Received 2 August 2022, accepted 4 September 2022, date of publication 12 September 2022, date of current version 15 September 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3206041

RESEARCH ARTICLE

Real-Life Implementation and Evaluation of Coupled Congestion Control for WebRTC Media and Data Flows

SAFIQUL ISLAM[1], (Member, IEEE), MICHAEL WELZL[2], (Member, IEEE), AND TOBIAS FLADBY[2]
[1]Department of Business and IT, University of South-Eastern Norway, 3800 Bø, Norway
[2]Department of Informatics, University of Oslo, 0315 Oslo, Norway
Corresponding author: Safiqul Islam (safiqul.islam@usn.no)

ABSTRACT WebRTC enables users to simultaneously transfer media (over the Real-Time Transport Protocol (RTP)) and data (over the Stream Control Transmission Protocol (SCTP)) between web browsers, multiplexed onto a single UDP port pair. This design choice of using two different transport protocols, each with their own congestion control mechanism, can lead to competition between the flows, resulting in undesirable spikes in queuing delay and packet loss. In this paper, we investigate solutions to the harmful effects WebRTC flows cause on each other by having the different congestion controllers of the flows collaborate. Using implementations in the Chromium browser, we show that our mechanism can combine a set of heterogeneous congestion control mechanisms, fairly allocate the available bandwidth between the flows, and reduce overall delay and losses.

INDEX TERMS FSE, RTP, SCTP, congestion control, WebRTC.

I. INTRODUCTION

WebRTC enables interactive real-time communication between web browsers, supporting a range of applications such as video conferencing, telephony and interactive gaming. It allows a user to simultaneously transfer media (over the Real-Time Transport Protocol (RTP)) and data (over the Stream Control Transmission Protocol (SCTP)), multiplexed onto a single UDP port pair. Since routers or other middle-boxes usually identify flows using the five-tuple of source and destination IP addresses, ports and the transport protocol, such multiplexed flows are normally regarded as a single flow and hence they are treated in the same way by network elements.

The separate congestion control (CC) mechanisms within the two different transport protocols in WebRTC can lead to competition between the flows, resulting in undesirable spikes in queuing delay and packet loss. Such competition can be eliminated by using a coupled CC mechanism which combines the congestion control mechanisms of all the flows sharing a common path. In [1] and [2], we have shown that our coupling scheme called "Flow State Exchange" (FSE) can significantly improve the overall performance of multiple congestion-controlled RTP sessions in terms of delay and packet loss, and that it allows to exert a precise allocation of the available bandwidth. However, this mechanism only combines a set of homogeneous congestion control mechanisms and therefore cannot be readily applied to combine the data and video flows in WebRTC, since they use two different CC mechanisms: a delay-based CC mechanism for media and a loss-based CC mechanism for arbitrary data.

Because loss-based CC mechanisms fill the queue until packets are dropped, the competition between the flows leads to undesirable spikes in queuing delay and packet loss for the RTP flow. Combining a heterogeneous set of CC mechanisms can therefore yield several performance benefits, especially when one of the mechanisms reacts to a congestion event earlier than the others. This has been shown by Flohr *et al.* in [1], [3] with an extension of the FSE called "FSE Next Generation" (FSE-NG). WebRTC's

The associate editor coordinating the review of this manuscript and approving it for publication was Alba Amato.

delay-based RTP CC mechanism reacts to the increasing delay as soon as the queue grows, allowing the FSE-NG mechanism to react to this signal and ensure that the queue does not grow even for the loss-based CC mechanism in SCTP.

FSE and FSE-NG were implemented and evaluated in simulators only. Consequently, we consider it worthwhile to investigate if it is feasible to implement coupling mechanisms in the browser, and if we can replicate the promising results in an actual browser implementation. With this paper, we take this important step towards practical use of these mechanisms. Specifically, our contributions are:

1) We implement and test our FSE mechanism, as specified in RFC 8699 [4], in the Chromium browser. The FSE only operates on media flows, and it serves as a basis for the subsequent solutions.
2) We implement and evaluate the FSE-NG [5] mechanism in the Chromium browser. For this, we had to extend FSE-NG to work with Google Congestion Control (GCC) [6], which is the only RTP CC mechanism that is available in Chromium. Our implementation in the browser couples GCC and SCTP flows. We evaluate its efficacy in a real test-bed.
3) Based on the derived implications from FSE-NG, we propose, implement and evaluate an improved version of FSE-NG, called ‘‘Extended FSE-NG’’, which improves several aspects of the original FSE-NG mechanism.
4) Finally, we design, implement and evaluate a new mechanism called ‘‘Flow State Exchange v2’’, which is a re-design of the entire FSE idea to actively couple heterogeneous flow types. Using implementations in the Chromium browser, we show that our proposed mechanism works better than all prior works and does not exhibit problems that we encounter in FSE, FSE-NG and Extended FSE-NG.

This paper is organised as follows: section II presents the background and related work. Section III introduces our testbed and shows a fairness problem between the media and data channel in Chromium, highlighting the need for a congestion control coupling solution. Then, in sections IV to VII, we introduce the FSE and its derivatives in four steps: i) the original FSE, which operates on media flows only; ii) FSE-NG, which couples the media and data channels; iii) Extended FSE-NG, a novel algorithm which fixes some of the problems that we found with FSE-NG; iv) FSEv2, another novel algorithm, which is a complete re-design of the FSE idea such that it incorporates both the media and data channels. FSEv2 incorporates the lessons that we learned in the process of implementing and extending FSE-NG. We then evaluate FSE-NG, Extended FSE-NG and FSEv2 in section VIII using our implementations in the Chromium browser. Finally, section IX concludes.

II. BACKGROUND AND RELATED WORK

A. WebRTC

WebRTC [7] is a standard that comprises an extensive collection of protocols and Application Programming Interfaces (API), providing real-time peer-to-peer communication and data transfer between web browsers. Historically, there was a tendency for real-time communication software to rely on proprietary protocols and third-party plugins. WebRTC presents a break from this pattern, letting applications communicate unconstrained in the browser.

The WebRTC W3C Working Group[1] is responsible for defining the APIs that applications can use to control the communication via javascript. The IETF Working Group named Communication in Web-Browsers (RTCWEB)[2] is responsible for defining the protocols, data formats and other essential facets needed to enable real-time peer-to-peer communication in the browser.

A handful of protocols and technologies are imposed by what WebRTC needs to offer in terms of services and functionality. WebRTC uses the Real-time Transport Protocol (RTP) [8] for media transmission and the Stream Control Transmission Protocol (SCTP) [9] to transmit arbitrary application data. These protocols are multiplexed over a single User Datagram Protocol (UDP) [10] connection. While WebRTC requires that all data be encrypted, vanilla RTP and SCTP are not encrypted. Therefore, WebRTC uses SRTP [11] (a secure version of RTP) and encrypts SCTP. Datagram Transport Layer Security (DTLS) [12] is used for key management.

B. CONGESTION CONTROL MECHANISMS IN WebRTC

1) DATA CHANNEL

SCTP's CC is based on TCP's CC [9], [13], and is always applied to the entire SCTP association and not to individual SCTP streams. The transmission rate is determined by the receiver window (RWND) and congestion window (CWND), of which the minimum is used. RWND is the amount of data the destination side can receive. CWND is the amount of data the SCTP sender can transmit into the network before receiving an acknowledgement (ACK). As in TCP, the four central algorithms of SCTP's CC mechanism, which determine the value of CWND, are Slow Start, Congestion Avoidance, Fast Retransmit, Fast Recovery.

2) VIDEO CHANNEL

RTP alone provides simple end-to-end delivery services for multimedia. Therefore, WebRTC must also incorporate a CC mechanism for RTP. Currently, three different congestion control mechanisms are being considered for RTP flows in WebRTC: Google Congestion Control (GCC) [6], Network-Assisted Dynamic Adaption (NADA) [14] and Self-Clocked Rate Adaptation for Multimedia (SCReAM) [15]. In this paper, we only focus on GCC because it is used by two

[1] www.w3.org/groups/wg/webrtc
[2] https://datatracker.ietf.org/wg/rtcweb/about/

prominent web browsers: Chrome (with its open-source counterpart Chromium) and Firefox.

a: GOOGLE CONGESTION CONTROL (GCC)

Google Congestion Control (GCC) specified in [6] is a CC algorithm proposed by Google and is currently used in Chromium's implementation of WebRTC. It consists of two controllers, one loss-based and one delay-based. The loss-based controller located on the sender-side uses the fraction of packets lost, reported via RTCP REMB feedback messages [16] to compute a target sending bit-rate. The delay-based controller uses packet arrival information to compute a maximum bit rate. The delay-based controller can either be implemented on the receiver-side or sender-side. The delay-based estimate is passed to the loss-based controller, which compares it to its calculation and sets the actual send rate to the lowest of the two.

The delay-based rate controller can be seen as a state machine with three states: Increase, Decrease and Hold. Initially, it starts in the Increase state, where it stays until over-use or under-use is detected. The increase rate is multiplicative when estimated that convergence is far away and additive when it seems close to convergence. If the incoming bit rate is close to an average of incoming bit rates calculated the last time it was in the decrease state, it is assumed that the system is close to convergence. If there is no valid estimate of that average yet, the system remains in the multiplicative increase. When the over-use detector signals over-use, the system goes into the Decrease state. In the Decrease state, the bit-rate decreases with a certain factor multiplied by the currently incoming bit rate. If the detector then signals underuse, the system goes into the Hold state where the bit rate stays constant, allowing queues in the network to empty.

The loss-based controller bases its decisions on RTT, packet loss and the bit-rate calculated by the delay-based controller. The controller is run every time an RTCP feedback message from the receiver-side is received. If more than 10% of packets have been lost, the controller decreases the estimate. If less than 2% of packets are lost, it will increase the estimate under the presumption that there is more bandwidth to utilize; otherwise, the estimate stays the same. The actual bit rate used is the minimum of the delay-based and loss-based estimates.

C. RELATED WORK

One of the oldest and best known mechanisms for coupling is ''The Congestion Manager'' (CM) [17]. CM couples CCs by offering a single shared congestion controller for all the flows. The downside is that it is considered complicated to implement because it requires an extra congestion controller and strips away all per-connection CC functionality, which is a drastic change. Research has also been done on coupling TCP CC mechanisms [18], [19], [20], [21], [22]; however, these solutions are only relevant for TCP.

Our prior solution for RTP flows called ''Flow State Exchange'' (FSE) [4] combines congestion controls sharing the same bottleneck while at the same time being easier to implement than the CM. As opposed to CM, the FSE utilizes the flows' congestion controllers by having them share information amongst each other instead of removing them. The mechanism has already shown promise in [1] and [2] when implemented with homogeneous CC mechanisms but so far has not been tested on heterogeneous CC mechanisms.

Two other mechanisms stem from the original FSE implementation that try to couple NADA and SCTP flows. ''Reduction of Self Inflicted Queuing Delay in WebRTC'' (ROSIEEE) [3] is a mechanism that limits queuing delay in WebRTC by coupling NADA and the SCTP congestion control. As opposed to other mechanisms like [4] and [17] that control the congestion window explicitly, the authors of [3] propose to only calculate a maximum congestion window $CWND_{max}$ for SCTP based on the rate calculated by NADA. The algorithm itself uses the change in send rate ΔR_i and RTT_i—which is the RTT received from NADA every time an RTCP message i is received—to gradually converge to a maximum allowed SCTP sending rate that is later converted to $CWND_{max}$.

While this mechanism does, in fact, couple the WebRTC congestion controllers, it does not provide the possibility to prioritize the different flows, which is an essential requirement for WebRTC. Accordingly, FSE-NG [5] combines the active FSE from [4] with the ROSIEEE algorithm to support the prioritization of flows while still being able to couple and manage both loss-based and delay-based flows. As with the original FSE, FSE-NG also calculates a sum of rates S_CR and assigns it based on the priority of the flows in the FG. The mechanism does not use information from the loss-based flows when calculating S_CR. To calculate the upper limits for the SCTP flows, it shares S_CR and splits it amongst the SCTP flows in the FG.

III. A FAIRNESS ISSUE IN CHROMIUM's WebRTC IMPLEMENTATION

In this section, we introduce our testbed in section III-A which we use in all our tests, and then present a GCC vs. SCTP fairness issue by exploring how these two mechanisms compete under different network settings in section III-B. This problem further motivates the use of a coupled congestion control mechanism, on top of the earlier mentioned benefits attainable with congestion control coupling (lower delay and packet loss, and precise control over the per-flow rate share).

A. TESTBED

Figure 1 shows the topology used in our experiments. It consists of three physical machines: a WebRTC sender, a receiver and a router. The sender and receiver are both connected to the router with Ethernet cables. The three nodes are equipped with Linux version 5.11.0 (router), 5.13.0 (sender) and 5.15.18 (receiver). Two of the nodes are running one session each of the Chromium browser (Linux 64-bit 100.0.4896.12) with an instance of a WebRTC test application, acting as

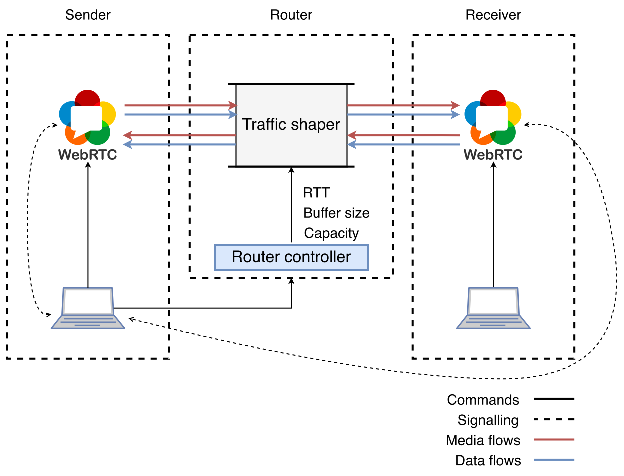


FIGURE 1. The testbed topology. Three nodes—one router performing traffic shaping and two nodes running Chromium—are connected through Ethernet cables. The sender node sends commands to the traffic shaper via ssh and hosts the signalling server, connecting sender and receiver applications.

sender and receiver. The receiver is only there to passively receive any streams coming from the sender and is running a release build of Chromium. The sender runs a build of the modified Chromium code, with an implementation of a coupled CC mechanism unless stated otherwise. The sender node is also hosting the Web server instance which performs the signalling and serves the WebRTC application. The web application is already loaded on the receiver side when an experiment begins. The sender node also acts as a testbed controller by using `ssh` to 1) set the link capacity, delay and the bottleneck queue size on the router node; and 2) start the sender application with a certain configuration, i.e. how many flows, which types of flows and when to start and stop them.

A video sequence with a resolution of 1280×720 and recorded in 60 frames per second is used for all video streams; this replaces the use of an ordinary webcam to achieve controllable conditions. Chromium allows us to replace the webcam source with the video being played in a loop. Sound is disabled for the media streams. We also set the video codec to be VP8 [23], which yields a maximum possible bitrate of 2.5 Mbps for the media streams.

B. RTP VS. SCTP FAIRNESS EVALUATION

Figure 2 shows three different experiments across a 10 Mbps capacity bottleneck with 50 ms RTT and a queue length configured to the Bandwidth×Delay Product (BDP), using different starting times for RTP and SCTP. In fig. 2a, with GCC and SCTP starting at the same time, GCC quickly can reach 2 Mbps and stay there for the duration of the experiment. In fig. 2b, GCC starts 10 seconds before SCTP and hardly seems affected, only experiencing a small dip in throughput when SCTP starts. Finally, fig. 2c shows a somewhat slow convergence before RTP reaches 2 Mbps; however, it does eventually adapt and stay at around 2 Mbps. We can see that, in general, GCC achieves a reasonably high throughput of around 2 Mbps regardless of which flow starts first, while the SCTP flow utilizes the rest of the link's capacity.

On the other hand, when limiting the bottleneck capacity to 5 Mbps, GCC is not able to compete with SCTP at all and is starved, as the plots in Figure 3 show. While most users from countries in the western world usually will have a much higher bandwidth than 5 Mbps and may therefore rarely notice this behaviour, it may be problematic for users in countries with poor internet service.

Recent performance evaluations of GCC [24], [25] show that GCC can aggressively compete against TCP-like congestion controls, which implies it should also be able to compete with SCTP. However, we could not replicate the same behavior in our testbed. As a sanity check, we tried to use the same topology and settings as described in [25], but GCC was still starved when competing with long-lived SCTP or TCP flows over the same bottleneck.

This problem motivates us to investigate if a coupled CC mechanism can fairly allocate the rates between SCTP and RTP on low capacity links.

IV. COUPLING, PART 1: THE FLOW STATE EXCHANGE (FSE)

We started our endeavor with an implementation and evaluation of the FSE in the Chromium browser. Being restrained to media flows, the direct benefits that can be attained with this first algorithm only apply to rather limited use cases, e.g. when simultaneously transferring video from a mobile phone's front and back camera. Use cases will become more realistic (e.g., screen/data and video sharing) when we come to the extensions of FSE that couple the media and data channels.

A. ALGORITHM

We briefly introduce the FSE algorithm since it serves as the basis for other coupling solutions in this paper. The FSE can be described as a manager that receives information from the different flows and calculates a new send rate for each flow based on all the information. When a flow starts, it registers itself with the FSE and a Shared Bottleneck Detection (SBD) element (in our case, simply the use of the same 5-tuple), and when it stops, it deregisters from the FSE. When a flow registers itself, the SBD will assign it to a Flow Group (FG) by giving it a Flow Group Identifier (FGI). A flow group is defined as a set of flows that share the same bottleneck and thus should exchange information with each other. Whenever a flow's congestion controller calculates a new rate, the flow executes an UPDATE call to the FSE with the newly calculated rate as a parameter.

Generally, the FSE keeps a list of all flows that are registered in it. For each flow, the FSE stores:

- A unique number f to identify the flow.
- The Flow Group Identifier (FGI).
- The priority value P(f).

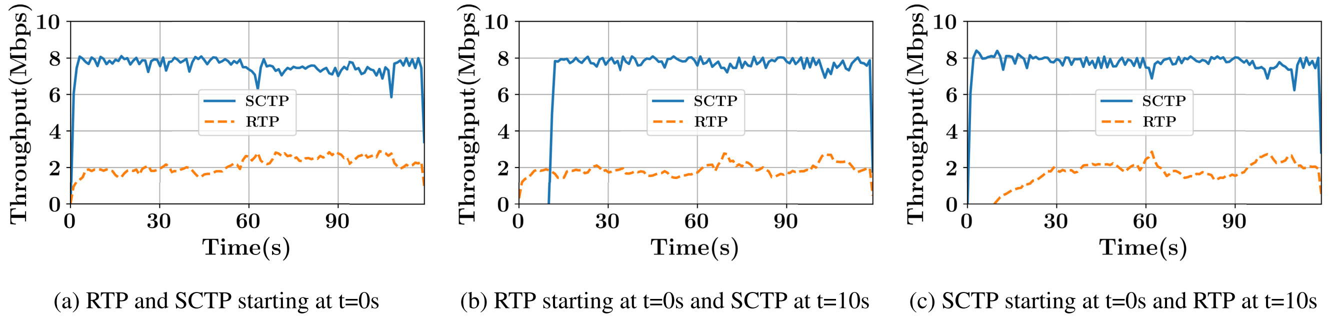


FIGURE 2. Throughput of SCTP and RTP (GCC) flows competing across a bottleneck with 10 Mbps capacity and 50 ms RTT with different starting times.

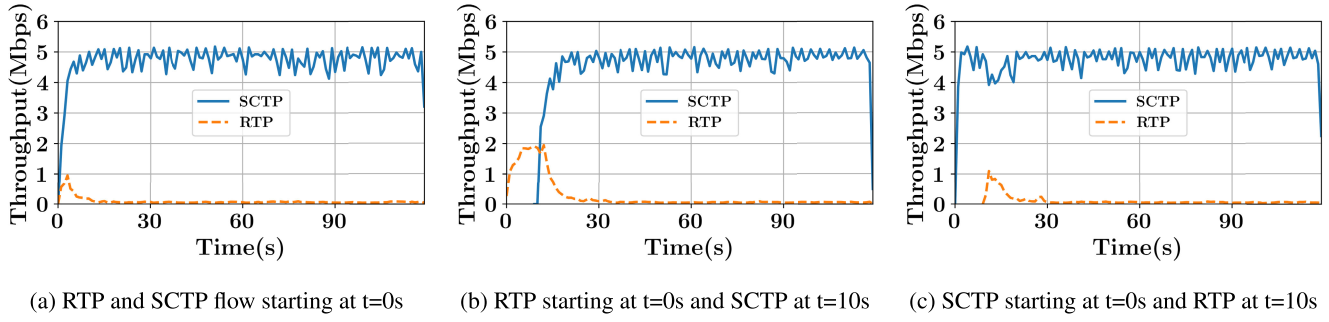


FIGURE 3. Throughput of SCTP and RTP (GCC) flows competing across a bottleneck with 5 Mbps capacity and 50 ms RTT with different starting times.

- 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).

The priority value is used to calculate the flow's rate via the priority portion of the sum of all priority values in the same FG (e.g., if flows *a*, *b* and *c* have priorities 1, 3, 1, respectively, then the rate assigned to flow *b* is 3/5 of the total and the others get 1/5). The desired rate might be lower than the calculated rate, e.g. because the application wants to limit the flow or simply does not have enough data to send. If the flow gives no desired rate value, it should just be set to the sending rate provided by the flows congestion controller. For each FG, the FSE keeps a few static variables:

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

Whenever a flow's congestion control normally updates the flow's rate, they carry out an UPDATE call to FSE instead. Through this call, they provide their newly calculated rate and optionally a desired rate. Then the FSE immediately calculates rate updates for all the flows and sends them back. When a flow f starts, FSE_R is initialized with the initial rate determined by f's congestion controller. After the SBD assigns the flow to an FG, it adds its FSE_R to S_CR.

TABLE 1. Variables used in the FSE algorithm from [4].

Name	Description
CC_R	Rate received from a flow's congestion controller
DR	The desired rate of a flow
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 same FG
FG	A group of flows sharing the same bottleneck
P	The priority of a flow
S_P	Sum of all priorities in a flow group
AR	Aggregate rate assigned to GCC flows that are not limited by a desired rate
TLO	Total leftover rate; sum of rate that could not be assigned to flows limited by a desired rate

In Table 1, the variables used in both algorithms are outlined. The pseudo-code for the update algorithm is shown in Algorithm 1.

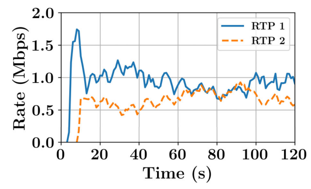
B. EVALUATION

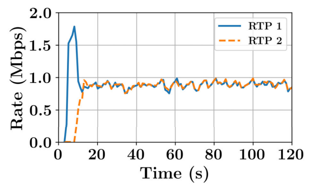
In this section, we first show results of coupling two RTP flows with the same priorities, then the results of two RTP flows with different priorities. Finally, we show the efficacy of the FSE solution when one of the RTP flow is rate-limited. These test cases have been designed by the IETF RMCAT Working Group[3] and are outlined in [26].

[3]https://datatracker.ietf.org/wg/rmcat/documents/

Algorithm 1 FSE Update Algorithm

```
1: function Update($flow$, $CC\_R$)
2:     $S\_CR \leftarrow S\_CR + CC\_R - FSE\_R(flow)$
3:     $S\_P \leftarrow 0$
4:     for $f$ in $FG$ do
5:         $S\_P \leftarrow S\_P + P(f)$
6:         $FSE\_R(f) = 0$
7:     end for
8:     $TLO \leftarrow S\_CR$
9:     $AR \leftarrow 0$
10:    while $TLO - AR > 0$ and $S\_P > 0$ do
11:        $AR \leftarrow 0$
12:        for $f$ in $FG$ do
13:            if $FSE\_R(f) < DR(f)$ then
14:                if $\frac{TLO \times P(f)}{S\_P} \geq DR(f)$ then
15:                    $TLO \leftarrow TLO - DR(f)$
16:                    $FSE\_R(f) \leftarrow DR(f)$
17:                    $S\_P \leftarrow S\_P - P(f)$
18:                else
19:                    $FSE\_R(f) \leftarrow \frac{TLO \times P(f)}{S\_P}$
20:                    $AR \leftarrow AR + \frac{TLO \times P(f)}{S\_P}$
21:                end if
22:            end if
23:        end for
24:    end while
25:    for $f$ in $FG$ do
26:        Update_CC($FSE\_R(f)$)$_f$
27:    end for
28:    $S\_CR \leftarrow 0$
29:    for $f$ in $FG$ do
30:        $S\_CR \leftarrow S\_CR + FSE\_R(f)$
31:    end for
32: end function
```





(a) Uncoupled (b) FSE

FIGURE 4. Sending rates of 2 RTP (GCC) flows across a 2 Mbps bottleneck, 100 ms RTT and 300 ms queue.

1) COUPLING TWO RTP FLOWS WITH THE SAME PRIORITIES

Figure 4 shows the sending rates of two RTP flows with and without the FSE. Priorities of both flows are set to 1 and flow 2 starts five seconds after flow 1. Without FSE, bandwidth usage is more sporadically divided between the flows. The FSE eliminates this problem by fairly dividing the rates between the flows (see fig. 4b)

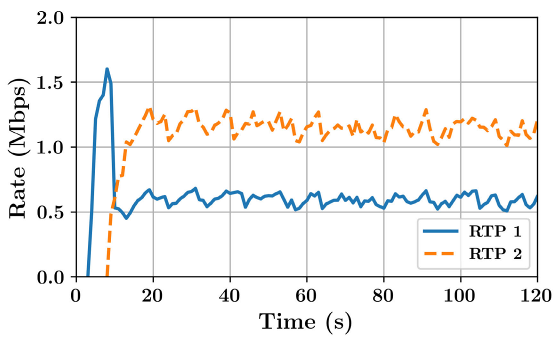


FIGURE 5. Sending rates of 2 coupled RTP (GCC) flows across a 2 Mbps bottleneck, 100 ms RTT and 300 ms queue. The priorities of flow 1 and 2 are 1 and 2, respectively.

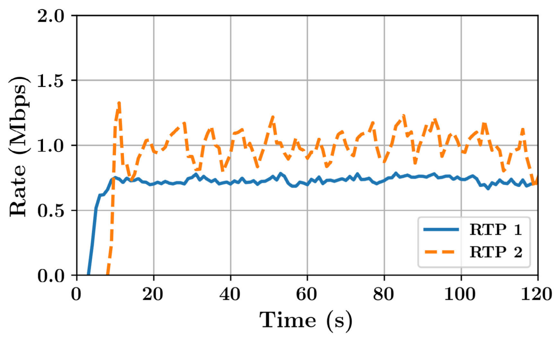


FIGURE 6. Sending rates of 2 coupled RTP (GCC) flows across a 2 Mbps bottleneck, 100 ms RTT and 300 ms queue. DR of RTP Flow 1 is set to 0.75 Mbps.

2) PRIORITISATION

The FSE can allocate rates based on the flows' priorities without requiring any modification in the congestion controller. This is shown in fig. 5 where the priorities of the two RTP flows are set to 1 and 2, respectively.

3) RTP FLOWS WITH DESIRED RATES

To show that flows limited by the desired rate share their leftovers with other flows, we ran an experiment with two flows, one with the desired rate configured to 0.75 Mbps and the other without a limited desired rate. Figure 6 shows that the first flow never exceeds 0.75 Mbps. The FSE allocates the leftover bandwidth to the second flow.

4) DERIVED IMPLICATIONS FOR THE FSE

As we have shown, the FSE works well for media flows—it improves fairness and offers possibilities for sharing leftover rates and prioritizing flows. However, the main problem plaguing WebRTC congestion control is the way SCTP affects GCC, which necessitates a mechanism that also incorporates SCTP flows. The most glaring limitation with the FSE is that it is *only* designed for media flows and therefore cannot directly be used to couple RTP and SCTP flows.

TABLE 2. Variables used in the FSE-NG implementation.

Name	Description
CC_R	Rate received from a registered GCC flow
last_rtt	RTT estimate received from a registered GCC flow
$CWND_{max}$	The current maximum CWND limit of an SCTP flow
FSE_R	The current rate allocated to a flow from the FSE
cwnd_flows	The set of registered SCTP flows
rate_flows	The set of registered RTP flows
DR	The current desired rate of an RTP flow
S_CR	Total sum of calculated rates for all flows
S_RTP_CR	The total sum of calculated rates for GCC flows
S_CWND_CR	The total sum of calculated rates for SCTP flows
P	The priority of a flow
S_P	Sum of all priorities in a flow group
S_CWND_P	Sum of all registered SCTP flows priorities
RTT_{base}	The lowest RTT that has been reported from any RTP flows
$Update\_CC_f$	The update callback function of some flow f
$R\Delta$	Relative rate change of a registered GCC flow

V. COUPLING, PART 2: FSE-NG

This section presents our design, implementation and evaluation of the FSE-NG mechanism in Chromium. Then we highlight some issues that we discovered during the evaluation. So far, FSE-NG was only designed and implemented in a simulator [27] to work with NADA [14] by its original authors. Since Chromium's RTP implementation uses GCC instead of NADA, we have to make some adjustments; we will also outline those.

A. DESIGN OVERVIEW

We try to stay as faithful to the pseudo-code and explanations in the original paper [5] as possible. However, we found some parts of the algorithm description in [5] to be ambiguous or lacking detail; for those cases, we choose the approach that seems to work best in practice.

The general structure of the new component is broadly similar to the FSE: registration, update and deregistration of GCC flows largely remain the same, but with a couple of necessary extra parameters for the update algorithm. Coupling SCTP flows brings about some new implementation aspects; most importantly, it means we must interact with the SCTP library in Chromium to get and set the maximum CWND limit. The following sections present explanations and some pseudo-code for the registration, update and deregistration phases. The implementation source code can be found in [28]. Table 2 provides an overview of the variables used in this section.

1) REGISTERING RTP FLOWS

Upon the first calculation of a new $CC\_R(f)$, a flow f registers in the **FseNg** (a class responsible for handling registrations, updates, and deregistrations for the GCC and SCTP flows) by sending in its $CC\_R(f)$ (as the initial rate), a flow priority $P(f)$, and a desired rate $DR(f)$. Moreover, the flow also registers a callback function called $Update\_CC(f)$. **FseNg** will call $Update\_CC(f)$ with $FSE\_R(f)$ as a parameter when it needs to update f. **FseNg** adds the initial rate to S_CR upon registration and creates a new **RateFlow** object to store $P(f)$, $Update\_CC$ and $DR(f)$. The set of **RateFlows** also stores a pointer to the **RateFlow** object. The callback function is simply a function that receives a rate and sets the value of the current estimate inside the GCC class that interacts with **FseNg**. The flows are also assigned a unique flow id by **FseNg**.

2) REGISTERING SCTP FLOWS

The WebRTC library uses a class called **UsrsctpTransport** to interact with the usrsctp library. In the class, there is a method called **Connect** which is called when a new SCTP association is being made. Accordingly, we choose to register SCTP flows in that method. Upon registration, the **Usrsctp-Transport** object of flow f sends in the initial $CWND_{max}(f)$, a callback function $Update\_CC(f)$ that gets called by **FseNg** to set $CWND_{max}(f)$ later, and lastly a flow priority $P(f)$. The $CWND_{max}(f)$ is stored so that **FseNg** may reset $CWND_{max}(f)$ in cases where all **RateFlow**'s deregister. In such a case there is no CC information being reported to **FseNG** and it should let SCTP flows use their default $CWND_{max}$. The SCTP flows are also assigned a unique flow id by **FseNg**.

3) DEREGISTERING FLOWS

As with the FSE implementation, **AimdRateControl** calls the deregister method of **FseNg** inside its destructor. **FseNg** removes the corresponding **RateFlow** from the set of **Rate-Flows**. **UsrsctpTransport** deregisters the corresponding SCTP flow inside a method called **CloseSctpSocket**. When a **PassiveCwndFlow** deregisters, **FseNg** simply removes it from the set of **PassiveCwndFlows**.

4) PERFORMING UPDATES

RTP flows send update information every time they calculate a new rate. This information includes the newly calculated rate CC_R and an estimate of the current RTT $last\_rtt$. For every update iteration, **FseNg** calculates a new rate for RTP streams, and a new $CWND_{max}$ for SCTP flows.

In line 2 of Algorithm 3, a function gets called that checks whether any GCC flows are application limited. The exact semantics of this function are not defined in [5]. However, we assume the function should check if all GCC flows f have been assigned an $FSE\_R(f)$ equal to $DR(f)$. The function is defined in Algorithm 2.

Algorithm 2 FSE-NG Checking If All RTP Flows Are Application Limited

```
1: function AllAppLimited
2:     for f in rate_flows do
3:         if $FSE\_R(f) \geq DR(f)$ then
4:             return false
5:         end if
6:     end for
7:     return true
8: end function
```

To find and update the *CWND_max* of any registered SCTP flows, we add some extra functions to the usrsctp library. After having calculated an FSE_R_f for a given SCTP flow f, $FSE\_R(f)$ is converted to $CWND\_max(f)$. **UsrsctpTransport** calls a function we added to usrsctp called **usrsctp_set_max_cwnd** that sets the *CWND_max* of the corresponding SCTP socket to the value given by **FseNg**.

Algorithm 3 FSE-NG Update Algorithm With Extensions

```
1:  function Update($flow$, $R\Delta$, $CC\_R$, $last\_rtt$)
2:      if IsEmpty(cwnd_flows) or AllAppLimited(rate_flows) then
3:          $S\_CR \leftarrow S\_CR + CC\_R - FSE\_R(flow)$
4:      else
5:          $S\_CR \leftarrow S\_CR + CC\_R - FSE\_R(flow) + R\Delta$
6:      end if
7:      $S\_RTP\_CR \leftarrow 0$
8:      for f in rate_flows do
9:          $FSE\_R(f) \leftarrow min(\frac{P(f)\times S\_CR}{S\_P}, DR(f))$
10:         Update_CC($FSE\_R(f)$)$_f$
11:         $S\_RTP\_CR \leftarrow S\_RTP\_CR + FSE\_R(f)$
12:     end for
13:     $RTT_{base} \leftarrow min(RTT_{base}, last\_rtt)$
14:     $S\_CWND\_CR \leftarrow S\_CR - S\_RTP\_CR$
15:     for f in cwnd_flows do
16:         $FSE\_R(f) \leftarrow \frac{P(f)\times S\_CWND\_CR}{S\_CWND\_P}$
17:         $CWND_{max}(f) \leftarrow FSE\_R(f) \times RTT_{base}$
18:         Update_CC($CWND_{max}(f)$)$_f$
19:     end for
20:     if IsEmpty(cwnd_flows) then
21:         $S\_CR \leftarrow S\_RTP\_CR$
22:     end if
23: end function
```

B. EXTENSIONS

Here, we detail our extensions that deviate from the algorithm description in [5]. The final version of the update algorithm is shown in Algorithm 3.

1) DEALING WITH EXCESS RATE

The original paper [5] does not specify how to handle situations where the *S_CR* is larger than the sum of desired rates, and there are no SCTP flows present. If we do not handle this case, there is excess rate at the end of the update; this leads to the *S_CR* growing every update call without allocating all the rate. Consequently, when an SCTP flow registers, all the leftover rate gets allocated, giving SCTP an almost unlimited $CWND_{max}$ because *S_CR* has grown too high. We therefore extend the FSE-NG algorithm with an extra check, as can be seen in lines 20-21 in Algorithm 3. If no SCTP flows are registered, the extension sets *S_CR* to *S_RTP_CR*. If there are SCTP flows registered, we know that any leftover rate has been allocated to the SCTP flows.

2) DESIRED RATE

The original FSE-NG algorithm uses the same *DR* for all the RTP flows; however, the maximum bit rate of RTP flows may vary. For instance, the WebRTC Javascript API offers an RTCRtpEncodingParameters object which lets the application set the maximum bit rate of the underlying RTP transmission of a mediaStreamTrack. Consequently, we extend the original algorithm by requiring each update call to also provide the flow's current DR. **FseNg** uses the individual flow's last reported DR instead of a shared global DR value when allocating bandwidth to the RTP flows.

3) CHOOSING *CC_R*

We make some adjustments when implementing the FSE-NG updates because of an inherent difference in how NADA and GCC work. NADA combines loss, delay, and ECN into a single aggregated value called "composite congestion signal" [14]. When coupling NADA flows in the FSE-NG, each NADA flow updates FSE-NG with the aggregated value as *CC_R* which FSE-NG then sets to *FSE_R* instead.

GCC, on the other hand, maintains two separate estimates, one based on loss (*As_hat*) and one based on delay (*A_hat*). The final rate used is $min(As\_hat, A\_hat)$. In the GCC implementation two classes are responsible for maintaining the estimates, **SendSideBandwidthEstimation** and **AimdRateControl**. **AimdRateControl** maintains the delay based estimate. **SendSideBandwidthEstimation** is responsible both for maintaining the loss-based estimate and then setting the final target based on the most conservative of the two values.

In our tests, we found that the *A_hat* will always be the most conservative value, and hence use *A_hat* as the rate that will be reported to FSE-NG.

C. EVALUATION

This section presents results from experiments performed with the evaluation testbed (see Section III-A). Firstly, we look at simple scenarios where the mechanism works as intended; then, we highlight some issues. We can trace some problems back to design flaws in the mechanism, while others arise because we implemented it with GCC while FSE-NG was originally designed to work with NADA. The IETF RMCAT Working Group developed test cases to evaluate real-time media flows in [26]. In accordance with [26], we use a bottleneck queue length of 300ms in all our tests (with the exception of fig. 11, as we will explain in section V-C2.a). We have also run tests with different queue lengths, which yielded similar results.

1) CORRECT OPERATION

a: COUPLING TWO RTP FLOWS

We start with the simplest case of two RTP flows to test the efficacy of the FSE-NG mechanism. Figure 7 shows that the effect of coupling two RTP flows with the FSE-NG is similar

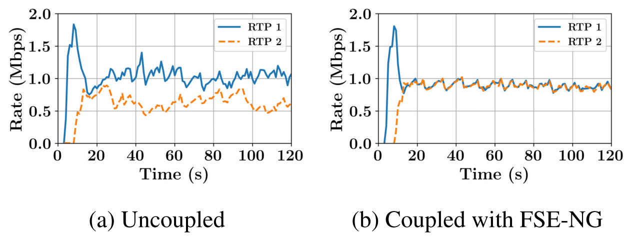


FIGURE 7. **Sending rates of 2 RTP (GCC) flows. The bottleneck has a capacity of 2 Mbps and a 300 ms queue. The RTT is 100 ms.**

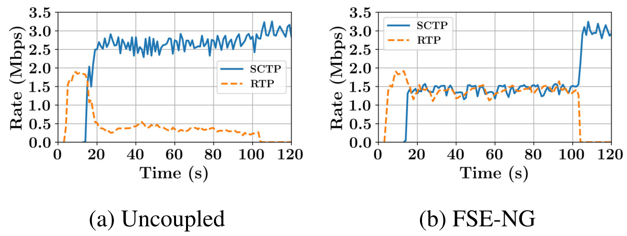


FIGURE 8. **Sending rates of one RTP (GCC) flow and one SCTP flow. The bottleneck has a capacity of 3 Mbps and a 300 ms queue. The RTT is 100 ms.**

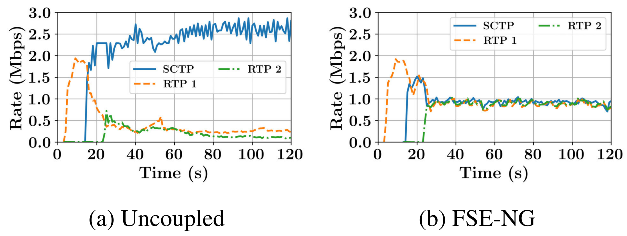


FIGURE 9. **Sending rates of two RTP (GCC) flows and one SCTP flow. The bottleneck has a capacity of 3 Mbps and a 300 ms queue. The RTT is 100 ms.**

to the previous FSE result. It can be seen from fig. 7b that the bandwidth is shared fairly between the FSE-NG-controlled media flows.

b: ONE RTP FLOW VERSUS ONE SCTP FLOW

The simplest case with heterogeneous flows is one RTP flow competing against one SCTP flow. fig. 8 illustrates the sending rates of an RTP and a SCTP flow, with and without the FSE-NG. The flows are given equal priority in the coupled scenario. The RTP flow starts 10 seconds earlier than SCTP and ends 20 seconds before. fig. 8a shows the fairness issue that we have identified in section III-B, where RTP gets starved almost entirely when running without the coupling mechanism. In contrast, fig. 8b clearly shows that coupling eliminates this problem, as flows are given their fair share of the total bandwidth when the FSE-NG is enabled.

c: TWO RTP FLOWS VERSUS ONE SCTP FLOW

We also ran similar experiments with two RTP flows competing with one SCTP flow. Again, we can see that, without coupling, the two RTP flows are unable to compete with SCTP and are starved completely (fig. 9a). fig. 9b shows that the capacity of 3 Mbps is fairly shared between all the flows so that each one gets 1 Mbps once all have started.

d: PRIORITISATION

To show that FSE-NG correctly handles and enforces priorities, fig. 10 presents sending rate plots of 2 RTP and 1 SCTP flows with different priority configurations. It can bee seen from fig. 10a to 10c that FSE-NG allocates rates based on the flows' priorities when the flow group is heterogeneous.

2) PROBLEMS

a: SCTP INITIALIZATION

At the beginning of an RTP connection, **AimdRateControl** has not yet measured an RTT and initializes the RTT to a default of 200 ms. We have also identified that it may take several seconds before an actual RTT is registered. In scenarios where there are registered GCC flows in the FSE-NG before any SCTP flows, this is not a problem since it gives ample time for GCC to find approximately the base RTT value. On the other hand, in cases where SCTP flows are registered before or simultaneously with any GCC flows, and the real base RTT is lower than 200 ms, the SCTP flow gets a much higher rate allocated by the FSE-NG than it should. This problem leads to GCC being out-competed by SCTP in the first few seconds of the transmission. The phenomenon is shown in fig. 11 with a high initial SCTP rate spike even though both flows are coupled and should be getting their fair share each. The RTT used in the figure is 50 ms, and, deviating from the common 300 ms configuration, we configured the router queue to 150 ms to ensure that the total measured RTT stays low enough. We also ran an experiment with a 100 ms RTT and found the same issue.

b: SCTP STARTING BEFORE RTP

Since FSE-NG only uses the congestion signals generated by GCC, S_CR stays at 0 as long as no RTP flows are registered, even though SCTP flows might be running and using a large share of the capacity. One side effect of this design is that when RTP flows start later than SCTP flows, the SCTP flow gets drastically pulled down when the an RTP flow registers. The phenomenon is illustrated in fig. 12—when the RTP flow starts, SCTP gets dragged down to around 750 Kbps; it should be close to 1.5 Mbps. The issue is that S_CR will start at the initial RTP rate, and the flows are both limited for some time until S_CR has grown enough for them to utilise the bandwidth. The impact of this problem could get slightly mitigated by the aforementioned RTT problem, because it accidentally gives SCTP a much higher $CWND_{max}$ than it is supposed to. However, this is not something that the FSE-NG mechanism is in control of, and it should therefore not rely on the default reported RTT being much higher than the RTT_{base}.

c: SLOW SCTP CONVERGENCE WHEN USING DR

When the desired rate limits the RTP flow, and the link's capacity is higher. the convergence time for SCTP becomes very long. GCC's delay-based controller has an ''increase'' state in which it may use an additive or multiplicative increase

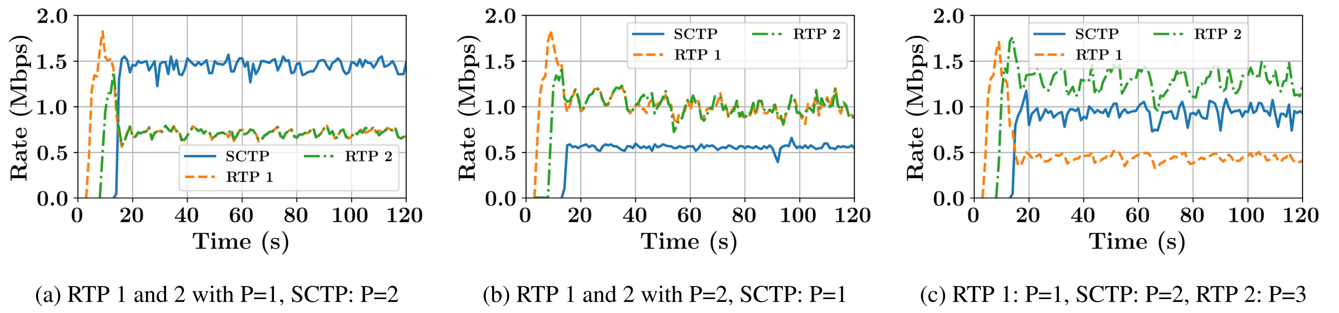


FIGURE 10. Sending rates of two RTP flows and one SCTP flow with different priority configurations. The bottleneck has a capacity of 3 Mbps and a 300 ms queue. The RTT is 100 ms.

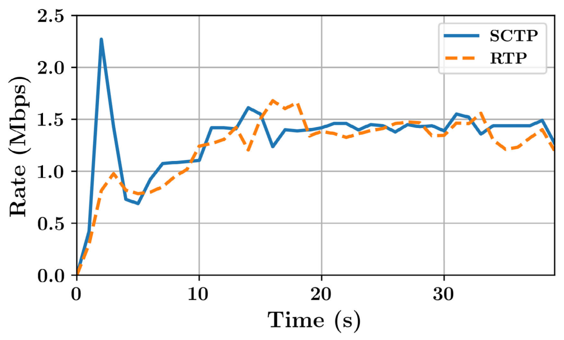


FIGURE 11. Sending rates of one RTP flow and one SCTP flow. The bottleneck has a capacity of 3 Mbps and a 150 ms queue. The RTT is 50 ms.

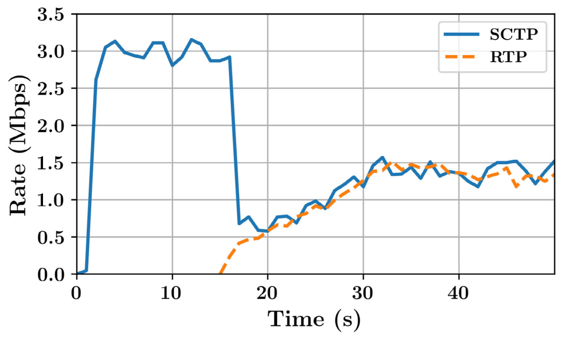


FIGURE 12. Sending rates of one RTP flow and one SCTP flow. The bottleneck has a capacity of 3 Mbps and a 300 ms queue. The RTT is 100 ms.

depending on how close to convergence the rate appears to be. In fig. 13, GCC is carrying out a multiplicative increase from time 0s to approximately time 40s. However, GCC's multiplicative increase is limited to a growth of 8% per second, so with a DR of 1.5 Mbps, GCC may only increase by $0.08\times$ 1.5 Mbps per second, which amounts to only a 120 kbps per-second increase factor. In other words, even though GCC is in the multiplicative increase state, it gets limited to an additive increase of 120 kbps per second because we are giving all the excess rate to SCTP flows instead and resetting GCC back to its DR. Naturally, this makes convergence for SCTP very slow; in the scenario of Figure 13 it takes approximately 30 seconds from SCTP's start until the total capacity of 6 Mbps is utilized.

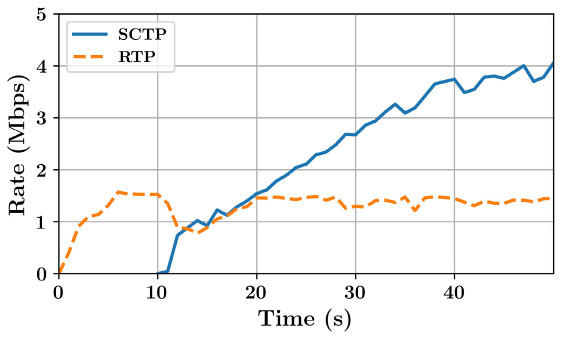


FIGURE 13. Sending rates of one RTP flow and one SCTP flow, RTP with DR set to 1.5 Mbps. The bottleneck has a capacity of 6 Mbps and a 300 ms queue. The RTT is 100 ms.

D. DERIVED IMPLICATIONS OF THE FSE-NG MECHANISM

This section summarizes the design issues and limitations of FSE-NG that we found when in our implementation using GCC.

- In the beginning, GCC has not yet gotten a realistic RTT report; therefore, it reports the default RTT of 200ms to FSE-NG; this leads to unfair bandwidth allocation between RTP flows and SCTP in the first couple of seconds (fig. 11).
- When SCTP flows start before any RTP flows, the FSE-NG will significantly throttle them once any RTP flow begins (fig. 12).
- Because GCC is limited to a maximum rate change of 8% no matter the conditions, only using GCC's rate as input leads to a very slow SCTP convergence when the link has a high capacity (fig. 13).
- Since FSE-NG originally was designed to assume that all RTP flows will have the same desired rate, it does not share leftovers between RTP flows in cases where one flow is limited to a given desired rate and another one is not (fixed by our extension described in section V-B2).

- In line 5 in Algorithm 3, the relative rate change is technically added to S_CR twice because the relative rate change $R\Delta$ is actually calculated via $CC\_R - FSE\_R$. We have not found the reason for this decision in [5] and therefore consider it a bug. However, it does help mitigate the problems discussed in sections V-C2.b and V-C2.c which would be more serious if the relative rate change was added once. On the other hand, it exacerbates the problem explained in Section V-C2.a. We decided not to change this part of the algorithm for our implementation since the advantages seem to outweigh the disadvantages; it is also more faithful to the original algorithm to leave it as is.
- It is also noteworthy that FSE-NG does not have any coupling effect in cases where several SCTP flows belonging to different SCTP associations are registered, and no GCC flows are present.

E. LESSONS LEARNED

The problem described in Subsection V-C2a underscores the benefits of using information from both types of controllers in update calls. For instance, letting the SCTP flows update the FSE-NG's RTT information would not conflict with the design decision that only RTP controllers should provide the rate updates and would fix that particular issue. The benefits of using information from both flow types are further driven home by the issues highlighted in V-C2c and V-C2b, which also are reactions to the mechanism only getting rate updates from one type of flow and likely may be fixed by getting information from SCTP flows as well.

VI. COUPLING, PART 3: EXTENDED FSE-NG

The basis for the following extensions is that we added SCTP information updates to the mechanism. SCTP updates are sent every time SCTP calculates a new CWND; what is included in the updates and how the FSE-NG uses the information will be explained in the following subsections. Table 3 provides an overview of the variables and functions used throughout this section when explaining the mechanism.

A. RTT INFORMATION

Because GCC needs some time before getting a real RTT measurement, SCTP flow could get a significant rate fluctuation in the beginning (see fig. 11). Our fix for this issue is to change the algorithm such that only SCTP will send RTT information to the manager instead of GCC. In addition to the initial $CWND_{max}$, SCTP sends in the current RTT estimate upon registration as well. SCTP also includes the RTT in update calls. Since SCTP registers upon the first new calculation of CWND, at which point it already has an accurate RTT measurement, this ensures that whenever any SCTP flows are registered, RTT_{base} has a proper value. Since RTT_{base} is only used to update $CWND_{max}$, it is not necessary to update it in the absence of SCTP flows.

B. SLOW SCTP CONVERGENCE

We also introduce fixes for the issues of SCTP being dragged down upon GCC registration (see fig. 12), and the slow SCTP convergence (see fig. 13): these issues stem from the fact that only GCC is responsible for the rate growth of both mechanisms. Consequently, both of these issues can be solved by also letting SCTP report a rate and add to S_CR growth.

Firstly, this fixes the issue of SCTP getting dragged down when it starts before any RTP flows because SCTP will already have converged to a rate reasonably close to the link's capacity, which then can be shared with the newly registered RTP flow. Secondly, the very slow SCTP convergence when the RTP flow is application limited is fixed because SCTP contributes to S_CR alongside RTP.

When both controllers contribute to S_CR, it grows much quicker. However, the FSE-NG's reduction of delay is based around only letting GCC control any rate increases. To ensure GCC is allowed to control any rate increase and keep the delay low when necessary, we therefore compromise by only adding SCTP's relative rate change under two conditions: 1) if there are no RTP flows registered, or 2) if S_CR is large enough to give all registered RTP flows their DRs. This fix also improves the initial start-up of the GCC flow because SCTP has already contributed to the aggregate S_CR.

C. FIXING THE DOUBLE RATE ADDITION TO S_CR

In the original algorithm, we discovered a bug which led to the rate change reported by GCC flows to be added twice when all GCC flows are application limited or when there are no registered SCTP flows. As we have discussed, this bug mitigated the slow SCTP convergence problem arising after a later-joining GCC flow, but since this problem is now fixed, it is also safe to eliminate this bug and ensure that the rate change is added only once.

D. IMPLEMENTATION DETAILS

The implementation of the Extended FSE-NG algorithm is very similar to FSE-NG's implementation; in this section, we will explain the extensions and changes.

1) EXTENDED FSE-NG CLASS DESCRIPTIONS

We reuse some classes from the FSE and FSE-NG. What follows is an overview of the new classes created for the module:

HybridCwndFlow
: This child class of **Flow** represents an SCTP flow that receives rate updates but may also send a rate to the Extended FSE-NG under certain conditions.

ExtendedFseNg
: This implements the Extended FSE-NG as a singleton class. It stores the same state as **FseNg**, but has a slightly changed update algorithm, and a new method allowing SCTP to also send rate updates.

TABLE 3. Variables and functions used in the extended FSE-NG.

	Name	Description
Var	CC_R	Rate received from a registered GCC flow
	CC_CWND	Cwnd received from a registered SCTP flow
	last_rtt	RTT estimate received from a registered SCTP flow
	CWND_{max}	The current maximum CWND limit of an SCTP flow
	FSE_R	The current rate allocated to a flow from the FSE
	cwnd_flows	The set of registered SCTP flows
	rate_flows	The set of registered RTP flows
	DR	The current desired rate of an RTP flow
	S_CR	Total sum of calculated rates for all flows
	S_RTP_CR	The total sum of calculated rates for GCC flows
	S_CWND_CR	The total sum of calculated rates for SCTP flows
	P	The priority of a flow
	S_P	Sum of all priorities in a flow group
	RTT_{base}	RTT that has been reported from any SCTP flows
	Update_CC_f	The update callback function of some flow f
Func	CwndFlowUpdate	Algorithm called when an SCTP flow sends an update
	RateFlowUpdate	Algorithm called when a GCC sends an update
	IsEmpty	Utility function, checks if a given set is empty
	AllAppLimited	Utility function, see pseudo-code in Algorithm 2

2) REGISTRATION, UPDATES AND DEREGISTRATION DETAILS

Here, we describe each stage of the coupling sequence, highlighting the difference between Extended FSE-NG and original FSE-NG.

a: REGISTRATION

Registration of GCC flows remains the same as in the FSE-NG implementation. Registration of SCTP, on the other hand, has changed. Upon registration, flow f sends in: initial $CWND(f)$, initial $CWND_{max}(f)$, initial $RTT(f)$, a callback function $Update\_CC(f)$, and flow priority $P(f)$. As with FSE-NG, the initial $CWND_{max}(f)$ is only stored for the occasion where all **Rateflows** deregister and the SCTP flow should use its default $CWND_{max}$. A **HybridCwndFlow** object is created and added to a set of **HybridCwndFlows**. **ExtendedFseNg**'s RTT_{base} value is also updated in the same fashion as for FSE-NG but this time with SCTP's $RTT(f)$ value. In either case, $RTT(f)$ is used to convert $CWND(f)$ to a rate added to S_CR. The intention is that since either the measured capacity is large enough to satisfy all GCC flows or there are no GCC flows to take care of, SCTP should be allowed to increase S_CR.

b: PERFORMING UPDATES

GCC's rate updates largely remain the same, except that SCTP is now responsible for sending in RTT measurements; thus, GCC only sends in the newly calculated rate. The update algorithm also remains the same, except that RTT_{base} is no longer updated because that is done when SCTP flows sends updates, which we will delve into next. Algorithm 4 includes pseudo-code for the update algorithm for rate-based flows.

Every time an SCTP flow f calculates a new CWND, it sends an update to **ExtendedFseNg** containing $CC\_CWND(f)$ and $last\_rtt(f)$. $last\_rtt(f)$ is used to update RTT_{base} in the same way as FSE-NG. Then $last\_rtt(f)$ is used to convert $CC\_CWND(f)$ into a rate. Lastly, similar to the registration algorithm, if all GCC flows are application limited or no GCC flows are registered, the newly calculated rate is used to update S_CR with the relative rate difference.

It is worth noting that though SCTP is now allowed to update S_CR, it does not trigger a recalculation or distribution of rates for other registered flows; this still only happens when GCC flows send updates. Algorithm 5 includes pseudo code showing what happens when SCTP flows send updates.

c: DEREGISTRATION

The deregistration phase for both types of flows remains the same as in FSE-NG. If all GCC flows deregister, the $CWND_{max}$ values for all SCTP flows are reset to their initial values.

Algorithm 4 Extended FSE-NG's GCC Update Algorithm

```
1:  function RateFlowUpdate(flow, CC_R)
2:      $S\_CR \leftarrow S\_CR + CC\_R - FSE\_R(flow)$
3:      $S\_RTP\_CR \leftarrow 0$
4:      for f in rate_flows do
5:          $FSE\_R(f) \leftarrow min(\frac{P(f) \times S\_CR}{S\_P}, DR(f))$
6:          Update_CC$(FSE\_R(f))_f$
7:          $S\_RTP\_CR \leftarrow S\_RTP\_CR + FSE\_R(f)$
8:      end for
9:      $S\_CWND\_CR \leftarrow S\_CR - S\_RTP\_CR$
10:     for f in cwnd_flows do
11:         $FSE\_R(f) \leftarrow \frac{P(f) \times S\_CWND\_CR}{S\_CWND\_P}$
12:         $CWND_{max}(f) \leftarrow FSE\_R(f) \times RTT_{base}$
13:         Update_CC$(CWND_{max}(f))_f$
14:     end for
15:     if IsEmpty(cwnd_flows) then
16:         $S\_CR \leftarrow S\_RTP\_R$
17:     end if
18: end function
```

Algorithm 5 Extended FSE-NG's SCTP Update Algorithm

```
1: function CwndFlowUpdate($flow$, $CC\_CWND$, $last\_rtt$)
2:     $S\_CWND\_CR \leftarrow S\_CR - S\_RTP\_CR$
3:     $RTT_{base} \leftarrow min(RTT_{base}, last\_rtt)$
4:     $CC\_R \leftarrow CC\_CWND \times RTT_{base}$
5:     if IsEmpty(rate_flows) or AllAppLimited(rate_flows) then
6:         $S\_CR \leftarrow S\_CR + CC\_R - FSE\_R(flow)$
7:     end if
8:     $FSE\_R(flow) \leftarrow CC\_R$
9: end function
```

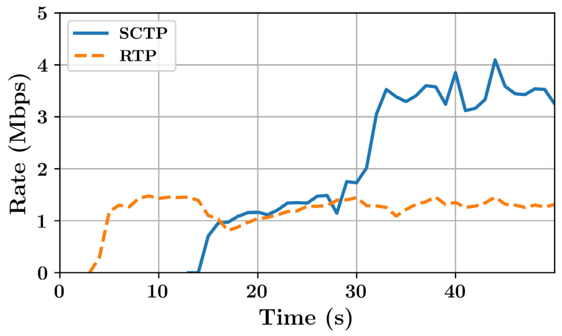


FIGURE 14. Sending rate of one RTP flow and one SCTP flow, RTP with DR set to 1.5 Mbps. The bottleneck has a capacity of 6 Mbps and a 300 ms queue. The RTT is 100 ms.

E. DERIVED IMPLICATIONS

Although some FSE-NG issues are now resolved, our tests of the Extended FSE-NG algorithm still show some limitations, necessitating yet another design step (accordingly, we will present evaluation results of Extended FSE-NG later, in section VIII, in comparison with the other variants). These limitations are:

- The introduced changes improve SCTP's slow convergence when there is enough capacity available to give GCC flows their desired rates. However, when this is not the case (in lower bandwidth conditions), SCTP will still experience slow convergence. This problem is illustrated in fig. 14, where it takes 15 seconds from the start of SCTP until S_CR has grown high enough for RTP to be allocated its DR of 1.5 Mbps. At that point our fix kicks in and lets SCTP add to S_CR, resulting in quick convergence from that point on.
- As with FSE-NG, the mechanism still does not share the leftover rate between GCC flows in cases where some flows are application limited and some are not.
- Extended FSE-NG cannot provide any coupling between separate SCTP flows (i.e., different SCTP associations) running alone, because SCTP update calls do not trigger rate updates from Extended FSE-NG to other flows.

F. LESSONS LEARNED

The original FSE-NG solely relies on the delay-based flow to drive the rate calculation and leaves SCTP passive. As we have discussed, a delay-based flow can also benefit from receiving information from a loss-based flow. However, FSE-NG is designed around the core concept that only the delay-based flow should lead, and therefore possibilities for incorporating SCTP updates are limited. This motivates us to investigate a different avenue where both types of flows are treated equally from the get-go.

VII. COUPLING, PART 4: FSEv2

We design and implement Flow State Exchange v2 (FSEv2), a new coupling mechanism for heterogeneous congestion control mechanisms that is based on lessons learned in the preceding sections.

A. DESIGN OVERVIEW

Our previously discussed extensions to FSE-NG mitigate some of FSE-NG's problems by using SCTP's rate changes under certain conditions. However, this works only for cases where capacity is large enough to accommodate all desired rates anyway. Furthermore, one potential problem with the FSE-NG mechanism is that it makes both types of coupled flows solely rely on GCC's ability to compete against other flows. To try a different approach, we base our new mechanism on the idea that the loss-based mechanism should be more active in the coupling process. That is, it should be allowed to contribute to rate changes, while still expecting that the delay-based mechanism will keep queuing delay down. The design is primarily based on the FSE mechanism but with support for loss-based mechanisms added. The basis for basing the update algorithm on FSE rather than FSE-NG is firstly that FSE-NG assumes all GCC flows have the same *DR*, thus not supporting sharing of leftover rate between GCC flows. Secondly, our mechanism couples SCTP in an inherently different way to FSE-NG by using the actual CWND of the flows as opposed to FSE-NG, which only sets the $CWND_{max}$ values for SCTP flows. Thus, FSE-NG concepts like, for instance, keeping track of RTT_{base} are no longer relevant. Loss-based flows are treated similarly to delay-based flows, except that we do not take any DR into consideration for them.

B. IMPLEMENTATION DETAILS

We now delve into more concrete details about the new mechanism and its implementation.

1) CLASS DESCRIPTIONS

The **RateFlow** is also reused for the new mechanism to represent GCC flows. Here is an overview of the new classes that we created:

ActiveCwndFlow
: This new type of flow class inherits from **Flow**. An object of this class is created for every registered SCTP flow.

FseV2
: This represents the new mechanisms manager, like the original **FlowStateExchange** class but extended to couple SCTP flows.

TABLE 4. Variables used in the FSEv2 implementation.

Name	Description
cwnd_flows	The set of registered SCTP flows
rate_flows	The set of registered GCC flows
S_CR	Total sum of calculated rates for all flows
FSE_CWND(f)	The CWND allocated to an SCTP by FSEv2
Update_CC$_f$	The update callback function of some flow f
CC_R	Rate received from a flow's congestion controller
CC_CWND	CWND received from a SCTP flow's congestion controller
DR	The desired rate of a GCC flow
FSE_R	Current rate allocated to a flow from the FSEv2
S_CR	Total sum of calculated rates for all flows in the same FG
FG	A group of flows sharing the same bottleneck
P	The priority of a flow
S_P	Sum of all priorities
AR	Aggregate rate assigned to flows that are not limited by a desired rate
TLO	Total leftover rate; sum of rate that could not be assigned to flows limited by a desired rate
last_rtt	The last reported RTT estimate from an SCTP flow

2) REGISTRATION, UPDATES AND DEREGISTRATION DETAILS

This subsection will go through the coupling sequence's registration, update, and deregistration parts. We reuse the same terms and variables as in the original FSE; some new concepts are introduced. All the variables and terms relevant to FSEv2 are listed in Table 4.

a: REGISTERING GCC FLOWS

Regarding registration, GCC flows are treated the same way as with the FSE: their initial rate is added to *S_CR*, and the newly created **RateFlow** is added to a set of **RateFlow** pointers.

b: REGISTERING SCTP FLOWS

When an SCTP flow f registers, it sends in its initial *CC_CWND*(f) value and its initial *last_rtt*(f) value. **FseV2**'s last RTT estimate is updated with *last_rtt*(f). *CC_CWND*(f) is converted to *CC_R* by dividing it by *last_rtt*(f). *CC_R*(f) is then added to *S_CR*. An **ActiveCwndFlow** object is created to store this information for the SCTP flow, and a pointer to the object is added to a set of **ActiveCwndFlow** pointers representing the flow group of registered SCTP flows.

c: GCC FLOW UPDATES

Whenever a GCC flow calculates a new rate, it sends an update to **FseV2** containing the newly estimated rate. **FseV2** then runs an update algorithm that is used by both types of flows. This algorithm updates *S_CR* and distributes *S_CR* among both types of flows.

d: THE UPDATE ALGORITHM

Algorithm 6 is used by both types of flows. The algorithm takes two arguments, the flow *flow* performing the update and the newly calculated CC rate *CC_R*(*flow*). If an SCTP flow is performing the update, we assume that it has already converted the reported CWND value to *CC_R*(*flow*). Firstly, *S_CR* is updated based on the *CC_R*(f) by adding the sum of the difference between *CC_R*(f) and the flow's previously allocated rate *FSE_R*(*flow*). Then, in lines 3-11, the algorithm calculates the total sum of priorities *S_P* by adding all priorities of both types of flows; it also initializes all allocated rates to zero. Next, in lines 14-32, the algorithm simultaneously allocates rates to all GCC and SCTP flows while ensuring that application-limited GCC flows do not get more than their desired rate. The leftover rate is shared fairly between all the other flows. The allocation for the GCC flows is the same as the original FSE algorithm (see Algorithm 1). However, an extra loop is added in lines 28-33. Finally, in lines 33-39, when all flows have been allocated a rate, the rates are distributed to the flows.

e: CHOOSING THE GCC UPDATE ESTIMATE

When **FseV2** sends updates to GCC flows, both the delay based estimate and the loss based estimates are updated with the *FSE_R*. The loss based estimate is calculated in a different GCC class called **SendSideBandwidthEstimation** than the delay based estimate, which is calculated in **AimdRateControl**. Both of these classes are controlled by yet another class called **GoogCc**, which is responsible for tying the various GCC components together. Accordingly, when **FseV2** sends updates to GCC, they are sent to **GoogCc** which then relays the information to **AimdRateControl** and **SendSideBandwidthEstimation** so that both estimates are updated to *FSE_R*.

f: SCTP FLOW UPDATES

When an SCTP flow f changes the CWND it sends an update to **FseV2** containing *CC_CWND*(f) and *last_rtt*(f). *CC_CWND*(f) is converted to *CC_R*(f) and Algorithm 6 is executed with *CC_R*(f) as input. **FseV2** sets the actual CWND when distributing rate updates. To accommodate for this we added another function called **set_cwnd** to the usrsctp library; **set_cwnd** is called from **UsrsctpTransport**.

g: SCTP STATE CONSIDERATIONS

Because the mechanism sets the actual CWND, which is tied to the state of SCTP's CC mechanism, we consider the following before setting CWND to *FSE_CWND*.

1) *Adopting FSE_CWND*: even though the actual CWND is stored in bytes in the usrsctp library, it only increases and decreases by the segment size number of bytes. Therefore, before setting the CWND, we round *FSE_CWND* down to the closest number of whole segments.
2) *Avoiding unnecessary Slow Start*: as opposed to SCTP's maximum CWND, which is simply an upper limit, the actual CWND is tied to SCTP's CC state. Specifically, if CWND is smaller than or equal to ssthresh, SCTP goes into the slow start phase;

Algorithm 6 FSEv2 Update Algorithm, Called When Either Type of Flow Sends Updates

```
function OnFlowUpdated($flow$, $CC\_R(flow)$)
    $S\_CR \leftarrow S\_CR + CC\_R - FSE\_R(flow)$
    $S\_P \leftarrow 0$
    for $f$ in $rate\_flows$ do
        $S\_P \leftarrow S\_P + P(f)$
        $FSE\_R(f) = 0$
    end for
    for $f$ in $cwnd\_flows$ do
        $S\_P \leftarrow S\_P + P(f)$
        $FSE\_R(f) = 0$
    end for
    $TLO \leftarrow S\_CR$
    $AR \leftarrow 0$
    while $TLO - AR > 0$ and $S\_P > 0$ do
        $AR \leftarrow 0$
        for $f$ in $rate\_flows$ do
            if $FSE\_R(f) < DR(f)$ then
                if $\frac{TLO \times P(f)}{S\_P} \geq DR(f)$ then
                    $TLO \leftarrow TLO - DR(f)$
                    $FSE\_R(f) \leftarrow DR(f)$
                    $S\_P \leftarrow S\_P - P(f)$
                else
                    $FSE\_R(f) \leftarrow \frac{TLO \times P(f)}{S\_P}$
                    $AR \leftarrow AR + \frac{TLO \times P(f)}{S\_P}$
                end if
            end if
        end for
        for $f$ in $cwnd\_flows$ do
            $FSE\_R(f) \leftarrow \frac{TLO \times P(f)}{S\_P}$
            $AR \leftarrow AR + \frac{TLO \times P(f)}{S\_P}$
        end for
    end while
    for $f$ in $rate\_flows$ do
        Update_CC($FSE\_R(f)$)$_f$
    end for
    for $f$ in $cwnd\_flows$ do
        $FSE\_CWND(f) \leftarrow FSE\_R(f) \times last\_rtt$
        Update_CC($FSE\_CWND(f)$)$_f$
    end for
end function
```

otherwise, it stays in Congestion Avoidance. It may happen that the **FseV2** sends a *FSE_CWND* value which is lower than ssthresh if, for instance, a GCC flow decreased the rate due to congestion. In the described scenario, if ssthresh remains unchanged and CWND is set to *FSE_CWND*, SCTP will go into Slow Start without having experienced a loss event because CWND will suddenly be smaller than ssthresh. To make sure this does not happen, we also set the ssthresh value to *FSE_CWND* minus the size of one segment in such cases.

h: DEREGISTRATION

Deregistrations in **FseV2** are treated in the same way for either type of flow. The flow is removed from the corresponding set of flows.

VIII. EVALUATION

We carry out experiments to evaluate the three heterogeneous CC coupling mechanisms that we presented in section V to VII. We present results from experiments performed with the evaluation testbed described in section III-A. The IETF RMCAT Group developed test cases to evaluate congestion control mechanisms for real-time media flows in [26]. Our test cases are inspired from [26] —- some cases are extended or modified to accommodate the fact that we are coupling heterogeneous flows. This section describes the general conditions surrounding the experiments. In accordance with [26], we use a bottleneck queue length of 300 ms in all our tests. The total run time of the experiments is 120 seconds. FIFO is used as the bottleneck queue type, and no artificial packet loss or jitter are added along the path.

We consider the following evaluation metrics:

- Sending rate, as observed by capturing the packets being sent on the sender node's interface with `tcpdump`.
- Throughput, as observed by capturing the packets arriving on the receiver node's interface with `tcpdump`.
- Bandwidth utilization, the ratio between available capacity and average throughput.
- Delay, gathered by logging the measured RTT in the GCC and SCTP Chromium code.
- Jain's fairness index [29]. When there are n flows, where x_i is the throughput for the ith flow, the fairness is rated with the following formula:

$$J(x_1, x_2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \times \sum_{i=1}^{n} {x_i}^2}$$

  The result ranges from $\frac{1}{n}$ to 1, with the former being the worst result and the latter being the best. A result of 1 means that all flows receive the same allocation, while a result of $\frac{1}{n}$ means that one flow receives all the allocation.
- RTP (GCC) packet loss. The number of RTP packets dropped during an interval of 500 ms is gathered through the WebRTC JavaScript API in the test application.

A. ONE GCC AND ONE SCTP FLOW COUPLED WITH EQUAL PRIORITIES

We begin by examining the behavior of two heterogeneous flows having equal priorities. This experiment aims to assert that the given coupling mechanism can solve the essential issue of assuring fairness between data and media flows in WebRTC. We expect the link to be shared fairly between the two flows in this test case. Specifically, Jain's fairness index should stay close to 1 whenever both flows are registered in the coupling mechanism and transmitting. We also expect

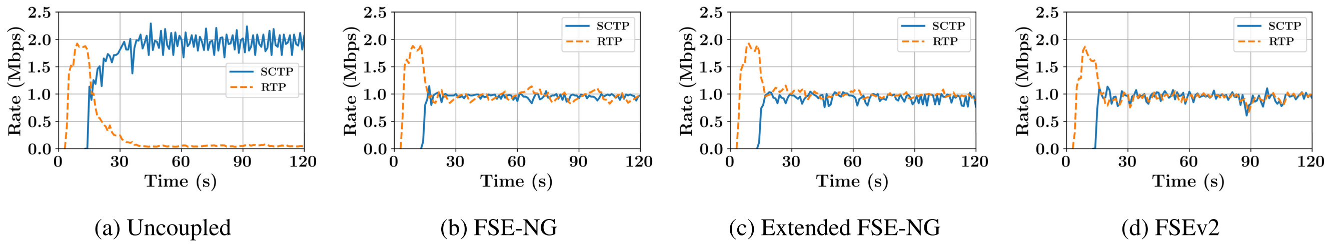


FIGURE 15. Sending rates one RTP (GCC) and one SCTP flow coupled with equal priorities. The bottleneck has a capacity of 2 Mbps and a 300 ms queue. The RTT is 100 ms.

TABLE 5. Average results based on 10 runs of the experiment with GCC (RTP) starting at t = 0s and SCTP starting at t = 10s. The bottleneck has a capacity of 2 Mbps and a 300 ms queue. The RTT is 100 ms. We only consider the time intervals when both flows are running at the same time.

Metric	Uncoupled	FSE-NG	Extended FSE-NG	FSEv2
Avg. GCC throughput (Mbps)	0.106	0.955	1.001	0.946
Avg. SCTP throughput (Mbps)	1.889	0.942	0.954	0.948
Avg. Cumulative Utilization	99.8%	94.9%	97.8%	94.7%
Jain's Fairness Index	0.556	1	0.999	1
Avg. GCC RTT (ms)	268	114	120	115
Avg. SCTP RTT (ms)	213	113	118	112

that the coupling mechanism prevents SCTP from filling the queue to make sure queuing delay is within acceptable levels.

Figure 15a shows the sending rates of an RTP and an SCTP flow without the coupling mechanisms. The bottleneck capacity and one-way delay are 2 Mbps and 50 ms, respectively. As Figures 15b, 15c and 15d show, coupling mechanisms effectively ensure fairness with all mechanisms, even between heterogeneous flows.

Table 5 shows metrics based on the average results. The most important thing to note is that all three coupling mechanisms can provide a Jain's Fairness Index score of 1 or very close to 1, while the case of uncoupled flows is close to the worst possible value of 0.5. We also ran the same test case with a bottleneck capacity of 1, 3 and 4 Mbps; it yielded the same close-to-perfect Jain's fairness index scores. Furthermore, the coupling mechanisms exhibit a much lower average RTT than the uncoupled case. The experiments were repeated ten times to ensure statistical significance.

The delay box plots in fig. 16 make it clear that the coupled CC mechanisms make the RTT vastly more stable for GCC when competing with an SCTP flow in all cases. Figure 16b shows that coupled CC also makes the RTT more stable for the coupled SCTP flow as well, which could be beneficial for WebRTC applications that, for instance, rely on the RTC-DataChannel for interactivity. We also checked the results for different queue lengths, and the results were the same for the coupling mechanisms.

It is worth mentioning that this is the only test case where some negligible (0.0016% of sent packets) GCC losses were observed for one of the coupling mechanisms. A couple of packets get dropped for FSEv2 when SCTP starts. The losses occur due to SCTP's Slow Start phase because SCTP is too aggressive for GCC's decreases to stop the queuing filling up. Once SCTP goes into Congestion Avoidance, no more losses are observed.

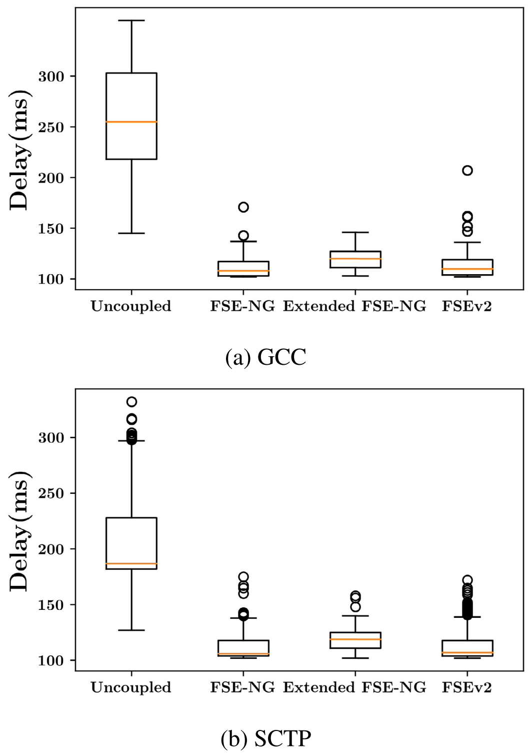


FIGURE 16. RTT measurements with equally prioritized flows. The bottleneck has a capacity of 2 Mbps and a 300 ms queue. The RTT is 100 ms. We only consider the cases when both flows are running at the same time.

B. PRIORITISATION

1) ONE GCC AND ONE SCTP FLOW COUPLED, WITH DIFFERENT PRIORITIES

This experiment aims to evaluate how well the coupling mechanism adheres to configured priorities when both types of flows are registered. A GCC flow is given a priority level

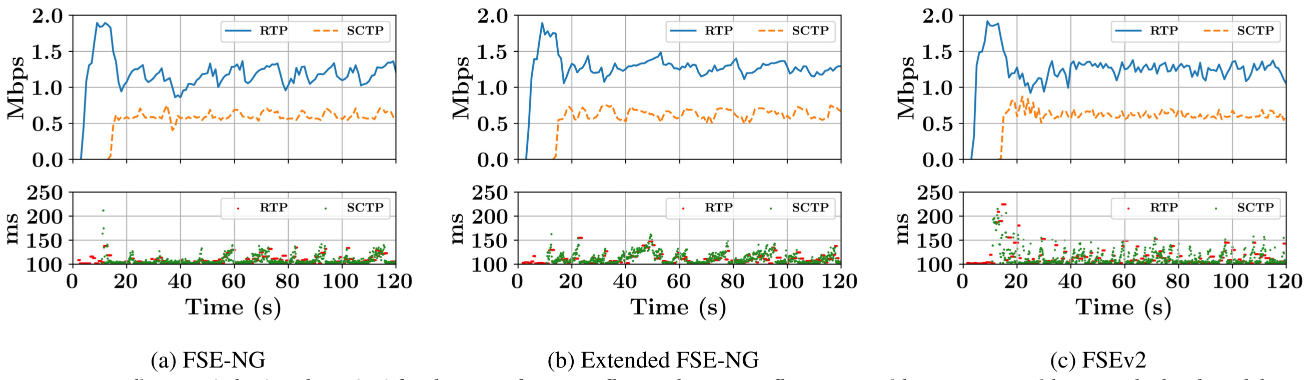


FIGURE 17. Sending rate (Mbps) and RTT (ms) for the case of one RTP flow and one SCTP flow. RTP 1 with P = 2, SCTP with P = 1. The bottleneck has a capacity of 2 Mbps and a 300 ms queue. The RTT is 100 ms.

TABLE 6. Average results based on 10 runs of the test case when GCC has P = 2 and SCTP has P = 1, the bottleneck has a capacity of 2 Mbps, 100 ms RTT and a 300 ms queue. We only consider the time intervals when both flows are running at the same time.

Metric	FSE-NG	Extended FSE-NG	FSEv2
Avg. GCC throughput(Mbps)	1.124	1.23	1.215
Avg. SCTP throughput(Mbps)	0.596	0.626	0.626
Avg. GCC share of total throughput	66%	66%	66%
Avg. SCTP share of total throughput	34%	34%	34%
Avg. Cumulative Utilization	86%	93%	92%
Avg. GCC RTT(ms)	108	112	115
Avg. SCTP RTT(ms)	107	111	111

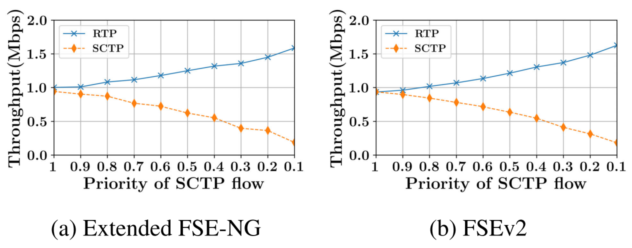


FIGURE 18. Throughput of a GCC flow and an SCTP flow coupled with Extended FSE-NG and FSEv2 mechanisms, with the GCC priority set to 1 while the SCTP flow's priority varies from 1 to 0.1. We only consider the time intervals when both flows are running at the same time.

twice as large as an SCTP Flow. Firstly, the coupled flows should reach a steady state where their rate equals their share corresponding to the configured priority. Secondly, the coupling mechanism should achieve this while still ensuring an appropriate level of bandwidth utilization. Consequently, it is expected that the GCC should comprise 2/3 or approx. 66% of the total bandwidth utilization while SCTP should utilize the remaining 33%. Meanwhile, the queuing delay should stay the same no matter what type of flow is being prioritized. The bottleneck capacity is 2 Mbps and one-way propagation delay is 50 ms.

Table 6 includes various metrics calculated by taking the average values based on 10 experiments. The table results show that all mechanisms can enforce the prioritization policy well since GCC gets 2/3 of the total throughput for all mechanisms.

Figure. 17 shows sending rates and RTT from one experiment. All three mechanisms do seem to experience some rate oscillations. For FSE-NG and extended FSE-NG, the oscillations follow a pattern of being stretched for longer periods, though FSE-NG's oscillations are more extreme. In the case of FSE-NG, this is due to the bug which adds GCC's rate increases and decreases twice (see section V-D). Because rate decreases become twice as large, FSE-NG has a much lower bandwidth utilization than the other mechanisms. However, we can see that FSE-NG keeps the delay lowest; extended FSE-NG experiences a bit more delay while FSEv2 has the most delay. Tests with larger queues also provided similar or very close results.

FSEv2's extra delay can be traced back to the fact that SCTP is also allowed to send rate updates to the manager; however, GCC seems to prevent SCTP from increasing the delay too much, making sure the delay is within an acceptable range. As fig. 17c shows, this also leads to quite a large initial delay increase when the SCTP flow is in the slow start phase.

Figure. 18 shows the the throughput for both flows when GCC's priority is set to 1 and SCTP's priority varies from 1 to 0.1. The throughput values for each different priority configuration in the plot are based on the average value of 10 different runs to ensure statistical significance. The mechanisms are able to distribute the rate according to varying priorities.

2) TWO GCC FLOWS AND ONE SCTP FLOW COUPLED, WITH DIFFERENT PRIORITIES

In this test case, two GCC flows run in parallel with one SCTP flow. The GCC flows are given extra priority of 50%

TABLE 7. Average results based on 10 runs of the test case when GCC has P = 1.5 and SCTP has P = 1. The bottleneck has a capacity of 4 Mbps and a 300 ms queue. The RTT is 100 ms. GCC flows are not shown on separate lines since their values are close to identical. Only time intervals when all flows are running are considered.

Metric	FSE-NG	Extended FSE-NG	FSEv2
Avg. GCC flow throughput(Mbps)	1.299	1.341	1.343
Avg. SCTP throughput(Mbps)	0.941	0.918	0.977
Avg. GCC flow share of total throughput	37%	37%	37%
Avg. SCTP share of total throughput	26%	26%	27%
Avg. Cumulative Utilization	88%	90%	92%
Avg. GCC RTT(ms)	108	109	115
Avg. SCTP RTT(ms)	107	107	113

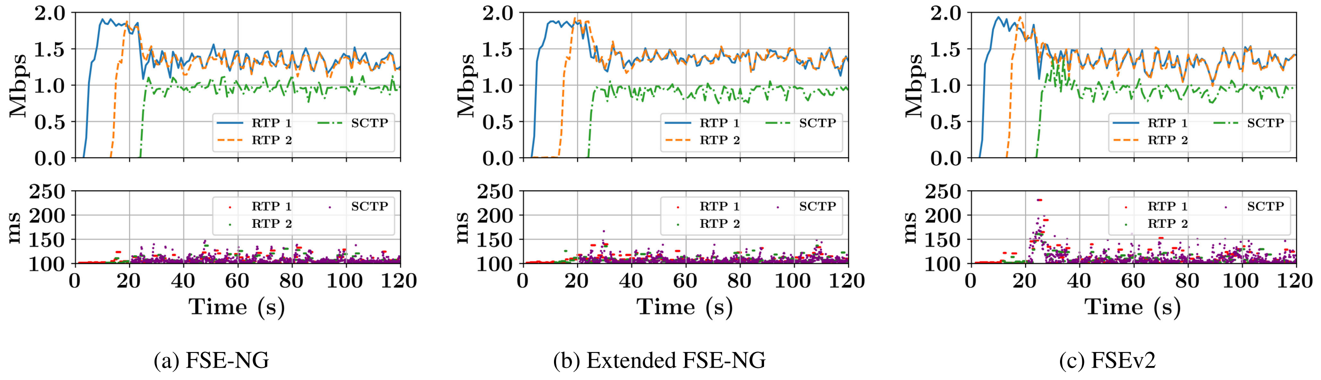


FIGURE 19. Sending rate (Mbps) and RTT (ms) for the 3 mechanisms. RTP 1 and 2 with P = 1.5, SCTP with P = 1. The bottleneck has a capacity of 4 Mbps and a 300 ms queue. The RTT is 100 ms.

in relation to SCTP. Similarly to the previous test case, it is expected that; 1) the prioritization is enforced, 2) there is still appropriate bandwidth utilization, and 3) finally, that queuing delay stays the same. Consequently, each GCC flow is expected to get 3/8 of the total utilized bandwidth while the final 2/8 parts are given to SCTP. The bottleneck capacity is 4 Mbps and one-way propagation delay is 50 ms.

Table 7 includes various metrics calculated by taking the average values based on 10 experiments. GCC flows are expected to comprise 37.5% of the total throughput while the SCTP flow should get 26% based on the priority levels. In Table 7, we can see that all three mechanisms achieve this very well. In terms of cumulative utilization, all mechanisms perform similarly.

Figure 19 plots the sending rates and delay for all 3 mechanisms. FSE-NG and extended FSE-NG show very similar behavior in terms of both metrics. The FSEv2 plot in fig. 19c exhibits oscillations between time 25s and 35s.

This behavior is caused by a conflict between the coupled flows, where SCTP is in Slow Start mode, aggressively increasing the CWND while the GCC flows decrease the rate because of the increased delay. Thus, the time before SCTP experiences loss and goes into Congestion Avoidance is prolonged. It is also apparent that GCC flows slightly under-perform while SCTP slightly over-performs during this time period, leading to the priorities not being precisely honored. This is because the media encoder is not able to change the video quality as quickly as the target rate changes. The FSE-NG based mechanisms do not experience this because they set the upper limit of the CWND and do not receive the rapid rate updates from SCTP. Avoiding this behavior with a mechanism that receives updates from SCTP, for instance, by skipping SCTP's Slow Start mode when it registers after GCC flows, would likely lead to slow convergence on higher capacity links; therefore it is a necessary trade-off.

C. ONE GCC AND ONE SCTP FLOW COUPLED, GCC HAVING DR = 1.5 MBPS

This test case aims to evaluate how well the mechanisms allow SCTP to utilize available bandwidth when there is enough capacity to satisfy RTP flows. In this test case, the GCC flow is configured to have a DR of 1.5 Mbps, and both flows are given equal priority. It is expected that GCC's throughput will converge to a stable rate of 1.5 Mbps. SCTP should be able to quickly converge to around 3.5 Mbps since the total capacity is 5 Mbps. The coupling mechanism should also ensure that delay is kept low despite SCTP sending at a higher throughput than GCC. The bottleneck capacity in this scenario is 5 Mbps and one-way propagation delay is 50 ms.

Figure. 20 shows the throughput and delay for the mechanisms when the SCTP flow is started before the GCC flow. In this case, some difference between the mechanisms is visible. Firstly, in fig. 20a, FSE-NG has two problems; 1) when the GCC flow starts, the SCTP flow's sending rate gets dragged all the way down to 1 Mbps (see Section V-C2b), 2) SCTP recovering convergence afterwards is very slow, taking approx. 20 seconds (see Section V-C2c). Our Extended

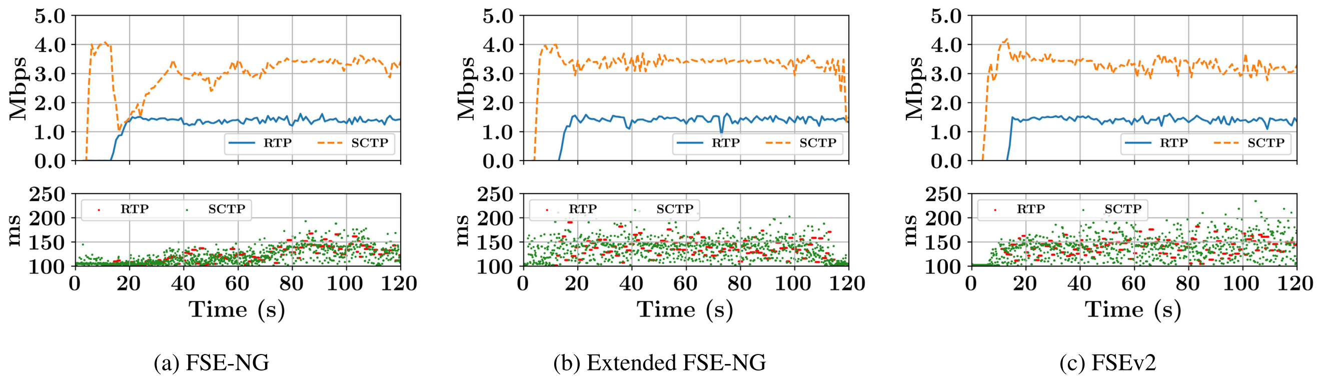


FIGURE 20. Sending rate (Mbps) and RTT (ms) of one RTP flow and one SCTP flow. The bottleneck has a capacity of 5 Mbps and a 300 ms queue. The RTT is 100 ms. DR of the RTP flow is set to 1.5 Mbps.

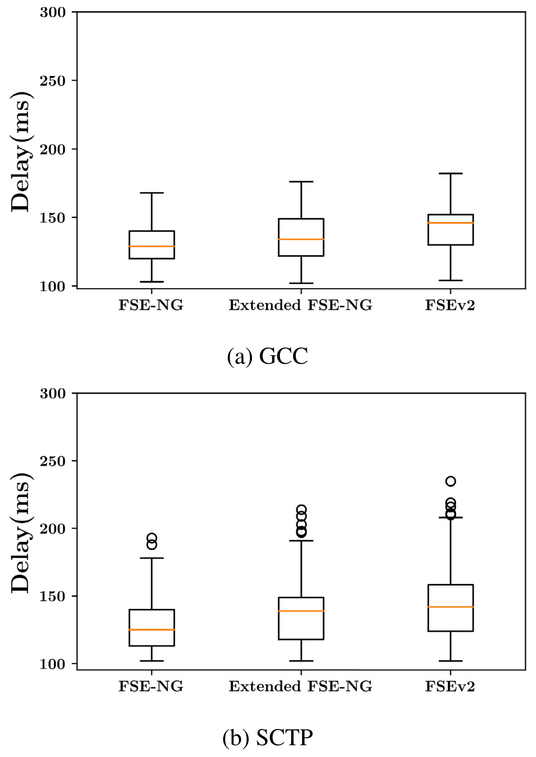


FIGURE 21. RTT measurements with the RTP flow being application limited. The bottleneck has a capacity of 5 Mbps and a 300 ms queue. The RTT is 100 ms. DR of the RTP flow is set to 1.5 Mbps. We only consider the time intervals when both flows have converged.

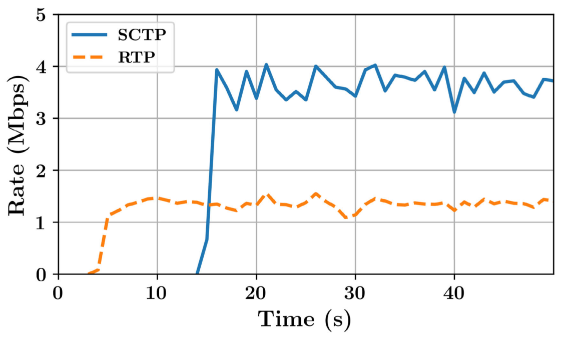


FIGURE 22. Sending rate of one RTP flow and one SCTP flow, RTP with DR set to 1.5 Mbps, and coupled with FSEv2. The bottleneck has a capacity of 6 Mbps and a 300 ms queue. The RTT is 100 ms.

FSE-NG changes mostly fix the first problem and entirely remove the second problem. However, testing shows that these fixes only matter whenever the total rate is large enough to satisfy the DR of all registered GCC flows. Thus, when this is not the case, SCTP gets dragged down if it starts first, and SCTP convergence is also slow.

FSEv2 never suffers from either of the aforementioned issues and in this case, the advantages of receiving SCTP rate updates in all cases become very clear. We also ran tests with the GCC flow starting first; in this case, FSE-NG also made SCTP convergence very slow, while Extended FSE-NG and FSEv2 allowed quick convergence. Furthermore, in case of FSEv2, we can see from fig. 20c that GCC converges faster than with the other mechanisms.

Figure. 21 shows delay box plots based on logging of GCC and SCTP RTT measurements. The GCC measurements in fig. 21a make it apparent that the mechanisms all are able to keep GCC delay reasonably low despite the parallel SCTP transmission. The SCTP RTT measurements illustrated in fig. 21b show FSE-NG keeping delay lower than the other two mechanisms. We can trace FSE-NG's lower delay back to the fact that the mechanism limits SCTP to a higher degree than the other mechanisms by not allowing it to send rate updates.

D. IMPROVING SCTP's SLOW CONVERGENCE

When the desired rate limits the RTP flow, and the link's capacity is large, the convergence time for SCTP can become very long. Both FSE-NG and our Extended FSE-NG are not able to solve this problem (see fig. 13 for FSE-NG and fig. 14 for Extended FSE-NG). Because FSEv2 takes rates from both GCC and SCTP flows, it can be seen from fig. 22 that FSEv2 fixes this problem.

IX. CONCLUSION

In this paper, we have shown how the design of using two different congestion control mechanisms within two different transport protocols in WebRTC can lead to competition between them, resulting in fairness issues and undesired spikes in queuing delay and losses. In the light of this, we have implemented two state-of-the-art congestion control coupling mechanisms in the Chromium browser: 1) the Active FSE from RFC 8699 [4] that couples media flows, and 2) FSE-NG from [5] that couples NADA and SCTP flows. Based on the derived implications and lessons learned from FSE and FSE-NG, we have implemented and evaluated two novel solutions for heterogeneous flows: 1) Extended FSE-NG and 2) FSEv2. Controlled testbed experiments have shown that our mechanisms can combine a set of heterogeneous congestion control mechanisms, fairly allocate the available bandwidth between the flows, and reduce overall delay and losses. Our experimental results confirm that our mechanisms reduce the negative impact of the data channel on the video channel.

Having implemented and tested them in Chromium, this paper has taken these congestion control coupling algorithms from theory to practice. We believe that only one final limitation must be addressed before real-life deployment in web browsers: while the introduced mechanisms cater for application-limited media flows, we have assumed that SCTP, when present, always fully utilizes its allowed cwnd. In a real implementation, a distinction between this case and the case of other limitations (SCTP running out of data, or being limited by the receiver window) must be made. Then, the implementation of an FSE variant would obtain a ''desired cwnd'' from SCTP, and the algorithm should be extended to use this value in the same way in which we have used the ''desired rate'' of GCC flows.

Our code is open source, and freely available from [28]; we believe that these implementations should serve as a good basis for code in widely-used WebRTC-capable browsers. Regarding the choice of algorithm, we recommend FSEv2. While our results have shown that, due to its heavier reliance on SCTP rate updates, this algorithm does not always consistently perform best, e.g., in terms of delay, the differences are miniscule. FSEv2 is, however, the only algorithm that fixes the SCTP convergence problem in the realistic case where GCC flows are application-limited. Also, at the time of writing this paper, the WebRTC developers began a transition to a new SCTP library, which might necessitate adapting the implementations to use the new library. This could be problematic for mechanisms such as FSE-NG since they are dependent on the SCTP library supporting a maximum CWND limit option, which the new library seemingly does not, for the time being.

The present work has only focused on coupling flows running between two peers. One possible avenue for further research is to explore a scenario with several peers, e.g., conference call applications where all flows sent from a given peer or several peers to one or more destinations may share the same bottleneck. As future work, we plan to investigate such scenarios using a shared bottleneck detection method [30], [31] to infer which flows share a common path. Such an extension could greatly amplify the benefits attained with these coupling mechanisms, since they would then operate on a much larger number of flows.

REFERENCES

[1] S. Islam, M. Welzl, S. Gjessing, and N. Khademi, ''Coupled congestion control for RTP media,'' *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 1–10, Aug. 2014, doi: 10.1145/2740070.2630089.

[2] S. Islam, M. Welzl, D. Hayes, and S. Gjessing, ''Managing real-time media flows through a flow state exchange,'' in *Proc. IEEE/IFIP Netw. Oper. Manag. Symp. (NOMS)*, Apr. 2016, pp. 112–120.

[3] J. Flohr and E. P. Rathgeb, ''ROSIEE: Reduction of self inflicted queuing delay in webRTC,'' in *Proc. 29th Int. Teletraffic Congr. (ITC)*, vol. 3, Sep. 2017, pp. 7–12.

[4] S. Islam, M. Welzl, and S. Gjessing, *Coupled Congestion Control for RTP Media*, document RFC 8699, Jan. 2020. [Online]. Available: https://rfc-editor.org/rfc/rfc8699.txt

[5] J. Flohr, E. Volodina, and E. P. Rathgeb, ''FSE-NG for managing real time media flows and SCTP data channel in webRTC,'' in *Proc. IEEE 43rd Conf. Local Comput. Netw. (LCN)*, May 2018, pp. 315–318.

[6] S. Holmer, H. Lundin, G. Carlucci, L. D. Cicco, and S. Mascolo, ''A Google congestion control algorithm for real-time communication,'' Internet-Draft draft-ietf-rmcat-gcc-02, Internet Eng. Task Force, Fremont, CA, USA, Tech. Rep., Jul. 2016. [Online]. Available: Available: https://datatracker.ietf.org/doc/html/draft-ietf-rmcat-gcc-02

[7] Google WebRTC Team. (2021). *Real-Time Communication for the Web*. [Online]. Available: https://webrtc.org/

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

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

[10] *User Datagram Protocol*, document RFC 768, Aug. 1980. [Online]. Available: https://rfc-editor.org/rfc/rfc768.txt

[11] E. Carrara, K. Norrman, D. McGrew, M. Naslund, and M. Baugher, *The Secure Real-Time Transport Protocol (SRTP)*, document RFC 3711, Mar. 2004. [Online]. Available: https://www.rfc-editor.org/info/rfc3711

[12] E. Rescorla and N. Modadugu, *Datagram Transport Layer Security Version 1.2*, document RFC 6347, Jan. 2012. [Online]. Available: https://www.rfc-editor.org/info/rfc6347

[13] E. Blanton, D. V. Paxson, and M. Allman, *TCP Congestion Control*, document RFC 5681, Sep. 2009. [Online]. Available: https://www.rfc-editor.org/info/rfc5681

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

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

[16] H. T. Alvestrand, ''RTCP message for receiver estimated maximum bitrate,'' Internet-Draft draft-alvestrandrmcat-remb-03, Internet Eng. Task Force, Fremont, CA, USA, Tech. Rep., Oct. 2013. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-alvestrand-rmcat-remb-03

[17] H. Balakrishnan and S. Seshan, *The Congestion Manager*, document RFC 3124, Jun. 2001. [Online]. Available: https://rfc-editor.org/rfc/rfc3124.txt

[18] S. Islam and M. Welzl, ''Start me up: Determining and sharing TCP's initial congestion window,'' in *Proc. Appl. Netw. Res. Workshop (ANRW)*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 52–54, doi: 10.1145/2959424.2959440.

[19] S. Islam, M. Welzl, K. Hiorth, D. Hayes, G. Armitage, and S. Gjessing, ''CtrlTCP: Reducing latency through coupled, heterogeneous multi-flow TCP congestion control,'' in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, May 2018, pp. 214–219.

[20] S. Islam, M. Welzl, S. Gjessing, and J. You, ''OpenTCP: Combining congestion controls of parallel TCP connections,'' in *Proc. IEEE Adv. Inf. Manag., Communicates, Electron. Autom. Control Conf. (IMCEC)*, Aug. 2016, pp. 194–198.

[21] L. Eggert, J. Heidemann, and J. Touch, "Effects of Ensemble-TCP," *SIGCOMM Comput. Commun. Rev.*, vol. 30, no. 1, pp. 15–29, Jan. 2000, doi: 10.1145/505688.505691.
[22] M. Savoric, H. Karl, M. Schläger, T. Poschwatta, and A. Wolisz, "Analysis and performance evaluation of the EFCM common congestion controller for TCP connections," *Comput. Netw.*, vol. 49, no. 2, pp. 269–294, 2005.
[23] P. Wilkins, Y. Xu, L. Quillio, J. Bankoski, J. Salonen, and J. Koleszar, *VP8 Data Format and Decoding Guide*, document RFC 6386, Nov. 2011. [Online]. Available: https://www.rfc-editor.org/info/rfc6386
[24] B. Jansen, T. Goodwin, V. Gupta, F. Kuipers, and G. Zussman, "Performance evaluation of webRTC-based video conferencing," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 45, pp. 56–68, Mar. 2018.
[25] G. Carlucci, L. De Cicco, S. Holmer, and S. Mascolo, "Analysis and design of the Google congestion control for web real-time communication (WebRTC)," in *Proc. 7th Int. Conf. Multimedia Syst. (MMSys)*. New York, NY, USA: Association for Computing Machinery, Feb. 2016, pp. 1–5, doi: 10.1145/2910017.2910605.
[26] Z. Sarker, V. Singh, X. Zhu, and M. A. Ramalho, *Test Cases for Evaluating Congestion Control for Interactive Real-Time Media*, document RFC 8867, Jan. 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc8867
[27] (2022). *OMNeT++ Discrete Event Simulator*. [Online]. Available: https://omnetpp.org/
[28] (2022). *Source Code—Coupled Congestion Control Mechanisms for Data and Video Flows*. [Online]. Available: https://github.com/tobiasfl/tobias-master-thesis-webrtc
[29] R. Jain, D. Chiu, and W. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer systems," *CoRR*, vol. cs.NI/9809099, Jan. 1998.
[30] D. Hayes, S. Ferlin, M. Welzl, and K. Hiorth, *Shared Bottleneck Detection for Coupled Congestion Control for RTP Media*, document RFC 8382, Jun. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8382
[31] D. A. Hayes, M. Welzl, S. Ferlin, D. Ros, and S. Islam, "Online identification of groups of flows sharing a network bottleneck," *IEEE/ACM Trans. Netw.*, vol. 28, no. 5, pp. 2229–2242, Apr. 2020.

SAFIQUL ISLAM (Member, IEEE) received the Ph.D. degree in computer science from the University of Oslo, Norway. He is currently an Associate Professor at the University of South-Eastern Norway. He is also an Adjunct Associate Professor at the University of Oslo. He actively participated in two EU projects (NEAT and RITE) and was a technical lead of an international project ("TCP-in- UDP") with Huawei, China. He is active in the IETF and IRTF, where he has contributed to several IETF/IRTF Working Groups. His research interests include performance analysis, evaluation, and optimization of transport layer protocols.

MICHAEL WELZL (Member, IEEE) received the Ph.D. and Habilitation degrees from the University of Darmstadt, Germany, in 2002 and 2007, respectively. He has been a Full Professor at the University of Oslo, Norway, since 2009. He has been active in the IETF and IRTF for many years, such as by chairing the Internet Congestion Control Research Group (ICCRG) leading the effort to form the Transport Services (TAPS) Working Group. He has also participated in several European research projects, including roles such as a co-ordinator and a technical manager. His main research interest includes transport layer.

TOBIAS FLADBY received the M.Sc. degree in computer science from the University of Oslo, Norway. He is currently a Software Engineer at Cisco, Norway. His research interests include performance analysis of transport protocols and WebRTC.