as earlier simulations.

Chapter 5

Evaluation

This chapter gives an insight into how the experimental process was performed, the results obtained from said experiments, and the implications they provide.

5.1 Experiment Setup

This section describes how the experiments are set up in order to evaluate the algorithms and the camera respectively.

5.1.1 Simulated Docker Environment

The experiments are set up using two Linux kernel Docker containers as the sender and receiver, using the testbed topology similar to the *Testbed Topology for Limited Link Capacity* [47]. The sender container is responsible for encoding the video and sending them as RTP packets. The receiver container would receive RTP packets and return RTCP feedback messages to the sender. Upon receiving RTCP messages, the sender container would execute the currently used algorithm to compute the new bitrate that would be set to the encoder. Depending on the algorithm used, the sender container may execute additional tasks such as computing additional parameters in order to limit how much data can be sent through the network. While data travels through the network, interference is introduced by using the `tc` command, allowing us to limit the bandwidth and introduce artificial delay and jitter.

The experiments are carried out on a computer provided by Axis, with the specifications provided in Table 5.1. They use a synthetic white noise video generated by a GStreamer element, also known as the snow pattern. The snow pattern was used above others because of its hard-to-encode properties due to the generated video being very noisy, ensuring that the video does not fall too far below the target bitrate, adversely affecting the results. However, this is a purely simulated video and allows great control over the output bitrate.

The following experiments were performed:

- **RMCAT-NS3 Test** - An experiment similar to the experiment performed in the study [7]. Here, the bandwidth is adjusted using a staircase pattern between 2000 kbps and 500 kbps. Every 20 seconds, the bandwidth would be reduced by 500 kbps until it reaches 500 kbps. It would then start increasing

Motherboard	ASUS PRIME X570-P
Processor	AMD Ryzen 9 3900X
Memory	32 GB DDR4/2666 MHz
Storage	SSSTC CL1-8D256-HP 256 GB Seagate Barracuda Q5 1000 GB
Graphics Card	Nvidia GP107GL [Quadro P400]
OS/Kernel	Debian 11 (Bullseye)

Table 5.1: Technical Specifications of the computer used in the experiments.

the bandwidth until it returns to 2000 kbps and then repeat the pattern. This experiment is performed in 200 seconds, similar to the study [7].

- **Sudden Reduction** - Intends to simulate the network behavior in 4G. The bandwidth limit is kept at 2000 kbps, but during three occasions, the bandwidth would drop to 500 kbps for a few seconds and return afterwards to 2000 kbps. The duration of these drops lasts 3, 5, and 2 seconds respectively. The entire experiment lasts 2 minutes.
- **Sweden to China** - Similar experiment to *Sudden Reduction*, but includes a delay of 320ms, which represents the ping between Stockholm and Hong Kong [48]. It also includes a jitter of 80ms in order to introduce variation. The intent with this experiment is to simulate how the algorithms behave if video generated in a low-latency environment would be sent through a stream to a location with a considerable distance between the sender and receiver.

Note that while the experiments run within a set duration, the algorithms continue to run while the Docker container attempts to shut down, making the algorithms run for an additional second or two. The experiments are executed 5 times in succession in order to validate whether the same patterns reoccur, and to identify any abnormalities that might occur.

5.1.2 Camera

The experiments done on the surveillance camera are set up similarly to Section 5.1.1. Instead of a sender container which sends RTP packets to the receiver container, the camera is responsible for sending RTP data to the receiver. Thus, the camera sends a live feed to the receiver container, which sends TWCC feedback messages in order for NADA to adjust the bitrate. The camera used during the experiments is an IoT surveillance camera with an ARTPEC-6 microchip. Furthermore, since the camera has a built-in video encoder, the NADA algorithm changes the bitrate of the encoder using a custom-made camera app. Here, interference is also introduced using the `tc` command. In order to properly identify that NADA is used to control the congestion of the camera, the *RMCAT-NS3 Test* is performed in order to validate that bitrate changes when bandwidth limitation is introduced. Considering that *RMCAT-NS3 Test* changes the bandwidth limit several times during test, there is no need to perform it several times in order to validate pattern recurrence as the experiment is mainly to show that the camera can respond to change in order to provide a *Proof of Concept*. The experiment is performed when the camera generates a live feed with minimal movement in 480p, 720p, and 1080p resolution respectively.

5.2 Statistics Computation

The statistics computed in order to evaluate the performance of the congestion control algorithms include graphs containing the average curve of each algorithm, and box plots showing the range of values throughout the test iterations of each algorithm. To obtain the average curve, the linear interpolation of the data points for each test iteration is computed, as the number of data points for each test iteration is not the same. Using these linear interpolations, we compute their mean which becomes the average curve.

The box plots may be computed differently depending on what parameter is looked upon. For link utilization, this is done by sampling the bitrate output and divide it with the sampled bitrate limit for each data point. Afterwards, the mean of each fraction is computed, which represents the mean percentage link utilization. This is done for each test iteration separately. Finally, the mean percentages of each iteration are stored in a dataset and used to create a box plot. A box plot consists of five elements: the minimum, the maximum, the sample median, and the first and third quartiles. The sample median is the middle value in the dataset, represented as the line inside the boxes. The first and third quartiles are the medians of the lower and upper half of the dataset respectively, represented as the horizontal sides of the boxes. The minimum and maximum represents the lowest and highest data points respectively, excluding any outliers. They are represented as horizontal lines outside the boxes. The distance between the minimum and first quartile, or the third quartile and maximum are called whiskers. Data points observed outside the boundaries of the whiskers are called outliers, represented as dots.

The computation of the latency box plots are similar to that of the link utilization box plots, where the average RTT is computed for each test iteration and The means are thereafter used to create the box plots. This computation uses data collected from the average minimum RTT each statistic cycle, which shows the fastest time to receive and acknowledge a packet. Packet loss box plots are computed identically to the latency box plots. However, because NADA sampled the loss in decimals, a multiplication of 100 was required in order to convert it to percentage.

Responsiveness is determined by computing the time taken for the algorithm output to change drastically. This is done by sampling the current algorithm output whenever the bandwidth limit is subjected to change. The output is compared to upcoming outputs by computing the difference between the current output and the upcoming outputs. If the difference is below 250 kbps, the response time is incremented by 50 ms, since packets are sent every 50 ms. Upon finding a difference that is greater or equal to 250 kbps, the response time is stored. The stored response times are then used to compute the means, which are then box plotted similar to link utilization and latency. The 250 kbps limit was chosen in case the algorithm would fluctuate throughout the tests, in order to combat improper readings.

Since the box plots are computed using the means from each test iteration, there are only five data points used to plot the box plots, as it uses the already computed mean of the individual tests. This can cause the median to be skewed to one of the sides of the boxes.

5.3 Results

In this section, we introduce the results extracted from the performed experiments described previously.

5.3.1 RMCAT-NS3 Test

Figure 5.1 shows the average algorithm output as well as the average round trip time, and packet loss for each algorithm in *RMCAT-NS3 Test*. Figure 5.2a shows the spread of the link utilization for *RMCAT-NS3 Test*, while Figure 5.2b displays the latency. Figures 5.2c and 5.2d presents the spread of responsiveness and packet loss respectively.

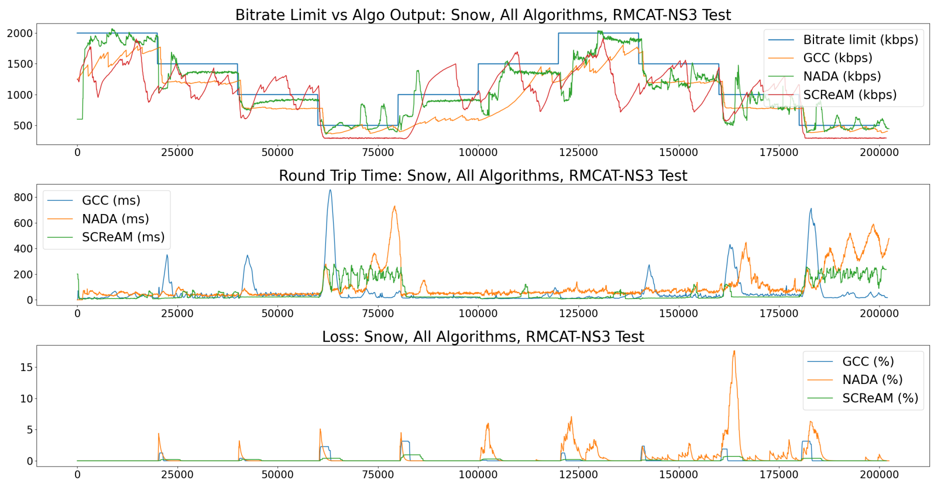
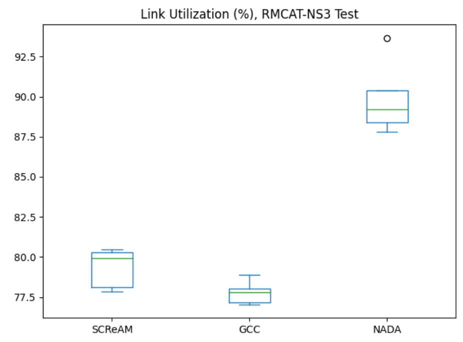
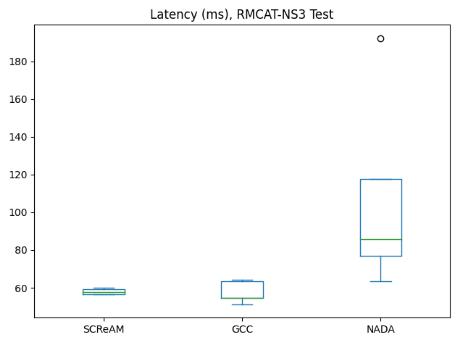


Figure 5.1: Average of each algorithm in the RMCAT-NS3 Test.

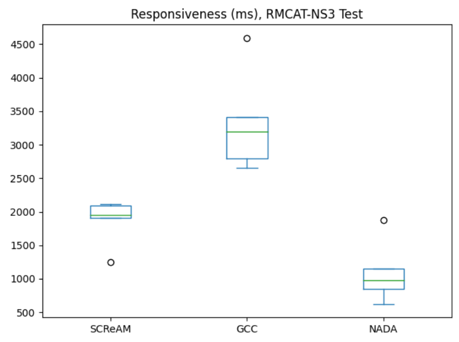
As seen in Figures 5.1 and 5.2a, NADA has the largest link utilization compared to the other algorithms. Figure 5.1 also shows that NADA attempts to obtain as much bandwidth as possible when bandwidth limit is changed, but then starts to stabilize. This directly correlates with responsiveness as seen in Figure 5.2c, as NADA's aggressiveness causes its responsiveness to be lower than the other algorithms. Figure 5.1 can also be seen that NADA has some cases of over



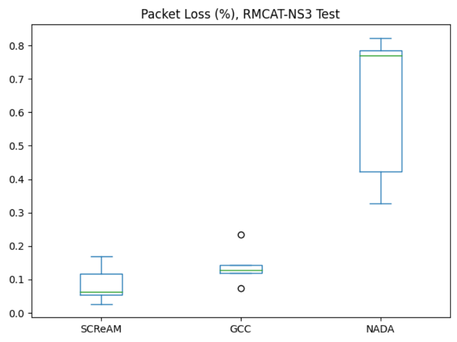
(a) Link utilization box plots of all algorithms during the RMCAT-NS3 Test.



(b) Latency box plots of all algorithms during the RMCAT-NS3 Test.



(c) Response box plots of all algorithms during the RMCAT-NS3 Test.



(d) Packet loss box plots of all algorithms during the RMCAT-NS3 Test.

Figure 5.2: Generated box plots from the RMCAT-NS3 Test.

utilization, but manages to stabilize itself very quickly. In terms of latency, NADA has a relatively large spread as seen in Figure 5.2b. However, looking at Figure 5.1, it is shown that NADA manages to maintain a stable round trip time with exceptions in middle and at then end of the experiment. While NADA does have the largest spread of packet loss, it is only less than 1%, as shown in Figure 5.2d. The spread also shows that the packet loss varies depending on which test iteration is observed. Figure 5.1 also shows that NADA has occurrence of peaks as high as 15%, however due to how infrequent the peaks occur, the average packet loss does not get as affected by the peaks, hence the $< 1\%$ spread in Figure 5.2d. This also shows that the main variation in spread is caused by the variation in peaks between the iterations.

Looking at the GCC algorithm, the average curve in Figure 5.1 manages to maintain the link utilization very well in the earlier parts of the tests. However, as the bandwidth limit increases, GCC starts to utilize the available very slowly compared to the other algorithms. This causes the low link utilization and high responsiveness as seen in Figures 5.2a and 5.2c respectively. In terms of latency, GCC maintains a low spread between each iteration as shown in Figure 5.2b.

Figure 5.1 also shows the occurrences of peaks which occur when the bandwidth limit changes. The peaks also seem to vary depending on the new limit value. In terms of packet loss, peaks seem to be present, however, they do not peak as high as NADA. GCC also seems to maintain a low spread across the test iterations as seen in Figure 5.2d.

SCReAM seems to fluctuate a great deal during the test, as seen in Figure 5.1. There are also cases where the algorithm tends to overutilize the available bandwidth. Despite the fluctuations, SCReAM manages to maintain a relatively good spread between each test iteration, as shown in Figure 5.2a. We can see a similar occurrence in responsiveness as shown in Figure 5.2c. SCReAM also manages to maintain very low latency during the test, only increasing whenever the bandwidth limit is set to 500 kbps, as seen in Figure 5.1. In Figure 5.2b, the latency also seems to be almost identical between the test iterations, showing a very small spread. Packet loss is also shown to be occurring during the test with SCReAM. However, the average curve shows very little signs of peaks as seen in Figure 5.1, and the spread between the iterations remains low as shown in Figure 5.2d.

5.3.2 Sudden Reduction

Figure 5.3 shows the average algorithm output, the average round trip time, and packet loss for all algorithms during *Sudden Reduction*. Figure 5.4a shows the spread of the link utilization during *Sudden Reduction*. Figure 5.4b shows the round trip time spread, Figure 5.4c shows the response spread across all test iterations, and Figure 5.4d focuses on the packet loss spread.

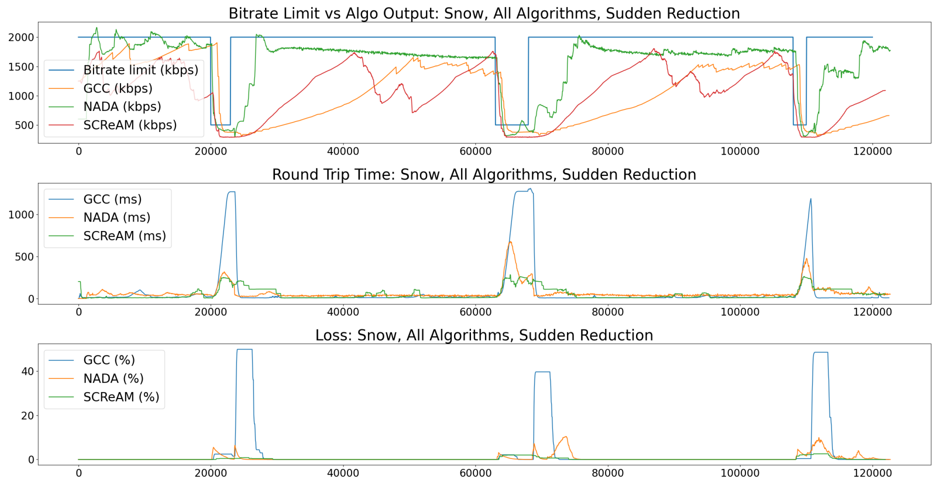


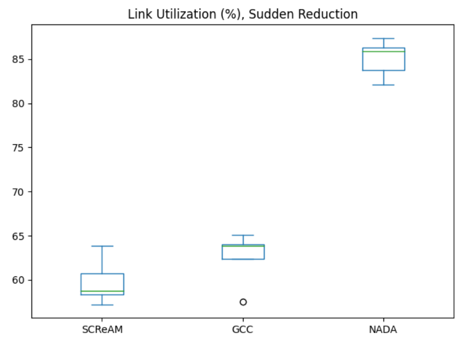
Figure 5.3: Average of each algorithm in the Sudden Reduction test.

Similar to the *RMCAT-NS3 Test*, NADA is the best performing algorithm in terms of link utilization, as seen in Figure 5.3 and 5.4a. This was also directly correlated to low responsiveness seen in Figure 5.4c. In terms of latency, Figure

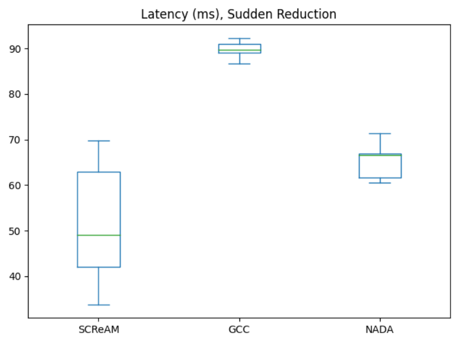
5.3 shows that NADA manages to maintain a stable latency throughout the test, with peaks occur during changes in bandwidth limit. NADA also manages to maintain a good spread between the test iterations in terms of latency, as seen in Figure 5.4b. There is also occurrences of packet loss peaks as seen in Figure 5.3, which occur during changes in bandwidth limit. However, loss only seems to occur during those events, which causes the loss to be $< 1\%$ with a low spread between the iterations.

GCC also seems to show similar behavior as in *RMCAT-NS3 Test*. Looking at Figure 5.3, GCC shows once again the ability to quickly respond to decreases in the bandwidth limit, while slowly attempting to utilize the available bandwidth whenever the bandwidth limit increases. Due to this, the responsiveness increases as much as 6000 ms as seen in Figure 5.4c. GCC also has the largest round trip time and packet loss, as observed in Figures 5.4b and 5.4d respectively. This is caused by the peaks in Figure 5.3 which reach as high as 1000 ms and 50% for round trip time and packet loss respectively.

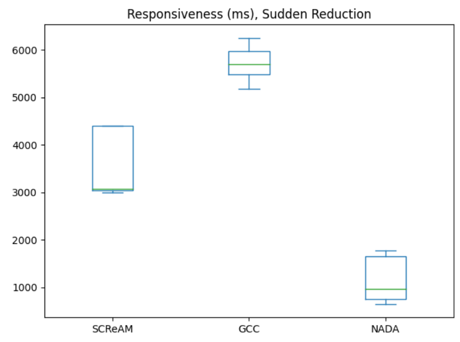
Similar to the other algorithms, SCReAM shows similar behaviors in *Sudden Reduction* with a fluctuating link utilization, but low packet loss and latency. Due



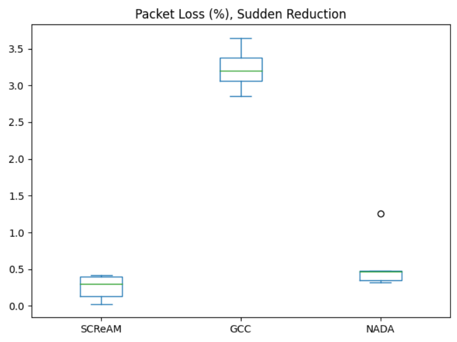
(a) Link utilization box plots of all algorithms during the Sudden Reduction test.



(b) Latency box plots of all algorithms during the Sudden Reduction test.



(c) Response box plots of all algorithms during the Sudden Reduction test.



(d) Packet loss box plots of all algorithms during the Sudden Reduction test.

Figure 5.4: Generated box plots from Sudden Reduction.

to the fluctuating link utilization, SCReAM has the lowest average link utilization as seen in Figure 5.4a. However, due to being able to respond quicker to increases in bandwidth limit as seen in Figure 5.3, its responsiveness is lower compared to GCC as shown in Figure 5.4c.

5.3.3 Sweden to China

Similar to Figures 5.1 and 5.3, Figure 5.5 shows the same parameters evaluated during *Sweden to China*. Figures 5.6a, 5.6b, 5.6c, and 5.6d shows the spread of link utilization, latency, responsiveness, and packet loss across all test iterations for *Sweden to China* respectively.

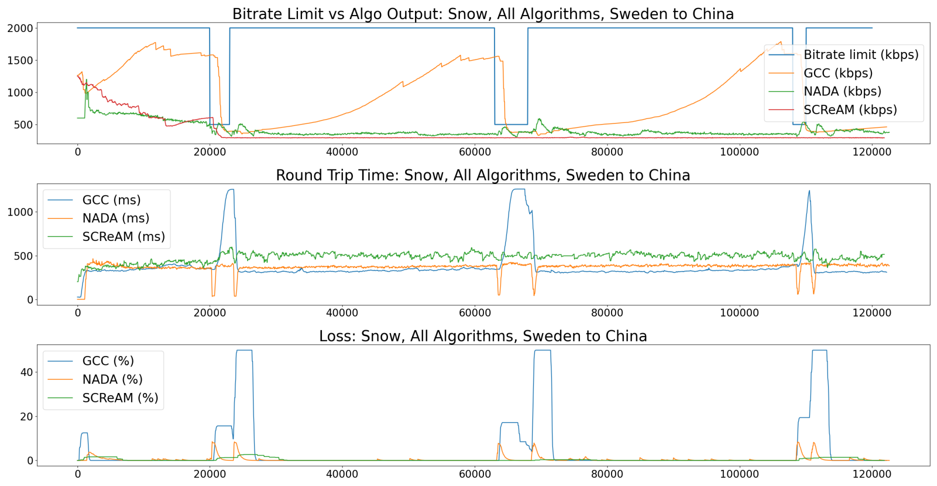
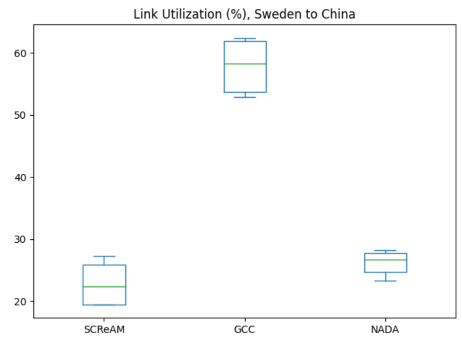


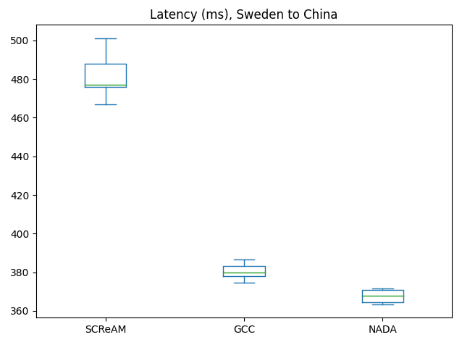
Figure 5.5: Average of each algorithm in the Sweden to China test.

One noticeable difference to the previous experiments is that NADA and SCReAM are unable to perform when exposed to high delay and jitter, as shown in Figure 5.5. This causes SCReAM and NADA to have poor link utilization during *Sweden to China*, as seen in Figure 5.6a. Furthermore, it is noticed that latency also correlates to link utilization when it comes to SCReAM. In Figure 5.5, when the average curve for algorithm output gets reduced, the latency curve rises. Due to the behavior seen in Figures 5.5 and 5.6c, responsiveness is not able to be properly measured for SCReAM and NADA, as they provide improper results. The assumption is that a combination of how response time is measured and the occurrence of abnormalities, creates an inaccurate sampling, giving a false impression that SCReAM is quicker than what it actually is, and NADA having a wider spread than it should. On the other hand, both SCReAM and NADA appear to be stable in terms of latency and loss, having relatively small spreads between each iteration, as seen in Figures 5.6b and 5.6d. It is worth noting that NADA also seems to reduce its latency during changes to the bandwidth limit.

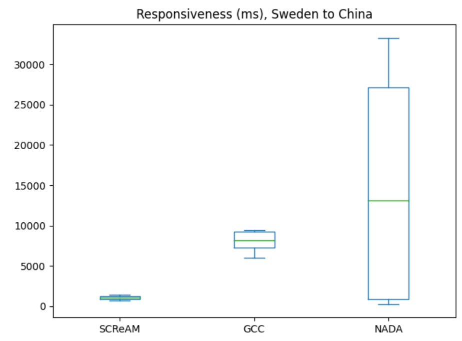
Looking at Figure 5.5, GCC displays similar behaviors as seen in *Sudden Reduction*, albeit slower increases in algorithm output, and larger round trip time



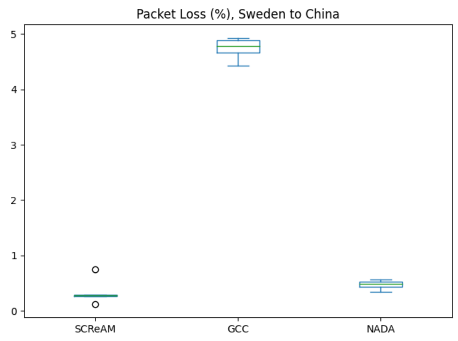
(a) Link utilization box plots of all algorithms during the Sweden to China test.



(b) Latency box plots of all algorithms during the Sweden to China test.



(c) Response box plots of all algorithms during the Sweden to China test.



(d) Packet loss box plots of all algorithms during the Sweden to China test.

Figure 5.6: Generated box plots from Sweden to China.

and packet loss. It has the largest link utilization compared to the other algorithms, with a reasonable latency, responsiveness, and loss as seen in Figures 5.6a, 5.6b, 5.6c, and 5.6d respectively.

5.3.4 Camera

As previously mentioned, the camera was experimented upon using 480p, 720p, and 1080p respectively. The bitrate relative to the bandwidth limit, the round trip time as well as the CPU and memory usage was collected and represented in a graph. See Figure 5.7 for the results. However, as it is running on a real camera, the results were not completely synchronized, hence the individual network limits are shown as well.

The CPU usage is measured as a total usage rather than per-thread usage. As the CPU has 4 threads a 75% utilization implies 3 threads are maximally utilized.

The limit can be seen to follow the reference bandwidth curve fairly well, similar to that of previous tests. A notable difference is that for low resolution it does not properly detect a drop towards the end, as the amount of data produced

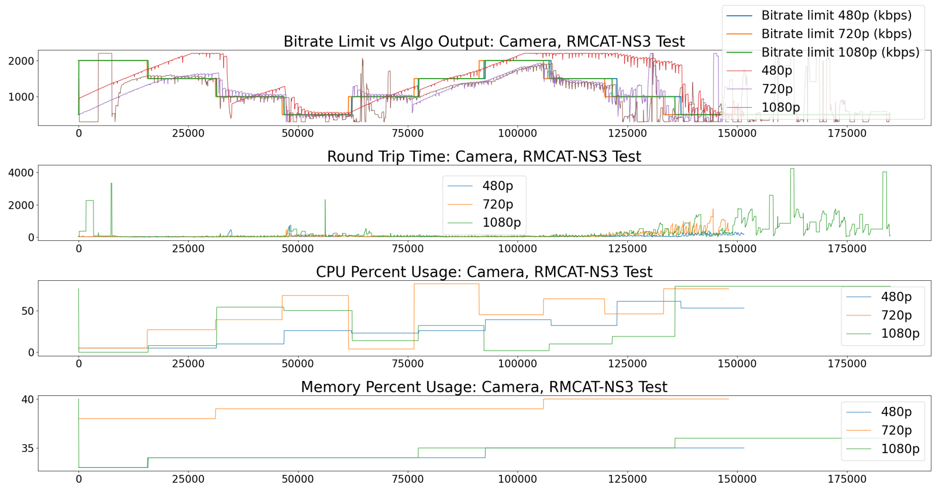


Figure 5.7: Results from RMCAT-NS3 Test generated by the ARTPEC-6 IoT surveillance camera.

is fairly low compared to the higher resolutions. Also, the CPU usage can for all resolutions be seen to increase substantially towards the end, pointing to a possible issue present during longer runs, also visible as the round-trip time increases quite a bit towards the end.

5.3.5 Additional Tests

Due to the behaviors in bitrate that SCReAM and NADA display during *Sweden to China*, additional tests were done in order to validate the reason behind said behaviors. Thus, an experiment was done similar to *Sweden to China* where only delay is introduced but not jitter, as introducing both delay and jitter makes it difficult to determine the cause of said behavior.

Looking at Figure 5.8, the round trip time and packet loss remain similar to the behavior observed in *Sweden to China*. However, a noticeable difference it that NADA is able to behave similarly to *Sudden Reduction* when only high delay is present.

5.4 Discussion

In this section, the results obtained from the experiments specified above will be discussed, as well as the difficulties encountered while obtaining the results.

5.4.1 Real data

During this experiment, real video (from a fake element) as well as live video from a camera was used to test the algorithms. Unlike [49, 7], which adhere to a stricter simulated environment to ensure replicable results and instead use

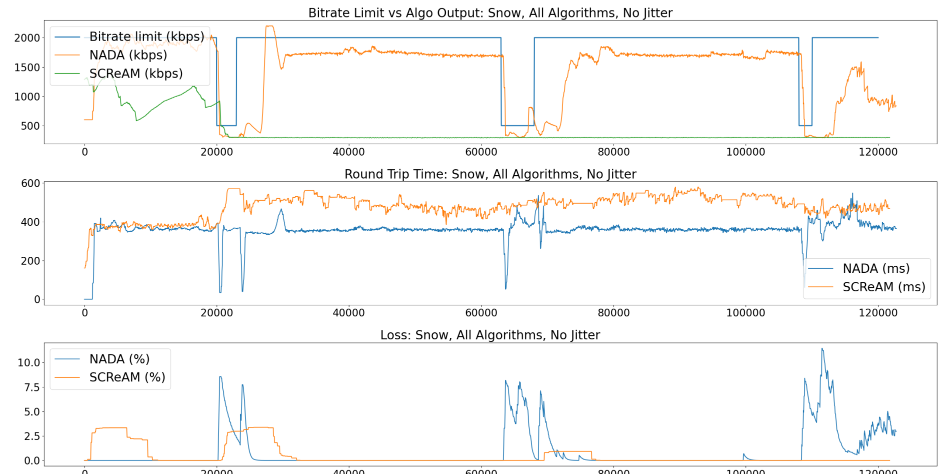


Figure 5.8: Average from NADA and SCReAM during the No Jitter test.

synthetic video codecs, these tests were done on a real camera system albeit with network conditions that are simulated. As a real codec uses target bitrate in different ways, for example as a ceiling to the maximum bitrate, and can not guarantee a consistent rate of data there can be a difference in the amount of data being produced to the amount of data expected by the algorithm. However, the important part is that the rate of data does not exceed the expected amount of data over amortized time.

5.4.2 Link Utilization

As previously stated, NADA has the highest link utilization in both *RMCAT-NS3 Test* and *Sudden Reduction*. It can also be seen that NADA attempts to obtain as much bandwidth as possible when bandwidth limit is changed, but then starts to stabilize. This is caused by NADA being set to accelerated ramp-up mode which increases the bitrate based on a multiplier determined by the observed round trip time and target feedback interval [32]. This can also cause NADA to overutilize the available bandwidth during certain occasions as seen in Figures 5.1 and 5.3. In *Sweden to China*, NADA shows an inability to properly utilize the available bandwidth when exposed to high delay and jitter, as seen in Figure 5.5. However, looking at Figure 5.8, we can see that this is actually caused by high jitter. The reasoning behind this statement has to do with how NADA computes its queuing delay. While the standard [33] states that it computes the queuing delay similar to LEDBAT [35], it only uses delay samples within an observation window. This means NADA only computes queuing delays from packets that arrive within the same observation window. If delay remains consistent, the queuing delay will not ramp up and cause the algorithm to switch to gradual rate update mode. However, the high jitter can cause the queuing delay to fluctuate, making it seems like it ramps up, causing NADA to switch into gradual rate update mode, thus reducing the bitrate.

While GCC still manages to hold a link utilization between 77% - 79% in *RMCAT-NS3 Test*, it utilizes less bandwidth during *Sudden Reduction* and *Sweden to China* where the bandwidth limit changes promptly. GCC also shows a trend of increasing its bitrate slowly if previously subjected to low bandwidth limit. This is due to the bitrate only increasing by 5% when σ is set to Increase, or if the fraction of packets lost is considered negligible [6]. However, GCC has the highest link utilization during *Sweden to China*, compared to NADA and SCReAM. This is due to how GCC behaves when introduced to high delay. GCC utilizes the one-way delay gradient, which is used to compute a series of parameters which in turn becomes A_r. As seen in Equation 3.2, the delay gradient is computed as the difference between the difference of arrival times and send times of two successive packets. Even if high delay is introduced, as long as the packets are sent and arrive within a consistent time window, the delay gradient will remain low. Thus, GCC will gradually start to utilize more bandwidth despite being exposed to high delay. Despite the test environment having a jitter of 80 ms, it was not enough for GCC to malfunction and start behaving unpredictably. This could be due to the adaptive threshold subcomponent, which adapts the measured one-way delay gradient to delay variations.

SCReAM is currently the algorithm with the lowest link utilization compared to GCC and NADA, with exception to *RMCAT-NS3 Test*. It is also prone to fluctuations and overutilization as seen in Figures 5.1 and 5.3. When exposed to high delay, the algorithm is unable to properly function, causing the bitrate to fall down to 300 kbps and remain there for the remainder of the experiment, whether jitter is present or not.

Compared to NADA and GCC, which update the bitrate based on delay of recently received packets, SCReAM computes its queuing delay similar to LEDBAT [33]. It utilizes a baseline forward delay which is the lowest recorded forward delay throughout the entire stream, and is computed as the difference between the new forward delay and the baseline delay. This means that if delay or jitter is introduced, packets with high forward delay will be sent, causing SCReAM to receive a high queuing delay, and reduce the congestion window and send window. This in turn will lower the bitrate sent through the network.

5.4.3 Responsiveness

When it comes to responsiveness, the algorithms respond quickly to decreases of the bandwidth limit but vary when it came to increases. GCC was usually the slowest algorithm in terms of response, taking over 6 seconds in *Sudden Reduction* to respond to the recent change in bandwidth limit. This is mainly due to GCC only increasing its bitrate by 5% when bandwidth is available. Meanwhile, NADA and SCReAM (especially NADA) would increase much quicker due to their modes allowing for larger multiplicative increases. This allows NADA to take less than 2 seconds to respond to the recent change. As seen in Figure 5.8, NADA is also able to respond pretty well despite being constrained by high delay, albeit taking longer than during *Sudden Reduction*.

5.4.4 Packet Loss

The measured packet loss varies quite a bit between the different algorithms. Looking at Figures 5.3 and 5.5, GCC measured substantial loss over smaller windows, reaching as much as 50% loss. This is also seen in the paper [6], showing a substantial increase in loss over a small window but almost no loss other than that interval. The assumption is that this is caused by the pacer queue receiving a lot of packets too quickly, which causes it to build up and eventually losing packets due to overflow. However, in Figure 5.1, the packet loss is barely non-existent, as the bandwidth limit only transitions 500 kbps each step compared to *Sudden Reduction* and *Sweden to China* which transitions 1500 kbps each step.

For NADA, the packet loss is relatively low across all the tests, except for *RMCAT-NS3 Test* which had peaks as high as 17% as seen in Figure 5.1. Furthermore, NADA only suffers packet loss whenever the bandwidth limit is reduced to 500 kbps, as seen in Figures 5.1, 5.3, and 5.5. However, the presence of packet loss can cause issues as it looks to loss as a means of determining which mode to enter, assuming that any loss present means it remains in gradual rate update mode [32]. As the loss experienced is non-zero it fails to enter accelerated ramp-up mode and does not properly utilize the link. The values returned from the TWCC data showed occasional drops in arrival time, which was below the time measured for creation. This can be an issue in the parser or the data returned from the TWCC pipeline is incorrect. Furthermore, this issue could be exacerbated by the filter imposed on incoming TWCC packets which filters out packets with an arrival time less than that of the creation time, possibly caused either by desync in clocks and/or invalid input data from the TWCC system.

Finally, SCReAM was the best performing algorithm across all tests when it came to loss, based on the results from Figures 5.2d, 5.4d, and 5.6d. However, similar to NADA, whenever a loss event is detected, the algorithm will begin to reduce the target bitrate and congestion window, despite only losing a small fraction of packets.

5.4.5 Algorithm behavior

Each algorithm exhibits special properties when it is running on a video stream. In the research paper [7], the utilization in a simulated environment was similar to the results shown above, with NADA having strong utilization, GCC being more defensive on the uptake, and SCReAM as a mobile-optimized algorithm, utilizes a bit less of the uplink. Furthermore, as the link speed increased NADA, thanks to its rapid increase, allowed it to quickly increase the output based on the available link; however, overshooting and going back down at times. SCReAM showcased an oscillating behavior, as it increases its output only to lower it after over-utilization. NADA holds its output quite constant and converges quickly to a value a bit below the available link, quickly reacting to a change and dropping until eventually converging towards a new value.

5.4.6 Axis Camera

To test out the algorithm as a *Proof of Concept*, NADA was installed on an Axis IoT camera, using TWCC as feedback information. While it successfully compiled, ran and correctly transmitted data, it was not with issues. Initially, the issue was simply performance as data received from the receiver was not batched and therefore caused delays; batching and moving processing to one thread solved this issue. However, the CPU usage seen in 5.7 is quite high and increases towards the end, reaching up towards 80% total CPU usage, causing the camera to lag and fall behind in sending buffers. This can be seen if the pacer time is measured, going from taking a few milliseconds to over 1000ms elapsed which causes the stream to lag. One noticeable difference is that using higher resolution causes higher CPU usage overall. In terms of memory usage, this includes the entire WebRTC system, so the overall memory usage is not too bad. As the memory usage does not explode vertically there are no memory leaks evident.

Ultimately, the adaptive control ran well on real hardware. It controlled the bitrate of the underlying video provider, ensuring that not too much data is provided. It also successfully reacted to changes in the available network uplink, quickly slowing down the amount of data produced. However, issues such as long-running delays in the pacer queue were visible as well as high performance requirements did not make the current version ideal but definitely a good start to implement adaptive control in production.

5.4.7 Difficulties

This subsection specifies the difficulties encountered while obtaining the results.

5.4.7.1 Hardware limitations

While the tests ran without noticeable issues on the computer, running tests on the low-powered hardware ran into issues with performance, especially noticeable when the resolution and (therefore) complexity in video encoding increased. This was not an issue visible when running on the computers as the computation required was not an issue in that case given Table 5.1. Naturally the implementation is meant to simply test out the capabilities of running an adaptive algorithm on the device and therefore has many improvement possibilities; however, it did pose an issue and a challenge in its current form running on a low-powered device.

5.4.7.2 NADA and TWCC

In this thesis NADA is implemented using TWCC as source of feedback. The original implementation in the Gecko engine also relied on TWCC packet feedback. However, the system is different compared to that of the WebRTC implementation in the browser. During testing, the output from the TWCC parser showed, for instance, an arrival time (at the receiver) less than that of the packet's creation. While the clocks are not synced this is only visible in a few packets, as well as being drastically different from the other packets (less than 6000 ms below packet creation time). As the timestamps appeared incorrect rather than simply being received that long ago they were filtered out of the output, as it otherwise caused severe fluctuations in the bitrate output; although reduced the fluctuations are still visible when viewing individual tests.

5.4.7.3 Concurrency, Races and Software bugs

Most of the work to implement the plugins into Axis' system was done in Rust, which guarantees that safe Rust cannot cause race conditions and guarantees thread safety. However, due to the prevalence of foreign APIs this is no longer a guarantee as most of the time C code was called to invoke the algorithms. For instance, statistics are collected from a separate thread, but it calls methods on the same object used to perform the algorithm. As issues caused by data races are transparent (unlike e.g. segmentation faults which crash the program) and might not be immediately visible identifying the source is a non-trivial task. Furthermore, as the PoC is running on low powered device the efficiency of the running code also matters. Therefore, consistent lock contention via mutexes has to be avoided, which meant almost all statistics and information regarding runtime are instead stored in thread-safe atomic variables that have no locking needed to read and load.

As the area of adaptive bitrate control in real-time streaming is fairly new libraries are not flawless. To implement these plugins into Axis' systems a few bugs had to be dealt with. The default configuration of GStreamer TWCC did not properly handshake the capabilities between sender and receiver and therefore had to be manually inserted, to allow TWCC to run in the stream. Furthermore, NADA had a floating point comparison issue where a non-zero loss forced gradual rate update. However, as the loss used exponential drop-off a loss such as 10^{-30} would qualify as enough loss to stop rapid update; by setting the limit to instead a very low value (less than 0.5%) the issue disappeared, and the algorithm was able to enter rapid increase mode again which allowed better utilization. Previously, it remained in mode 1 for the duration of the test.

Furthermore, the signal used to notify TWCC feedback in NADA was accidentally reused many times. This caused a higher load on the CPU but did not cause any incorrect input to the algorithm as identical TWCC feedbacks were filtered out.

Chapter 6

Conclusion

This thesis evaluated three congestion control algorithms in an emulated testbed in Axis' systems, and attempted to run them directly on the camera hardware. Based on the results and the discussion, it is shown that NADA is the most optimal congestion control algorithm when working in environments where no form of delay or jitter exists. This is due to its excellent performance in terms of link utilization and response, where the network is not as exposed to network delays, such as Local Area Networks. The implementation of NADA used during the experiments also shows the ability to operate under high delay, provided that jitter is not introduced. GCC on the other hand, is the most optimal algorithm when working in environments where the link is more susceptible to latency and delay, such as Wide Area Networks. While defensive in terms of increasing its bitrate, it provides exceptional link utilization when exposed to high delay and jitter compared to the other two algorithms. GCC is also a generic solution algorithm, showing great performance in network environments where there exists little to no forms of network constraints. This makes GCC a great choice for networks where constraints can occur unpredictably and be volatile. For devices utilizing remote control, for instance a Pan-Tilt-Zoom camera, responsiveness and low-latency is important. If they are not suspect to high network delay SCReAM remained the most responsive in this use case, maintaining the lowest round-trip time. Otherwise, in the case of high latency without jitter, NADA and GCC maintained very similar latency.

While the camera implementation suffered from performance issues such as high CPU usage and long-running delays in the pacer queue, it showed that a congestion control algorithm could adapt the bitrate of a hardware video encoder and respond to change when bandwidth is limited. While not a perfect solution that is ready for production, it proves that congestion control algorithms can be implemented and used in Axis Cameras (and other potential solutions), as performance issues are solved and more scientific research is done within the area.

Although network speeds and latency get better with time and research, the induced increase of that utilization gives incentive to always keep congestion control in check, to avoid scenarios such as congestion collapse.

6.1 Future Work

The area of congestion control is an area under active research, with many papers, e.g. [7, 5], released in recent years. This offers up many possibilities to improve upon the work in this thesis as the area of congestion control in streaming media is fairly new. While this thesis evaluated and tested three candidates for congestion control in Axis' devices it is only an initial implementation and can be improved. For instance, optimizing algorithms for certain environments as they are configurable, something not explored in this thesis.

In this thesis, a simple topology was considered, testing the viability of a single camera connected to a receiver. However, there are situations where the topology is different and/or other metrics can be used to evaluate the efficacy on an algorithm. For instance, evaluating the fairness when other streams (video streams or other data communication) are present and seeing how the available link is balanced between sources. There is also work that can be done as an extension to this thesis, such as better optimizing the algorithms for Axis' systems and compare how they perform to this thesis.

While modern congestion algorithms perform well, there are improvements to be made, for instance improving latency [10] by extending the algorithms. Also, there are papers [12] that introduce new algorithms to compete against current congestion control algorithms, and designing completely new or derivatives from current algorithms as a better fit for the IoT environment is also an area of interest.

References

[1] Avi Networks, "Network Congestion," Accessed: 2022-04-06. [Online]. Available: https://avinetworks.com/glossary/network-congestion/

[2] https://www.freesoft.org/CIE/RFC/896/2.htm, accessed: 2022-5-24.

[3] R. Jesup and Z. Sarker, "Congestion Control Requirements for Interactive Real-Time Media," RFC 8836, Jan. 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc8836

[4] Axis Communications AB, "About Axis," Accessed: 2022-04-07. [Online]. Available: https://www.axis.com/about-axis

[5] B. Jansen, T. Goodwin, V. Gupta, F. Kuipers, and G. Zussman, "Performance Evaluation of WebRTC-Based Video Conferencing," *SIGMETRICS Perform. Eval. Rev.*, vol. 45, no. 3, p. 56–68, mar 2018. [Online]. Available: https://doi-org.ludwig.lub.lu.se/10.1145/3199524.3199534

[6] G. Carlucci, L. De Cicco, S. Holmer, and S. Mascolo, "Analysis and design of the google congestion control for web real-time communication (WebRTC)," May 2016, pp. 1–12.

[7] S. Zhang, "Congestion Control for RTP Media: a Comparison on Simulated Environment," *CoRR*, vol. abs/1809.00304, 2018. [Online]. Available: http://arxiv.org/abs/1809.00304

[8] G. Carlucci, L. De Cicco, C. Ilharco, and S. Mascolo, "Congestion control for real-time communications: A comparison between NADA and GCC," in *2016 24th Mediterranean Conference on Control and Automation (MED)*, 2016, pp. 575–580.

[9] M. Guerrero Viveros, "Performance analysis of Google Congestion Control algorithm for WebRTC," Master's thesis, Delft University of Technology, Nov. 2019.

[10] L. Wu, A. Zhou, X. Chen, L. Liu, and H. Ma, "GCC-beta: Improving Interactive Live Video Streaming via an Adaptive Low-Latency Congestion Control," in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, 2019, pp. 1–6.

[11] I. Johansson, S. Dadhich, U. Bodin, and T. Jönsson, “Adaptive video with SCReAM over LTE for remote-operated working machines,” *Wirel. Commun. Mob. Comput.*, vol. 2018, pp. 1–10, 2018.

[12] B. Kreith, V. Singh, and J. Ott, “FRACTaL: FEC-Based Rate Control for RTP,” in *Proceedings of the 25th ACM International Conference on Multimedia*, ser. MM ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1363–1371. [Online]. Available: https://doi.org/10.1145/3123266.3123373

[13] J. Flohr, E. Volodina, and E. P. Rathgeb, “FSE-NG for managing real time media flows and SCTP data channel in WebRTC,” in *2018 IEEE 43rd Conference on Local Computer Networks (LCN).* IEEE, 2018, pp. 315–318.

[14] R. R. Stewart, “Stream Control Transmission Protocol,” RFC 4960, Sep. 2007. [Online]. Available: https://www.rfc-editor.org/info/rfc4960

[15] J. Flohr and E. P. Rathgeb, “Reducing End-to-End Delays in WebRTC using the FSE-NG Algorithm for SCReAM Congestion Control,” in *2021 IEEE 18th Annual Consumer Communications Networking Conference (CCNC)*, 2021, pp. 1–4.

[16] L. Zhang, Y. Cui, M. Wang, Z. Yang, Y. Jiang, and Y. Cui, “Machine Learning for Internet Congestion Control: Techniques and Challenges,” *IEEE Internet Comput.*, vol. 23, no. 5, pp. 59–64, 2019.

[17] T. Litman, “Generated Traffic and Induced Travel: Implications for Transport Planning,” *Institute of Transportation Engineers Journal*, vol. 71, pp. 38–47, 04 2004.

[18] S.-W. Lee, S.-M. Choi, J.-S. Park, and M.-C. Park, “Empirical analysis of induced demand resulted from LTE service launching,” *J. Korean Inst. Commun. Inf. Sci.*, vol. 37, no. 8C, pp. 741–749, 2012, translated to English.

[19] A. Ghaffari, “Congestion control mechanisms in wireless sensor networks: A survey,” *Journal of Network and Computer Applications*, vol. 52, pp. 101–115, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804515000557

[20] C. Lim, “A Survey on Congestion Control for CoAP over UDP,” *International Journal of Internet, Broadcasting and Communication*, vol. 11, no. 1, pp. 17–26, Feb. 2019.

[21] A. P. Silva, S. Burleigh, C. M. Hirata, and K. Obraczka, “A survey on congestion control for delay and disruption tolerant networks,” *Ad Hoc Networks*, vol. 25, pp. 480–494, 2015, new Research Challenges in Mobile, Opportunistic and Delay-Tolerant Networks Energy-Aware Data Centers: Architecture, Infrastructure, and Communication. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570870514001668

[22] J. Postel, “User Datagram Protocol,” RFC 768, Aug. 1980. [Online]. Available: https://www.rfc-editor.org/info/rfc768

[23] M. Kihl and J. A. Andersson, *Datakommunikation och Nätverk.* Studentlitteratur, 2013.

[24] L. Rosencrance, G. Lawton, and C. Moozakis, “User Datagram Protocol (UDP),” Oct. 2021, Accessed: 2022-02-17. [Online]. Available: https://www.techtarget.com/searchnetworking/definition/UDP-User-Datagram-Protocol

[25] International Standard and International Electrotechnical Commission, “Information Processing Systems - Open Systems Interconnection - Transport Service Definition,” ISO/IEC 8072, Aug. 1996.

[26] H. Schulzrinne, S. L. Casner, R. Frederick, and V. Jacobson, “RTP: A Transport Protocol for Real-Time Applications,” RFC 3550, Jul. 2003. [Online]. Available: https://www.rfc-editor.org/info/rfc3550

[27] D. Singer, H. Desineni, and R. Even, “A General Mechanism for RTP Header Extensions,” RFC 8285, Oct. 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8285

[28] K. Lindros, “Real-Time Transport Control Protocol (RTCP),” Jun. 2014, Accessed: 2022-02-02. [Online]. Available: https://www.techtarget.com/searchunifiedcommunications/definition/Real-Time-Transport-Control-Protocol-RTCP

[29] S. Holmer, M. Flodman, and E. Sprang, “Draft-holmer-RMCAT-transport-wide-cc-extensions-01,” Oct. 2015. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-holmer-rmcat-transport-wide-cc-extensions-01

[30] S. Holmer, H. Lundin, G. Carlucci, L. D. Cicco, and S. Mascolo, “A Google Congestion Control Algorithm for Real-Time Communication,” Internet Engineering Task Force, Internet-Draft draft-ietf-rmcat-gcc-02, Jul. 2016, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-rmcat-gcc-02

[31] J. Uberti, “WebRTC Forward Error Correction Requirements,” RFC 8854, Jan. 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc8854

[32] X. Zhu, R. P. *, M. A. Ramalho, and S. M. de la Cruz, “Network-Assisted Dynamic Adaptation (NADA): A Unified Congestion Control Scheme for Real-Time Media,” RFC 8698, Feb. 2020. [Online]. Available: https://www.rfc-editor.org/info/rfc8698

[33] I. Johansson and Z. Sarker, “Self-Clocked Rate Adaptation for Multimedia,” RFC 8298, Dec. 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8298

[34] “sergio-mena/gecko-dev,” Accessed: 2022-04-10. [Online]. Available: https://github.com/sergio-mena/gecko-dev/tree/nada

[35] S. Shalunov, G. Hazel, J. Iyengar, and M. Kühlewind, “Low Extra Delay Background Transport (LEDBAT),” RFC 6817, Dec. 2012. [Online]. Available: https://www.rfc-editor.org/info/rfc6817

[36] C. Authors, “The difference between jitter and latency in webrtc,” Mar. 2018. [Online]. Available: https://www.callstats.io/blog/2018/03/07/difference-between-jitter-and-latency-webrtc

[37] ——, “What is containerization? containerization definition - citrix.” [Online]. Available: https://www.citrix.com/solutions/app-delivery-and-security/what-is-containerization.html

[38] I. C. Education, “What is Docker?” Jun. 2021. [Online]. Available: https://www.ibm.com/cloud/learn/docker

[39] “What is Docker?” [Online]. Available: https://opensource.com/resources/what-docker

[40] “tc(8) - Linux manual page,” https://man7.org/linux/man-pages/man8/tc.8.html, accessed: 2022-4-13.

[41] “WebRTC,” Accessed: 2022-04-14. [Online]. Available: https://webrtc.org/

[42] I. Grigorik, “WebRTC,” https://hpbn.co/webrtc/, Oct. 2013, accessed: 2022-4-20.

[43] “What is GStreamer?” [Online]. Available: https://gstreamer.freedesktop.org/documentation/application-development/introduction/gstreamer.html?gi-language=c

[44] “Foundations,” https://gstreamer.freedesktop.org/documentation/application-development/introduction/basics.html?gi-language=c, accessed: 2022-5-4.

[45] “EricssonResearch/scream,” Accessed: 2022-04-10. [Online]. Available: https://github.com/EricssonResearch/scream

[46] “yuangrongxi/razor,” Accessed: 2022-04-10. [Online]. Available: https://github.com/yuanrongxi/razor

[47] Z. Sarker, V. Singh, X. Zhu, and M. A. Ramalho, “Test Cases for Evaluating Congestion Control for Interactive Real-Time Media,” RFC 8867, Jan. 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc8867

[48] “Global Ping Statistics ← Stockholm and Hong Kong,” Accessed: 2022-04-14. [Online]. Available: https://wondernetwork.com/pings/Stockholm/HongKong

[49] “RTP Media Congestion Avoidance Techniques (rmcat),” Accessed: 2022-04-14. [Online]. Available: https://datatracker.ietf.org/wg/rmcat/documents/

Appendix A

Appendix

A.1 NADA

```rust
#[link(name = "nada")]
#[allow(dead_code)]
extern "C" {
    //Creates a new controller. cb represents a callback function that
    //fetches the time from a GStreamer element.
    fn NewController(cb: extern "C" fn(ptr: *mut super::nadatx) -> u64)
    -> NadaController;
    //Frees resources for a controller.
    fn FreeController(c: NadaController);
    //When feedback has been collected from TWCC. FeedbackFF uses #[repr(C)]
    //to have the same structure as in C, to allow easy conversion into
    //std::vec<Feedback> with 0 cost.
    fn Feedback(c: NadaController, batch: *const FeedbackFF,
    count: usize) -> bool;
    //Current estimate bitrate from the controller.
    fn Bitrate(c: NadaController) -> u32;
    //Current estimate round-trip time.
    fn RTT(c: NadaController) -> u64;
    //The clock used, as a gstreamer element. To be used for
    //time measurement.
    fn SetClock(c: *mut super::nadatx);
    //Clears and returns the clock, to be freed in Rust.
    fn ClearClock() -> *mut super::nadatx;
}
```

Listing A.1: NADA FFI in Rust.

A.2 Google Sender

```rust
#[link(name = "cc", kind = "static")]
#[link(name = "bbr", kind = "static")]
#[link(name = "estimator", kind = "static")]
#[link(name = "common", kind = "static")]
#[link(name = "pacing", kind = "static")]
extern "C" {
    #[allow(improper_ctypes)]
    fn sender_cc_create(
        trigger: *const u8,
        bitrate_cb: extern "C" fn(
            trigger: *const u8,
            bitrate: u32, loss: u8,
            rtt: u32),
        handler: *const u8,
        pace_send_func: extern "C" fn(
            handler: *const u8,
            id: u32,
            retrans: i32,
            size: usize,
            pad: i32,
        ),
        feedback_func: extern "C" fn(
            trigger: *const Gcctx,
            id: u32,
            ts: u64),
        queue_ms: i32,
    ) -> RazorController;
    fn sender_cc_destroy(cc: RazorController);
    fn sender_cc_heartbeat(cc: RazorController);
    fn sender_cc_add_pace_packet(cc: RazorController, packet_id: u32,
        retrans: i32, size: usize);
    fn sender_on_send_packet(cc: RazorController, seq: u16, size: usize);
    fn sender_on_feedback(cc: RazorController,
        feedback: *const u8, size: i32);
    fn sender_cc_update_rtt(cc: RazorController, rtt: i32);
    fn sender_cc_set_bitrates(cc: RazorController, min: u32,
        start: u32, max: u32);
    fn sender_cc_time() -> i32;
}
```

Listing A.2: Google Sender FFI in Rust.

A.3 Google Receiver

```rust
#[link(name = "cc", kind = "static")]
#[link(name = "estimator", kind = "static")]
#[link(name = "common", kind = "static")]
#[link(name = "pacing", kind = "static")]
#[link(name = "bbr", kind = "static")]
extern "C" {
    #[allow(improper_ctypes)]
    pub(crate) fn receiver_cc_create(
        min_bitrate: i32,
        max_bitrate: i32,
        packet_header_size: i32,
        handler: *const u8,
        send_feedback_func: extern "C" fn(
            handler: *const u8,
            payload: *const u8,
            size: usize),
    ) -> RazorController;
    pub(crate) fn receiver_cc_destroy(cc: RazorController);
    pub(crate) fn receiver_cc_heartbeat(cc: RazorController);
    pub(crate) fn receiver_cc_on_received(
        cc: RazorController,
        seq: u16,
        timestamp: u32,
        size: usize,
        remb: i32,
    );
    pub(crate) fn receiver_cc_time() -> i32;
    pub(crate) fn receiver_cc_update_rtt(cc: RazorController, rtt: i32);
    pub(crate) fn receiver_cc_set_max_bitrate(cc: RazorController, max_bitrate: i32);
}
```

Listing A.3: Google Receiver in Rust.

A.4 Example TWCC Code

```
#define RTP_TWCC_URI
"http://www.ietf.org/id/"
"draft-holmer-rmcat-transport-wide-cc-extensions-01"
GstRTPHeaderExtension *video_twcc;
//Creates the extension
video_twcc = gst_rtp_header_extension_create_from_uri(RTP_TWCC_URI);
//Fetches the video payloader by name "videopay"
GstElement* videopay = gst_bin_get_by_name(GST_BIN(m_pipeline), "videopay");
//Sets ID of TWCC to 3.
gst_rtp_header_extension_set_id(video_twcc, 3);
//Add the extension to the payloader
g_signal_emit_by_name(videopay, "add-extension", video_twcc);
```

Listing A.4: Example TWCC Code, written in C++.

Printed by Tryckeriet i E-huset, Lund 2022

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2022-872
http://www.eit.lth.se