Awesome title here

Managing Real- Time Video and Data Flows with Coupled Congestion Control Mechanism

Tobias Fladby

Thesis submitted for the degree of
Master in programming and system architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021

Awesome title here

Managing Real- Time Video and Data Flows with Coupled Congestion Control Mechanism

Tobias Fladby

Awesome title here

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Contents

List of Figures

List of Tables

Preface

Chapter 1

Introduction

1.1 Problem statement

1.2 Contributions

1.3 Research questions

Overall:

- Can two heterogenous control mechanisms be coupled? Will it improve overall performance?

Simplicity:

- Can such a mechanism be designed simple enough for widespread implementation? Moreover can it be easily integrated with other congestion control mechanisms?

Fairness:

- Will both flows get their allocated share of bandwidth when needed?
- Will the coupled flows be fair to other flows sharing the same bottleneck?
- Will the bandwidth be shared according to configured priority?

Delay:

- Can it reduce delay spikes?

Link utilization:

- Will link utilization be equal to a single flow using the full link?

Responsiveness:

- Will any of the congestion control mechanisms be more responsive to congestion in the network?

Packet loss:

- Will any of the flows experience less packet loss?
- Can it reduce packet loss spikes for the flows?

1.4 Organization

Chapter 2

Background

2.1 WebRTC architecture

2.1.1 Real- time communication

2.1.2 Standardization

2.1.3 Protocol Stack

2.1.4 User API

Services

RTCPeerConnection API

DataChannel API

2.1.5 Signalling

NAT

ICE Framework

TURN

STUN

SDP

2.1.6 Encryption

TLS

DTLS

2.1.7 Usage

2.1.8 Browser engine

Aquiring WebRTC statistics

2.2 Transport protocols

2.2.1 TCP

2.2.2 UDP

Message- oriented protocols

2.2.3 RTP and RTCP

2.2.4 SCTP

SCTP is a transport protocol offering many of the same services as TCP while also bringing additional features and improvements to the table. SCTP offers a point- to- point connection- oriented reliable delivery service while also using the same flow and congestion control algorithms as TCP.

As opposed to TCP, SCTP is message- oriented. An SCTP connection is called an association.

SCTP separates application data into chunks, each identified by a separate chunk header. These chunks are bundled into a single SCTP message that consists of an SCTP message header followed by several data chunks. A key feature here is that the data chunks are independently identified with a separate header, thus a single SCTP message can contain data from separate streams of application data. For example one stream being text messages and another being the transfer of a file in a messaging application. The advantage of this packet structure is that it means SCTP can support multi- streaming since it can send multiple data streams in parallel through a single SCTP association.

Multi- streaming means that an application can transmit several independent streams of data in parallel.

SCTP separates application data into chunks, each identified by a separate header. A single SCTP packet can contain several data chunks from different application data streams.

2.3 Congestion control

2.3.1 Loss- based congestion control

2.3.2 Delay- based congestion control

2.3.3 ECN

2.4 WebRTC Congestion controls

Video data by nature is large in size so transmitting it creates a lot of traffic. This makes real- time communication challenging because it requires low latency in order to assure a good user experience.

History and previous research [cite relevant stuff, like congestion collapse]has shown that protocols should employ mechanisms that limit the amount of data sent per second to a reasonable level in order to avoid congestion as well as keep the latency low.

2.4.1 Google Congestion Control

RTP by itself only provides simple end- to- end delivery services for multimedia[cite RTP standard], since real- time communication requires congestion control it must implemented on top of RTP. Chromium's WebRTC implementation uses an algorithm called Google Congestion Control [4] to provide the mechanism. It consists of two controllers, one loss- based and one delay- based. The loss- based controller located on the sender- side, uses loss rate, RTT and REMB[Cite REMB message definition] messages to compute a target sending bitrate. The delay- based controller can either be implemented on the receiver- side or sender- side. It uses packet arrival info to compute a maximum bitrate which is passed to the

loss- based controller. The actual sending rate is set to the minimum of the two bitrates.

The loss- based controller

The loss- based controller is run every time a feedback message from the receiver- side is received. If more than 10% of packets have been lost when feedback is received the controller decreases the estimate. If less than 2% is lost it will increase the estimate under the presumption that there is more bandwidth to utilize. Otherwise the estimate stays the same.

The delay- based controller

The delay- based controller consists of several parts: pre- filtering, an arrival- time filter, an over- use detector and a rate controller.

Pre- filtering is used to make sure that channel outages, events unrelated to congestion are not interpreted as congestion. Packets will naturally be delayed when a channel outage occurs so without this filter the algorithm would unnecessarily lower bitrate, thus lowering the quality of the communication for no reason. A channel outage will cause the packets to be queued in network buffers, thus when the channel is restored the packets will arrive in bursts. The filter utilizes the fact that the packet groups will arrive in bursts during a channel outage and merges them under such conditions.

The arrival- time filter is responsible for calculating the queueing time variation which is an estimation of how the delay is developing at a certain time. The goal of the over- use detector is to produce a signal that drives the state of the remote rate controller. The goal of the over- use detector is to compare the queueing time variation obtained as output from the arrival-time filter with a threshold. If the estimate is above the threshold for a certain amount of time and not sinking it wil signal the rate contol.

Performance

2.4.2 NADA

Network-Assisted Dynamic Adaptation (NADA) [8] specified in [9] is a proposed congestion control for WebRTC designed by Cisco. The key design goal of NADA is to offer both fast adaption time to congestion whilst also being able to compete with TCP flows sharing the same bottleneck. In order to achieve this, NADA combines loss, delay and ECN marked packets into a composite network congestion signal.

System overview

We will first give a basic overview of the central components of the system, then explain each of them afterwards more thoroughly. The live video encoder is a component responsible for encoding incoming raw video frames into RTP packets. It tries to output RTP packets at a rate as close

as possible to the target input rate R_v decided by the rate control. The actual output rate is denoted by R_o which will be a number within a range $[R_{min}, R_{max}]$ depending on the video scene complexity and can change over time. The value of the output rate R_o may fluctuate randomly around the input target rate R_v. On top of that the live video encoder is only capable of reacting to changes in R_v over long time intervals that might be in the order of seconds. The encoder's typical reaction time is denoted by T_v. The NADA sending agent has the responsibility of calculating a reference rate R_n based on the composite congestion signal reported by the receiver. The reference rate calculated by the sending agent is then used to regulate the video sending rate R_s. However there will be a difference between actual video encoder ouput and regulated send rate, a rate shaping buffer is used to absorb that difference. The size of the buffer L_s along with R_n determine the video encoder target rate and the sending rate. NADA is designed to work with nodes in different operation modes and it support many different queueing modes. The NADA receiving agent is responsible for calculating the composite congestion signal used by the sending agent to regulate the sending rate. To do this it uses derived one- way delay of each packet, loss events and ECN markings. The one- way delay D_n is derived from RTP header timestamps while ECN markings are extracted from the IP headers. The resulting composite signal is in the form of an "equivalent delay" d_n. A time smoothed version x_n of the signal is periodically reported back to the sending agent through RTCP messages.

Receiver agent

The receiver agent has four main tasks: a) Monitor one- way delay, packet loss and gather ECN marking statistics. b) Aggregate the different congestion signals into a composite network congestion signal. c) Calculate a time- smoothed value of the composite congestion signal. d) Send periodic reports of congestion to the sender.

a. Monitoring one- way delay, packet loss and gathering ECM marking statistics

b. Aggregating different congestion signals into a composite network congestion signal

c. Calculating a time- smoothed value

d. Sending reports of congestion to the sender

Sender agent

Performance

Usage

2.4.3 SCReAM

2.5 Coupled Congestion Control

2.5.1 Problems with combined controls

There are inherent differences in how quickly different types of congestion control react to congestion and combining them can therefore easily lead to unintentional side- effects. This is a known issue especially when it comes to combining loss- based controls with delay- based controls. In It has been shown to lead to...FIND A CONCRETE EXAMPLE TO REFER TO HERE

The main issue stems from the fact that delay always happens earlier than loss. The first thing that happens when there is congestion is that the bottleneck queues will start filling, thus making packets more delayed but not necessarily dropped until the queue is full or close to full. Intuitively, this means that delay is observable earlier than loss when there is congestion. The consequence of all this is that the delay-based control will decrease it's sending rate sooner than the loss- based will. Such behaviour leads to a very bad dynamic where the delay- based control lowers the bitrate at such an early stage of congestion that the loss-based never experiences packet loss and thus keeps increasing its send rate. The final result is that the delay- based control will get a smaller and smaller share of the available bandwidth because the loss- based control keeps increasing while the delay- based keeps backing down because of the congestion caused by the still- increasing traffic from the loss- based controller.

Coupling the controls A possible solution is to have the controllers cooperate by sharing their information i.e. coupling the flows. Such a mechanism could benefit both of the controllers by giving them more information, and different types of information to base their decisions on. This could also be used to ensure that the loss- based controllers behave more fairly to other types of controllers. One could for instance make sure that the loss- based controller also decreases its rate when the delay-based controller decreases the send rate. One of the oldest and most well-known mechanisms for coupling is "The Congestion Manager" (CM) [1]. CM couples flows by offering a single shared congestion controller for all the flows instead. The downside is that it is considered quite hard to implement because it requires an extra congestion controller and strips away all per- connection congestion control functionality, which is a drastic change.

A newer solution called "Coupled Congestion Control" [6] combines

congestion controls travelling over the same bottleneck while at the same time being easier to implement than the congestion manager. As opposed to The Congestion Manager, Coupled Congestion Control tries to utilize the flows' own congestion controllers by having them share information amongst each other instead of removing them. The mechanism has already shown promise in [5, 7] when implemented with homogenous congestion controls but has not been tested on heterogenous congestion controls.

Coupled Congestion Control Architecture The design philosophy of Coupled Congestion Control is that the amount of required changes to existing applications should be minimal. The system consists of three elements, Shared Bottleneck Detection(SBD), Flow State Exchange(FSE) and the flows.

2.5.2 The Flow State Exchange

The FSE can be described as a manager that maintains information exchanged between the flows and calculates a bit rate for each flow based on all the information gathered.

When a flow starts it registers itself with the FSE and SBD, when it stops it deregisters from the FSE and every time the congestion controller calculates a new rate the flow executes an UPDATE call to the FSE. When a flow registers itself the SBD will assign it to a Flow Group by giving it a Flow Group Identifier. A flow group is defined as a group of flows that share the same bottleneck and thus should exchange information with each other. The SBD is responsible for reassigning a flow to a different FG whenever the bottleneck changes.

The FSE component can be implemented in two ways: *active* or *passive*. In the active version, the FSE will actively initiate communication with each flow and SBD. The passive version does not actively initiate communication and only has the task of internal state maintenance. While the passive version has its own advantages and disadvantages and is relevant for other research projects it is beyond the scope of this paper.

Generally, the FSE keeps a list of all flows that have registered with it and for each flow the FSE will store the following:

- A unique number f to identify the flow.
- The Flow Group Identifier (FGI).
- The priority value P(f).
- The rate used by the flow which is calculated by the FSE in bits per second FSE_R(f).
- The desired rate of the flow, DR(f).

CC_R	Rate recevied from a flow's congestion controller
new_DR	Desired rate of a flow when it calls UPDATE
FSE_R	Rate allocated to a flow from the FSE
S_CR	Total sum of calculated rates for all flows in the samme FG
FG	A group of flows sharing the same bottleneck
P	The priority of a flow
S_P	Sum of all prioritites in a flow group
DELTA	Used to calculate the difference between CC_R and FSE_R

Table 2.1: Variables used in active FSE

The priority value P is used to calculate the flow's priority portion out of the sum of all priority values. The desired rate might be smaller than the calculated rate, e.g. because the application wants to limit the flow or simply does not have enough data to send. If there is no desired rate value given by the flow it should just be set to the sending rate provided by the flows congestion control.

For each FG the FSE keeps a few static variables:

- The sum S_CR of calculated rates for all flows in th FG.
- The sum S_P of all priorities in the FG.
- The total leftover rate TLO. This is the sum of leftover rate by rates limited by desired rate.
- Aggregate rate AR given to flows that are not limited by desired rate.

Every time a flow's congestion control normally would update the flow's rate they carry out an UPDATE call to FSE instead. Through the UPDATE call they provide their newly calculated rate and optionally a desired rate. Then FSE calculates rates for all the flows and sends them back. When a flow f starts, FSE_R is initialized with the initial rate calculated by f's congestion controller. After the SBD assigns the flow to an FG, it adds its FSE_R to S_CR. The desired rate is smaller than the calculated rate when the flow is limited by an application, otherwise it will be the same as the calculated rate.

2.5.3 Active FSE

In the active version FSE recalculates rates and notifies all the other flows in the FG as well whenever there is an UPDATE call from a single flow.

In [6] there are two examples of active FSE algorithms outlined. In table 2.1 the variables used in both algorithms are outlined.

Example algorithm 1 The first active FSE algorithm was designed to be the simplest possible method for assigning rates according to prioritites of flows. It consists of three steps:

1. When a flow f starts, it registers itself with SBD and the FSE. FSE_R(f) is initialized with f's congestion controllers' initial rate. SBD will also give f a correct FGI. After having received its FGI the FSE adds the FSE_R(f) to the flow group's corresponding S_CR.

2. When a flow f stops or pauses it gets removed from the list of registered flows.

3. Every time a flow f's congestion controller updates the send rate CC_R(f), i make an UPDATE call to the FSE. The UPDATE function completes four basic tasks (a- d) in order to calculate the new send rate for all flows in the same FG. A flow's UPDATE function uses three local (per- flow) temporary variables: S_P, TLO and AR.

UPDATE's tasks: a) First it has to update S_CR, this is done by adding the difference between CC_R(f) and the previous FSE_R(F) to the S_CR. b) Then it calculates a new S_P value and initializes FSE_R(f) values of all flows in the FG to 0. c) In the third step it distributes the S_CR among all the flows in the FG by calculating new FSE_R values, while also ensuring that each flow's desired rate is not surpassed. d) Lastly the FSE actively distributes the newly calculated FSE_R values to all the flows in the FG.

Even though algorithm 1 is simple and intuitive, it is shown to give both higher packet loss and queueing delay in [7] when tested with two heterogenous controls coupled. The authors concluded that the reason for these unsatisfactory results happened because the FSE *de-synchronizes* the flows. To illustrate the problem, consider some arbitrary congestion control that halves its send rate when experiencing congestion. The problem essentially was that normally without FSE, two flows will often naturally get synchronized and halve their rates at the same time, thus reducing the total rate by half. Now, if we consider the two flows coupled with the FSE again, since they are de-synchronized only one will halve its rate at a time thus only reducing the total rate a quarter and giving less of a chance for the queues to drain resulting in more queue growth and loss. As a consequence there was a new active FSE algorithm made in order to fix the loss ratio and average queue growth.

Example algorithm 2 The second algorithm aims to emulate a behavior similar to what happens when flows get synchronized. It does this by proportionally reducing the aggregate rate on congestion. Step 3a is altered by introducing the local variable DELTA which is used to calculate the difference between CC_R and previously stored FSE_R. To prevent flows from either ignoring congestion or overreacting, a timer is used to stop any flow form changing their directly after a shared rate reduction caused by a congestion event. The timer starts at two RTT's of the flow that experienced the congestion event. The reasoning for using two RTT's is that it is assumed that a congestion event may last up to one RTT for the

Algorithm 1: Active FSE - Example 1

```
/* Update S_CR                                                    */
S_CR = S_CR + CC_R(f) - FSE_R(f);
/* Calculate new S_P and initialize FSE_R(f)                      */
S_P = 0;
foreach flow f in FG do
    S_P = S_P + P(f);
    FSE_R(f) = 0;
end
/* Distribute S_CR among all flows                                */
TLO = S_CR;
while TLO > 0 and S_P > 0 do
    AR = 0;
    foreach flow f in FG do
        if FSE_R(f) < DR(f) then
            if TLO * P(f) / S_P >= DR(f) then
                TLO = TLO - DR(f);
                FSE_R(f) = DR(f);
                S_P = S_P - P(f);
            else
                FSE_R(f) = TLO * P(f) / S_P;
                AR = AR + TLO * P(f) / S_P;
            end
        end
    end
end
/* Distribute calculated FSE_R among all flows                    */
foreach flow f in FG do
    send(FSE_R(f), f)
end
```

Figure 2.1: Example algorithm 1 of active FSE

Algorithm 2: Active FSE - Example 2

```
/* Update S_CR based on DELTA                                    */
if Timer has expired or was not set then
    DELTA = CC_R(f) - FSE_R(f);
    if DELTA < 0 then
        /* Reduce S_CR proportionally                            */
        S_CR = S_CR * CC_R(f) / FSE_Rf;
        Set Timer for 2 RTTs;
    else
        S_CR = S_CR + DELTA
    end
end
```

Figure 2.2: Step a of Example algorithm 2's altered UPDATE function

flow, with the extra RTT added in order to compensate for fluctuations in measured RTT value. Except for step 3a where S_CR is updated based on DELTA, the rest of the algorithm remains the same as 2.1 on the facing page. The altered part of the algorithm is shown in figure 2.2

Another coupling mechanism called "Reduction of Self Inflicted Queueing Delay in WebRTC" (ROSIEE) [2] tries to reduce self- inflicted queueing delay in WebRTC by coupling NADA and the SCTP congestion control. As opposed to other mechanisms like [6] and [1] that try to explicitly control the congestion window they propose only calculating a max congestion window $CWND_{max}$ solely based on the rate calculated by NADA. The algorithm itself uses change in send rate ΔR_i and RTT_i received from NADA every time an RTCP message i is received to gradually converge on a maximum allowed SCTP sending rate that is later converted to $CWND_{max}$. Though the mechanism did in fact couple the WebRTC congestion controllers it did not provide the possibility to prioritize the different flows which is a key requirement for WebRTC. Accordingly they released another paper [3] that combines active FSE from [6] with the ROSIEE algorithm in order to also support prioritization of flows while still being able to couple and manage both loss- based and delay- based flows. As with original FSE, FSE-NG calculates a sum of rates and assigns it based on priority, but only based on the rate received from RTP flows. They also still don't use information from the loss- based flows when calculating the sum of rates. To calculate the upper limits for the SCTP flows they take a share of the sum of rates and split it amongst the SCTP flows.

2.6 Shared Bottleneck Detection

The SBD is an entity that is responsible for determining which flows are traversing the same bottleneck. In [6] three methods for deriving if flows share the same bottleneck are mentioned.

2.6.1 Multiplexed flows

One way is through comparing multiplexed flows. Since the flows with the same five- tuple will be routed along the same path, SBD can assume that they share the same bottleneck. However this method cannot be used for coupled congestion controllers with one sender talking to multiple receivers, given that they will not have the same five- tuple. Since WebRTC uses both SRTP and SCTP multiplexed on UDP, this ensures that they have the same five- tuple and that the first method will work.

2.6.2 Measurement

One might also use measurements of e.g. delay and loss and look at correlations to derive if flows have a shared bottleneck.

2.6.3 Configuration

Chapter 3

Design

3.1 Active FSE algorithm

Registering and de- registering flows

Converting SCTP's CWND to bit rate Since the Active FSE algorithm uses bit rate as means to measure available bandwidth we will have to convert SCTP's CWND to a bit rate before being able to use it.

Chapter 4

Implementation

4.1 Manager

The coupling mechanism itself will be implemented through a manager in the same manner as the FSE with an active FSE algorithm. However, the FSE described in [6] only supports homogenous delay- based flows and must be extended to support the coupling of both delay- based and loss-based congestion controls as in the case of WebRTC.

4.2 System Architecture

Manager

Shared Bottleneck Detection

IPC In order for flows in Chromium to register, de- register and complete UPDATE calls with our system we will use an IPC mechanism.

Chapter 5

Evaluation

5.1 Testbed

5.2 Experiments

Chapter 6

Conclusion

6.1 Research Findings

6.2 Further work

6.3 Closing remarks

Bibliography

[1] Hari Balakrishnan and Srinivasan Seshan. *The Congestion Manager*. RFC 3124. June 2001. DOI: 10.17487/RFC3124. URL: https://rfc-editor.org/rfc/rfc3124.txt.

[2] J. Flohr and E. P. Rathgeb. 'ROSIEE: Reduction of Self Inflicted Queuing Delay in WebRTC'. In: *2017 29th International Teletraffic Congress (ITC 29)*. Vol. 3. 2017, pp. 7–12. DOI: 10.23919/ITC.2017.8065803.

[3] 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)*. 2018, pp. 315–318. DOI: 10.1109/LCN.2018.8638084.

[4] Stefan Holmer et al. *A Google Congestion Control Algorithm for Real-Time Communication*. Internet-Draft draft-ietf-rmcat-gcc-02. Work in Progress. Internet Engineering Task Force, July 2016. 19 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-rmcat-gcc-02.

[5] S. Islam et al. 'Managing real-time media flows through a flow state exchange'. In: (Apr. 2016), pp. 112–120. ISSN: 2374-9709. DOI: 10.1109/NOMS.2016.7502803.

[6] Safiqul Islam, Michael Welzl and Stein Gjessing. *Coupled Congestion Control for RTP Media*. RFC 8699. Jan. 2020. DOI: 10.17487/RFC8699. URL: https://rfc-editor.org/rfc/rfc8699.txt.

[7] Safiqul Islam et al. 'Coupled Congestion Control for RTP Media'. In: *SIGCOMM Comput. Commun. Rev.* 44.4 (Aug. 2014). ISSN: 0146-4833. DOI: 10.1145/2740070.2630089. URL: https://doi.org/10.1145/2740070.2630089.

[8] Xiaoqing Zhu and Rong Pan. 'NADA: A Unified Congestion Control Scheme for Low-Latency Interactive Video'. eng. In: *2013 20th International Packet Video Workshop*. IEEE, 2013, pp. 1–8. ISBN: 1479921726.

[9] Xiaoqing Zhu et al. *Network-Assisted Dynamic Adaptation (NADA): A Unified Congestion Control Scheme for Real-Time Media*. RFC 8698. Feb. 2020. DOI: 10.17487/RFC8698. URL: https://rfc-editor.org/rfc/rfc8698.txt.