GSO-Simulcast: Global Stream Orchestration in Simulcast Video Conferencing Systems

Xianshang Lin*, Yunfei Ma*†, Junshao Zhang*, Yao Cui, Jing Li, Shi Bai, Ziyue Zhang,
Dennis Cai, Hongqiang Harry Liu, Ming Zhang
Alibaba Group
* co-first authors

ABSTRACT

We present GSO-Simulcast, a new architecture designed for large-scale multi-party video-conferencing systems. GSO-Simulcast is currently deployed at full-scale in Alibaba's Dingtalk video conferencing that serves more than 500 million users. It marks a fundamental shift from today's Simulcast, where a media server locally decides how to switch and forward video streams based on a fragmented network view. Instead, GSO-Simulcast globally orchestrates the publishing, subscribing, as well as the resolution and bitrate of video streams for each participant using a centralized controller that is aware of all network constraints in a meeting. The controller automatically modifies stream configurations to meet the participants' real-time network changes and updates. In doing so, GSO-Simulcast achieves multiple goals: (1) reducing video and network mismatch, (2) less path congestion, and (3) automated stream policy management. With the deployment of GSO-Simulcast, we observed more than a 35% reduction in the average video stall, 50% reduction in the average voice stall, and 6% improvement in the average video framerate. We describe the principle, design, deployment, and lessons learned.

CCS CONCEPTS

• **Networks** → **Application layer protocols**; **Public Internet**.

KEYWORDS

Video Conferencing, Teleconferencing, Simulcast, Mobile Transport

ACM Reference Format:
Xianshang Lin*, Yunfei Ma*†, Junshao Zhang*, Yao Cui, Jing Li, Shi Bai, Ziyue Zhang,, Dennis Cai, Hongqiang Harry Liu, Ming Zhang. 2022. GSO-Simulcast: Global Stream Orchestration in Simulcast Video Conferencing Systems. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3544216.3544228

†For questions and comments, please contact yunfei.ma@alibaba-inc.com.

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08…$15.00
https://doi.org/10.1145/3544216.3544228

1 INTRODUCTION

Real-time voice/video communication has fundamentally reshaped our society. With the Covid variants continuing to surge, video-conferencing emerges as the beating heart of remote collaboration, learning, and personal interaction for billions of people around the globe. These ever-growing needs of virtual connectivity continue to drive the flourish of a vast number of video-conferencing services such as Zoom [1], Microsoft Teams [2], Google Meet [3], Amazon Chime [4], and Alibaba Dingtalk [5].

As one of the world's major teleconferencing and collaboration apps, Dingtalk serves more than 500 million users. It is mission-critical for us to ensure satisfactory user experience in video conferencing. In this paper, we present GSO-Simulcast, a new architecture designed for large-scale multi-party video conferencing. The introduction of GSO-Simulcast is by reason of the following trends we face:

- *Better video quality.* In order to create a more interactive and engaging conversation, there is a growing need for users to see higher video qualities at low latency. Thus, we want our conferencing system to support higher average bitrate and framerate even without the upgrade of underlay infrastructure.
- *Bigger conference.* We notice that the average meeting size is increasing with more and more events going virtual. Today, it is not uncommon for our users to host a meeting with hundreds of participants. Therefore, ensuring a good user experience in large conferences becomes a focal point.
- *Improved slow-link performance.* We note that the meeting fluency is heavily impacted by the participant who has the worst network conditions (slow links). Moreover, as meeting size grows, the likelihood of someone in the room having a slow link increases, so it is important for us to improve the media transport under slow links.
- *More accessibility.* We want to make our service even more accessible so that users can access our technology from anywhere with any device. To do so, we need a cost-effective solution. On top of that, such a solution must support a wide range of desktop and mobile devices.

Serving high-quality real-time media streams in a large-scale multi-party video conference is particularly challenging due to (i) the heterogeneous network conditions of different participants, and (ii) the complicated subscribing relations between each pair of participants [1]. For example, when a publisher sends a high-quality video stream to a subscriber with insufficient bandwidth, high packet losses will occur, causing video/voice stalls and delays.

[1]The number of pairs increases quadratically with the number of participants

At a fundamental level, these challenges translate to the essential need for a conferencing architecture that swiftly adapts video stream qualities to each participant's network constraints. With years of evolution, the industry has converged on three main approaches to tackle this problem: transcoding [6–8], SVC [9, 10], and Simulcast [11–13]. All these approaches have pros and cons. Transcoding can near-optimally fit video streams into downstream network paths, but in doing so, it places a significant burden on the server, which needs to generate additional alternate streams on its own in real-time. As a result, transcoding is not cost-effective for us. SVC allows a single video stream to adapt to multiple bitrates. However, the problem with SVC is codec compatibility since it requires every participant in a video conference to support scalable encoding/decoding. We find that SVC is difficult to deploy in our scenarios because we need to cover a broad range of mobile devices, many of which use hardware H264 codecs that do not support scalability. In Simulcast [12, 14], a client encodes a video source multiple times in different bitrates and sends these video streams in parallel to an selective-forwarding unit (SFU), who then decides how to switch and forward these streams. Compared to transcoding and SVC, Simulcast is more scalable and cost-effective since it requires neither encoding/decoding media on a server, nor specialized functions in video codecs.

Unfortunately, state-of-the-art Simulcasts adapt video streams with stream policies (e.g., Twilio [15], Amazon Chime [16], and Chromium [17]) that have a number of limitations: (1) the uplink policy and downlink policy are isolated, where a publisher decides what to push based on his/her local view of the upstream network and the video resolution captured [15–17]. There is no coordination between uplink and downlink, and among different participants. (2) The stream policies are template-based and are tuned based on empirical observations, and hence can only cover cases of a small number of participants (typically smaller than 6) [16]. (3) They support only few coarse-grained bitrate levels (typically 2-3 levels [2]) [16] and the target bitrate ratio between two adjacent streams could be as large as 5 [17]. As a result, they suffer from several drawbacks, including but not limited to (1) mismatch between videos and networks, (2) susceptibility to uplink congestion, (3) poor manageability when conference size becomes large.

The fundamental flaw of today's Simulcast lies in the fact there is a lack of coordination among senders, receivers, and media servers. *As a result, video adaptation in Simulcast is limited to a fragmented view of the conference.* Instead, we should consider every path from the upstream to the downstream, including the network constraints in the uplinks and downlinks, and the subscriptions between each pair of participants. Therefore, we decided to build GSO-Simulcast, which marked a fundamental shift in the design philosophy. GSO-Simulcast globally orchestrates the publishing, subscribing, as well as the resolution and bitrate of video streams for each participant using a centralized controller that is aware of the full network, subscription, and codec capability constraints in a meeting. In doing so, GSO-Simulcast achieves the following unique characteristics:

- **Less path congestion**. GSO-Simulcast's controller has a global view of the conference, so it can intelligently decide the most suitable video streams for each participant to publish or subscribe to, avoiding network congestion. For example, if no receiver wants to subscribe to a stream at a specific bitrate, the controller will inform the publisher to stop pushing that stream, which not only saves bandwidth and CPU costs, but also reduces the likelihood of video/voice stalls.
- **Better manageability with automated stream policy**. Today's Simulcast is driven by template-based policies that consist of adaptation rules for each network condition and participant number [12]. Not only is tuning such rules time-consuming, but when the conference size becomes larger, or the network conditions become more complex, enumerating all possible situations becomes unmanageable. GSO-Simulcast solves the manageability problem as a global optimization, which automatically adapts to different networks and participants' situations. Devices' codec interoperabilities are also considered explicitly in the optimization.
- **Reducing video and network mismatch**. GSO-Simulcast supports video streams at a much finer bitrate granularity, thanks to its ability to precisely optimize and orchestrate streams at a global level. In our deployment, we support up to 15 bitrate levels. Such a fine bitrate granularity enables us to more efficiently fit video streams into network bandwidths, leading to better video quality. Moreover, fine-grained bitrate levels also permit smoother quality transitions as the network degrades.

Challenges: In our efforts to deploy GSO-Simulcast, we need to address a number of challenges:

- *Control in real-time.* The stream orchestration problem is combinatorial in nature. Specifically, the centralized controller needs to determine the combination of streams for each participant to publish and subscribe to. The result must simultaneously satisfy the network bandwidth constraints, the codec capability constraints, and the subscription constraints. The complexity of a brute-force searching grows exponentially with the meeting size and number of bitrate options. Hence, in order to enable GSO-Simulcast for large meetings, we must first find a way to solve the global orchestraion problem in real-time.
- *Managing priority.* The situations are further complicated in a real meeting because different streams can have different priorities. For example, a speaker's video or a screen share is usually more important than others. Dropping such streams may significantly degrade the client's experience. Hence, we need to consider stream priorities and carefully manage them in a meeting.
- *Capturing the global picture.* GSO-Simulcast relies on the global view of a meeting to make decisions. Hence, we must collect three things: codec capabilities, subscription relations, and network bandwidths. In particular, the network constraints vary frequently and are usually more precisely measured at the sender-side. Therefore, we must find a way to timely collect these disaggregated pieces of information in a meeting.
- *Compatibility with current architecture.* Today's video conferencing has a complex technology stack that uses many other network protocols, such as ICE [18], SDP [19], STUN [20], TURN [21], and RTP/RTCP [22]. Hence, innovations are usually not accessible

[2]For example, Amazon Chime employs a stream policy that sets the 360P stream to 600Kbps (if uplink bandwidth >300Kbps) or not-used (if uplink bandwidth < 300Kbps) when the number of participants is smaller than 6 [16].

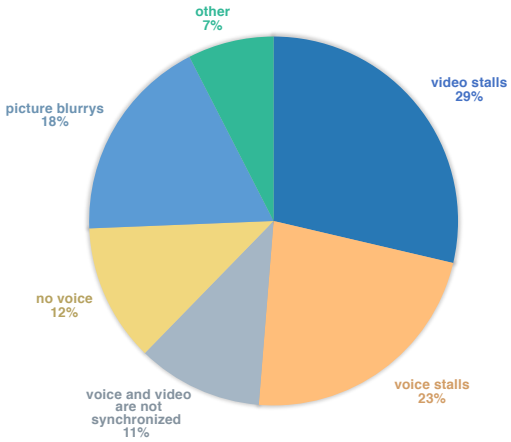


Figure 1: Pie chart of user reported issues in our video conferencing service.

due to ossified technology stacks. We must figure out a way to make GSO-Simulcast compatible with current architecture so that it can be incrementally deployed.

This paper shows how we address each of the above challenges to make GSO-Simulcast widely deployable with a layered architecture that splits a conference into the user plane, the media plane, and the control plane. We introduce the GSO controller that acts as the "brain" of a conference. We outline the key design components including the algorithm, information collection, and feedback execution that make GSO-Simulcast deployable in production.

Main results: GSO-Simulcast is currently deployed at full-scale in Dingtalk video conferencing services, serving more than 500 million users. Based on the statistics from 1 million video conferences per day over 3 months, we observed significant improvements in key metrics. The average video stall and voice stall are reduced by more than 35% and 50%, respectively. The video framerate is improved by 6%. These improvements show the value of the centralized stream orchestration approach in large-scale video conferencing for the first time.

Contributions: We make the following contributions:

- To the best of our knowledge, GSO-Simulcast is the first widely deployed video conferencing system that globally orchestrates Simulcast flows.
- GSO-Simulcast significantly improves all key QoE metrics in large-scale deployment. We believe its success has provided a new path for the future evolution of video conferencing architectures.
- We discuss the design choices, present key principles and techniques, and share our experience in deploying GSO-Simulcast.

Claim: *This work does not raise any ethical issues.*

2 BACKGROUND

We start by providing the background to help readers better understand the scope of this paper.

2.1 User reported issues

The fast growth of our video conferencing business introduces new challenges. To help us better identify areas to improve, we collected user experience issues. The pie chart of the reported issues is shown in Fig. 1. The top three reported issues are video stalls (29%), voice stalls (23%), and blurred videos (18%), all of which are root-caused in the slow-link problem.

2.2 Slow-link problem

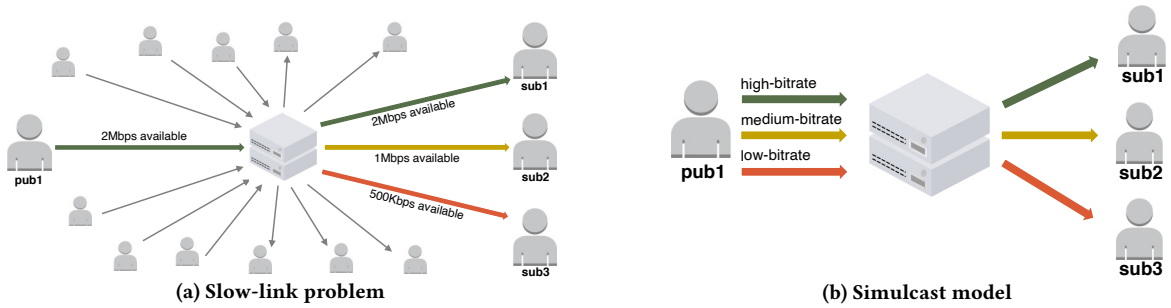
One of the biggest challenges in multi-party video conferencing is the heterogeneous network conditions facing different participants, which give rise to the slow-link problem, as shown in Fig. 2a. For simplicity, let's focus on one publisher (pub1) with an uplink bandwidth at $2Mbps$ and three subscribers (*sub1*, *sub2*, and *sub3*) with downlink bandwidth at $2Mbps$, $1Mbps$, and $500Kbps$, respectively. A natural question arises: What stream bitrate should *pub1* send? Pushing video a stream at $2Mbps$ serves *sub1* best, but in doing so, it will cause video stalls at *sub*2 and *sub*3, as their network cannot sustain such a high bitrate. As a result, the *pub*1 has no choice but to push a video stream smaller than $500kbps$, which inevitably hurts *sub1*'s and *sub2*'s user experiences. In other words, in a multi-party video conference, the subscriber with the slowest link will determine the video quality received by a group of subscribers. As the conference size grows bigger, the likelihood of someone who has a slow link increases. Addressing the slow link problem is crucial in delivering any satisfactory user experience.

2.3 Why GSO-Simulcast?

Simulcast was introduced to alleviate the slow-link problem discussed above, as illustrated in Fig. 2b. In Simulcast, a client encodes a video source multiple times in different bitrates and sends those streams in parallel to an SFU server. The SFU server selects one of the streams to forward for each receiving participant based on the receiver's network constraint. In Fig. 2b, to accommodate the bandwidth constraint for each receiver, the SFU forwards the high-bitrate stream to *sub1*, the medium-bitrate stream to *sub2*, and the low-bitrate stream to *sub3*, respectively. Regarding the implementation, one can send multiple RTP streams in a single RTP session as defined in RFC 8861 [23], so enabling Simulcast with an SFU server is relatively easy. On top of that, parallel streams used in Simulcast can be easily generated with standard codecs at user-ends. Hence, Simulcast retains scalable and cost-effective properties.

However, Simulcast encounters new problems that lead to sub-optimal performance, owing to the lack of coordination among senders, receivers, and media servers. To better understand these issues, we explain through a couple of examples as shown in Fig. 3a, 3b, and 3c. It is worth noting that the actual scenarios are usually more complicated. In Fig. 3d, 3e, and 3f, we show how the situations in those examples can be improved, respectively, when we have the capability to globally control the conference with a GSO controller that is aware of the full network information.

- *Example1: Susceptibility to network congestion.* In the upstream, sending multiple streams consumes more bandwidth. In many circumstances, the valuable network resource is simply wasted when a sender pushes a stream that no one subscribes to. In the example shown in Fig. 3a, *pub*1 pushes three streams to the SFU, while the SFU decides to only use a 300Kbps stream to *sub*1, and a 600Kbps stream to *sub*2. As *pub*1 is unaware of the SFU's decision, it continues to send a 1.5Mbps stream. As a result, a



(a) Slow-link problem (b) Simulcast model

Figure 2: (a) The slow-link problem in multi-party video conferencing, and (b) The Simulcast model.

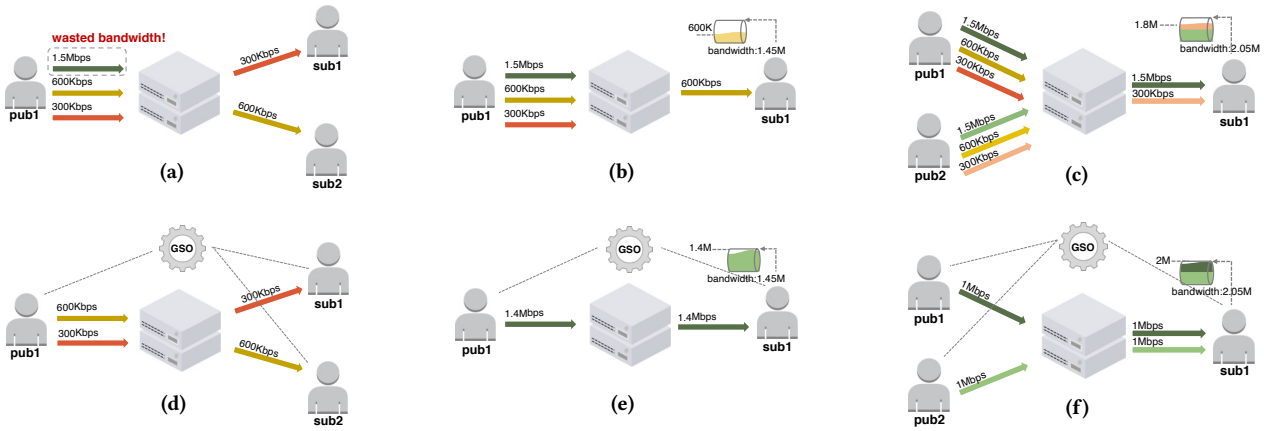


Figure 3: Examples of today's Simulcast's problems:(a) - (c), and how situations can be improved with global stream orchestration (GSO) in: (d)-(f), respectively.

major proportion of *pub*1's uplink bandwidth is wasted and the network is more likely to congest.

Improvement: The above situation can be improved with GSO, as shown in Fig. 3d. In the same situation as above, a GSO controller will notify *pub*1 to stop sending the 1.5Mbps stream as it is not needed, which ends up not only saving bandwidth but also reducing the possibility of network congestion.

- *Example2: video and network mismatch.* When a sender is not aware of the network constraints of a receiver, it cannot decide the right bitrate. As shown in Fig. 3b, *sub*1's downlink bandwidth is 1.45Mbps, 50Kbps below the large stream's bitrate (1.5Mbps) that *pub*1 sends. As a result, the SFU has to downgrade the video bitrate to 600Kbps, which is significantly lower than what *sub*1 is capable of receiving, causing video and network mismatch and leading to poor video quality for *sub*1.

  Improvement: The above situation can be improved with GSO, as shown in Fig. 3e. In the same situation as above, a GSO controller will inform *pub*1 the exact bandwidth of *sub*1, so *pub*1 can tune the video stream at a fine granularity and fits the network by sending a video stream at 1.4Mpbs to provide better video quality for *sub*1.
- *Example3: stream competition.* When a receiver's downlink bandwidth is limited, streams from different senders are forced to compete with each other. As shown in Fig. 3c, *sub*1's downlink bandwidth is limited to 2.05Mbps, so if the SFU selects a large stream at 1.5Mbps from *pub*1, the room left can only fit a small stream at 300Kbps from *pub*2, leading to uneven visual effects.

  Improvement: The above situation can be improved with GSO, as shown in Fig. 3f. In the same situation as above, a GSO controller will instruct both *pub*1 and *pub*2 to send a stream at 1Mbps to fairly share the bandwidth, providing a more uniform visual experience.

3 GSO-SIMULCAST OVERVIEW

In the above discussion, we explain why we need GSO in Simulcast. In this section, we provide an overview of GSO-Simulcast. As shown in Fig. 4, at a high level, GSO-Simulcast separates a meeting into three planes: the user plane, the media plane, and the control plane. The user plane is formed by clients who play the role of publishers and subscribers. Above the user plane sits the media plane, which consists of a group of accessing nodes that are interconnected. The accessing node is responsible for providing media access to clients and routing media data based on instructions from the control plane. When a client in the user plane publishes a media stream, the media packets are received by one of the accessing nodes, which then forwards the packets directly to receiving clients in the same region, or to other accessing nodes to further relay the data to receiving clients in different regions.

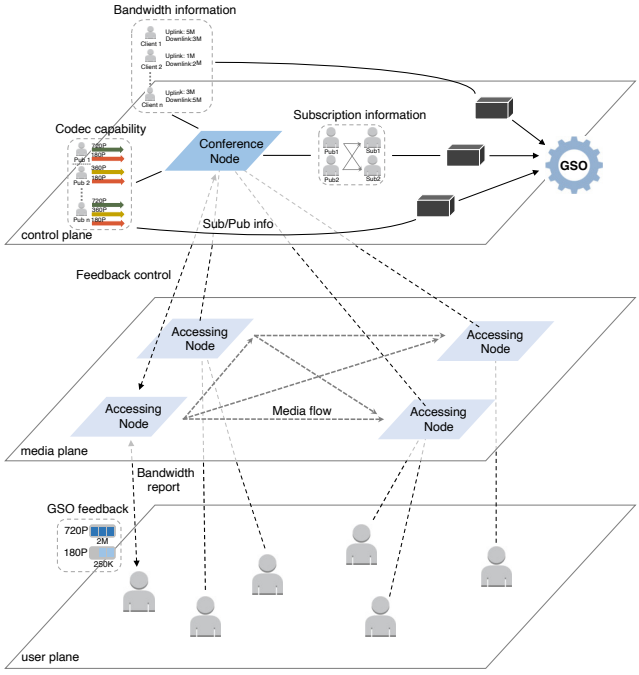


Figure 4: The overview architecture of GSO-Simulcast.

On top of the media plane is the control plane that comprises the conference node and the GSO controller. The conference node has two main functions: (1) it handles the signaling with clients and accessing nodes, and (2) it captures the global picture of a conference, which is used as inputs to the GSO controller. The information captured by the conference node includes network bandwidth constraints, subscriptions among clients, clients' codec capabilities, and other meeting-specific data such as who is the current speaker and who shares the screen. The GSO controller is the "brain" of the conference. It performs two jobs: (1) it synthesizes the inputs from the conference node to generate vectors of fine-grained stream bitrates that each client is able to send or receive, as well as the QoE-utility vectors, which contain the QoE utility weights for each stream bitrate, and (2) it solves a global optimization problem given the subscription constraints, the codec capability constraints, and the network constraints. This optimization aims to satisfy each subscriber's needs with maximized QoE utility, and the outputs determine the stream resolution and bitrate configurations for each client in the user plane.

4 DESIGN

4.1 Control Algorithm

We first introduce the core control algorithm used in GSO-Simulcast. The challenge is how to map our real-life situations into a mathematical formulation that serves user QoEs, involves practical constraints, and at the same time, can be computed in real-time. At a high level, the algorithm is executed in iterative loops of a three-step (Knapsack-Merge-Reduction) operation, with each step handling a particular set of constraints.

Goal: The goal of the control algorithm is to satisfy each subscriber's needs with maximized QoE utility while complying with three sets of constraints: the network bandwidth constraints, the codec capability constraints, and the subscription constraints.

Network bandwidth constraints: Let $\mathcal{I} = \{1, ..., |\mathcal{I}|\}$ be a finite set that represents the labels of $|\mathcal{I}|$ clients. For each client $i \in \mathcal{I}$, B_i^u and B_i^d denote the uplink bandwidth constraint and downlink bandwidth constraint of the i-th client, respectively. Notice that each client can play the role of subscriber and/or the publisher at a given time, thus for the i-th client, the sum of the subscribed stream bitrates must not exceed B_i^d, while the sum of the published stream bitrates must not exceed B_i^u.

Codec capability constraints: Recall that each client $i \in \mathcal{I}$ may play the role of publisher and/or subscriber. Let $\mathcal{S}_i = \{s_i : 1 \le i \le |\mathcal{S}_i|\}$ be the set of feasible stream bitrates associated with the i-th client when he/she plays the role of publisher. Moreover, assume that each bitrate s_i is associated with a unique resolution and a unique QoE utility weight, let $\mathcal{R}$ and $\mathcal{Q}$ be the set of resolutions and QoE utility weights, respectively. Two functions are introduced to define these two associations, $Res_i : \mathcal{S}_i \mapsto \mathcal{R}, \quad i \in \mathcal{I}$, and $QoE_i : \mathcal{S}_i \mapsto \mathcal{Q}, \quad i \in \mathcal{I}$. As one can imagine, for each $s_i \in \mathcal{S}_i$, $Res_i(s_i)$ and $QoE_i(s_i)$ indicate respectively the resolution and QoE utility weight associated with the bitrate s_i. In GSO-Simulcast, a publisher can send multiple streams at different resolutions in parallel. For each resolution, a stream can be sent at different fine-grained bitrates, but no more than one stream is sent at a time. More specifically, for the i-th publisher, the streams he/she sends is some subset $\mathcal{S}_i'$ of $\mathcal{S}_i$ such that, $Res_i(s_{i_1}) \neq Res_i(s_{i_2}), \quad \forall s_{i_1}, s_{i_2} \in \mathcal{S}_i'$.

Subscription constraints: We assume that each client i may subscribe and be subscribed to by sets of other clients excluding himself/herself. Thus let $\mathcal{N}_i$ be the set of clients that i intends to subscribe to and $\mathcal{M}_i$ the set of clients that i is asked to serve, we must have $\mathcal{N}_i \subseteq \mathcal{I} \setminus \{i\}$, and $\mathcal{M}_i \subseteq \mathcal{I} \setminus \{i\}$. Now, consider a pair of clients $i, i' \in \mathcal{I}$ such that $i \in \mathcal{N}_{i'}$ and $i' \in \mathcal{M}_i$, or equivalently, i is a publisher that is subscribed to by i'. Usually, i' will indicate the maximum resolution, denoted as $R_{ii'}$, that he/she is willing to accept from i. Thus, the feasible bitrate set under the subscription relation between publisher i and subscriber i' can be denoted as a subset $\mathcal{S}_{ii'}$ of $\mathcal{S}_i$ as $\mathcal{S}_{ii'} = \{s : s \in \mathcal{S}_i \ \wedge Res_i(s) \le R_{ii'}\}$. In GSO-Simulcast, a subscriber i' is allowed to subscribe to no more than one stream from each publisher $i \in \mathcal{N}_{i'}$[3]

The optimization problem described above is combinatorial in nature, and hence, the time-complexity of brute-force searching grows exponentially with the number of participants and the bitrate options. Therefore, we need to find a way to transform this problem into an appropriate approximation to solve it in real-time. Below, we discuss how GSO-Simulcast solves this problem through iterations in three steps (Knapsack, Merge, and Reduction), decomposed so that each step handles a particular set of constraints and is also efficient to compute. Moreover, to help readers better understand the procedure, Fig. 5 visualizes the algorithm with three clients A, B, and C.

4.1.1 Step1 (Knapsack): addressing downlink and subscription constraints. We first make an attempt to address the downlink network

[3]Later, we discuss how subscribers can subscribe to more than one stream from a publisher in 4.4.

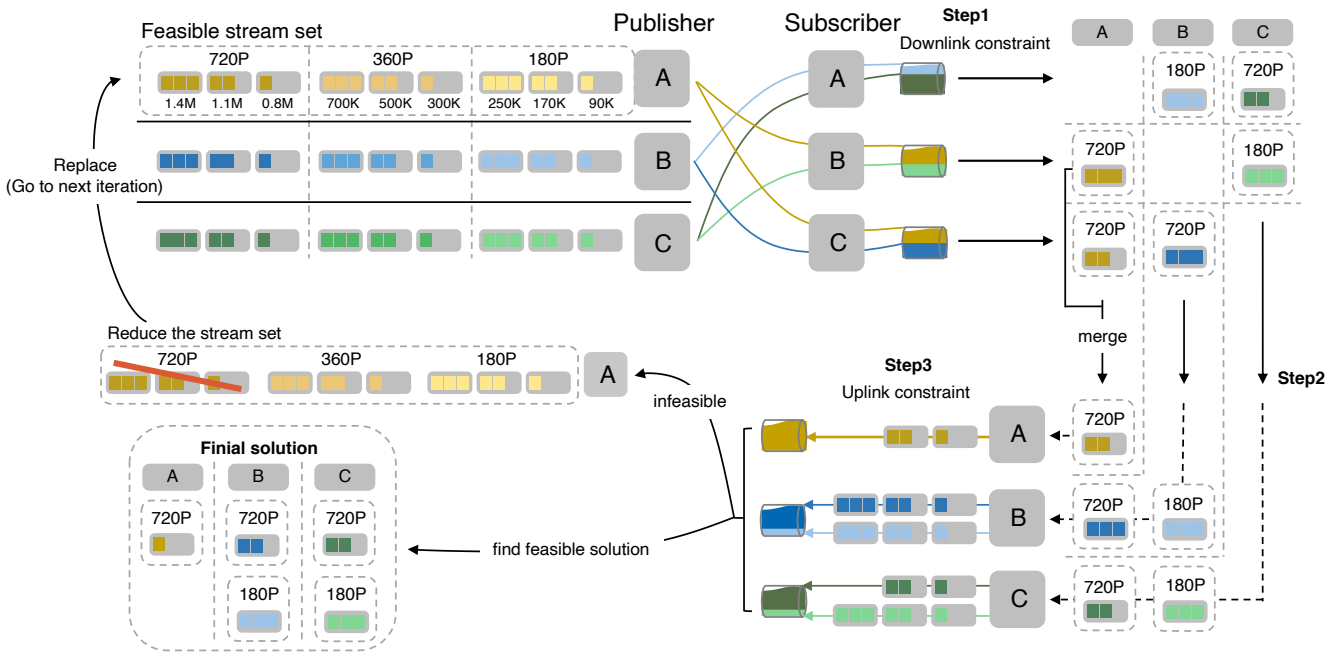


Figure 5: A brief illustration of the control algorithm with three clients A, B, and C. Each client plays the role of both subscriber and publisher. For simplicity, each client sends streams in three different resolutions (720p, 360p, and 180p), with each resolution having three bitrate options. The algorithm is executed in a loop of three-step (Knapsack-Merge-Reduction) operations.

constraints and the subscription constraints discussed above [4]. It turns out that for each subscriber $i' \in \mathcal{I}$, what we need is to fill the downlink bandwidth by choosing at most one stream from $\mathcal{S}_{ii'}, \forall i \in \mathcal{N}_{i'}$, such that the QoE utility of the subscriber i' is maximized. To make notations cleaner, the elements in $\mathcal{S}_{ii'}$ is relabeled as $\mathcal{S}_{ii'} = \{s_{ik} : 1 \le k \le |\mathcal{S}_{ii'}|\}$. Then we have,

$$\max \sum_{i \in \mathcal{N}_{i'}} \sum_{k=1}^{|\mathcal{S}_{ii'}|} QoE_i(s_{ik}) x_{ik}, \ i' \in \mathcal{I} \tag{1}$$

subject to:

$$\sum_{i \in \mathcal{N}_{i'}} \sum_{k=1}^{|\mathcal{S}_{ii'}|} s_{ik} x_{ik} \le B^d_{i'}, \ i' \in \mathcal{I} \tag{2}$$

$$x_{ik} \in \{0, 1\}, \ 1 \le k \le |\mathcal{S}_{ii'}|, \ i \in \mathcal{N}_{i'} \tag{3}$$

$$\sum_{k=1}^{|\mathcal{S}_{ii'}|} x_{ik} \le 1, \ i \in \mathcal{N}_{i'} \tag{4}$$

Given a particular $i' \in \mathcal{I}$, let $\hat{x}_{ik}, \forall i \in \mathcal{N}_{i'}, \forall k \in [1, |\mathcal{S}_{ii'}|]$ be the optimal solution of (1). Then for each $i \in \mathcal{N}_{i'}$, the corresponding stream bitrate in $\mathcal{S}_{ii'}$ selected by $\hat{x}_{ik}$ can be denoted as,

$$\hat{s}_{ii'} = \sum_{k=1}^{|\mathcal{S}_{ii'}|} \hat{x}_{ik} s_{ik}, \quad i \in \mathcal{N}_{i'}, \ i' \in \mathcal{I} \tag{5}$$

[4] In a switched video conference, participants tend to receive more streams than they send. The downlink bandwidth limits are likely to be hit before uplink in a large conference. Therefore once the downlink constraints are met, small changes to the solutions are expected to meet the uplink constraints.

together with (3) and (4), we must have that either $\hat{s}_{ii'} \in \mathcal{S}_{ii'}$ or $\hat{s}_{ii'} = 0$. Here for each client $i' \in \mathcal{I}$, we introduce a set $\mathcal{D}_{i'}$ to record the corresponding result $\hat{s}_{ii'}$ as follow,

$$\mathcal{D}_{i'} = \{(i, \hat{s}_{ii'}) : i \in \mathcal{N}_{i'} \wedge \hat{s}_{ii'} \text{ via (5)} \wedge \hat{s}_{ii'} \ne 0\}, \ i' \in \mathcal{I} \tag{6}$$

notice that an ordered pair $(i, \hat{s}_{ii'})$, instead of a pure stream $\hat{s}_{ii'}$, is introduced to properly record the case where $\hat{s}_{i_1 i'}$ and $\hat{s}_{i_2 i'}$ are possibly the same for two different publishers $i_1, i_2 \in \mathcal{N}_{i'}$. The above problem can be solved as $|\mathcal{I}|$ independent multi-choice Knapsack problems [24, 25]. To see why this is the case, for a given subscriber i', the downlink is equivalent to a Knapsack with capacity $B^d_{i'}$, and a stream $s'_k \in \mathcal{S}_{ii'}$ can be viewed as an item with value $QoE_i(s'_k)$ and weight s'_k. There are $|\mathcal{N}_{i'}|$ number of classes, with each class containing $|\mathcal{S}_{ii'}|$ items, and we only pick at most one item from each class. The nice thing about such a formulation is that, despite that such a problem is a computationally hard problem, we can solve it using dynamic programming, which runs in pseudo-polynomial time [25, 26]. *Step1* determines the candidate set $\mathcal{D}_{i'}$ that contains the publisher and stream pairs that each subscriber $i' \in \mathcal{I}$ requests, but it depends on later steps whether or not these requests will be fulfilled.

Step1 is also visualized in Fig 5. Take client A as an example, A's downlink pipe can be viewed as a knapsack, and our goal is to pick one item from each of the B's and C's feasible bitrate sets to fill the knapsack such that its utility is maximized. The resulted bitrate selection is shown on the right. In this example, A subscribes to the $250Kbps(180p)$ bitrate from B, and the $1.1Mbps(720p)$ bitrate from C.

4.1.2 Step2 (Merge): addressing codec capability constraints. The result from *Step1* also equivalently determines the set $\mathcal{U}_i$ that contains subscriber and stream pairs that each publisher i is asked to serve, which can be denoted as,

$$\mathcal{U}_i = \{(i', s_{ii'}) : i' \in \mathcal{M}_i \wedge (i, s_{ii'}) \in \mathcal{D}_{i'}\} \tag{7}$$

Now given a particular resolution R, define a subset $\mathcal{U}_i^R$ of $\mathcal{U}_i$ as,

$$\mathcal{U}_i^R = \{(i', s_{ii'}) \in \mathcal{U}_i : Res(s_{ii'}) = R\} \tag{8}$$

For simplification purpose, let's consider three levels of resolutions, 720, 360, 180 [5], and thus a partition of $\mathcal{U}_i$ can be denoted as,

$$\mathcal{U}_i = \mathcal{U}_i^{720} \cup \mathcal{U}_i^{360} \cup \mathcal{U}_i^{180} \tag{9}$$

It is possible that $\mathcal{U}_i^R$ contains more than one stream bitrate, which violates the codec capability constraints discussed before. We introduce a merging function $Meg(\cdot)$ that maps the set $\mathcal{U}_i^R$ to an ordered pair as follow,

$$Meg(\mathcal{U}_i^R) = \begin{cases} \{(\mathcal{M}_i^R, s_i^R)\}, & \mathcal{U}_i^R \neq \emptyset \\ \emptyset, & \mathcal{U}_i^R = \emptyset \end{cases} \tag{10}$$

with,

$$\mathcal{M}_i^R = \{i' : (i', s_{ii'}) \in \mathcal{U}_i^R\} \tag{11}$$

$$s_i^R = \min_{(i', s_{ii'}) \in \mathcal{U}_i^R} s_{ii'} \tag{12}$$

Notice that an ordered pair $(\mathcal{M}_i^R, s_i^R)$ can be interpreted as a potential policy of publisher i as this: Publisher i intends to broadcast a stream with resolution R at bitrate s_i^R to a set of subscribers $\mathcal{M}_i^R$. Denote the updated results after merging as,

$$\mathcal{P}_i = Meg(\mathcal{U}_i^{720}) \cup Meg(\mathcal{U}_i^{360}) \cup Meg(\mathcal{U}_i^{180}) \tag{13}$$

It is not hard to see that $|\mathcal{P}_i|$ is smaller than the number of resolutions.

Step2 is also visualized in Fig. 5. In the example, B and C respectively subscribe to the $1.4Mbps(720p)$ bitrate and the $1.1Mbps(720p)$ bitrate from A, which are in the same resolution, so they are merged into one $1.1Mbps(720p)$ bitrate. In contrast, A and C respectively subscribe to the $250Kbps(180p)$ bitrate and the $1.4Mbps(720p)$ bitrate from B, which are in different resolutions, so both bitrates are kept in the potential policy.

4.1.3 Step3 (Reduction): addressing uplink constraints. After *Step1* and *Step2*, for each publisher $i \in \mathcal{I}$, we obtain a potential policy set $\mathcal{P}_i$ that satisfies both the downlink network constraint and the codec capability constraint. However, it is possible that $\mathcal{P}_i$ does not comply with the uplink network constraints. There are basically three situations:

- *Solution found.* If $\mathcal{P}_i, \forall i \in \mathcal{I}$ satisfies the uplink constraint with:

$$\sum_{(\mathcal{M}_i^R, s_i^R) \in \mathcal{P}_i} s_i^R \leq B_i^u, \forall i \in \mathcal{I} \tag{14}$$

  Then, $\mathcal{P}_i$ can be regarded as the optimal policy for each publisher i, and the algorithm terminates.

- *Solution violates constraints but is fixable.* If (14) is violated for certain $i \in \mathcal{I}$, that is

$$\sum_{(\mathcal{M}_i^R, s_i^R) \in \mathcal{P}_i} s_i^R > B_i^u \tag{15}$$

  then we will try to iterate over the policies $(\mathcal{M}_i^R, s_i^R)$ in $\mathcal{P}_i$ to see if (15) can be fixed by replacing s_i^R with some $s_i \in \mathcal{S}_i$ satisfying

$$\begin{cases} s_i < s_i^R \\ Res(s_i) = R \end{cases} \tag{16}$$

  which turns out to be a knapsack problem with a small number of feasible combinations, and thus can be brute forced easily. A necessary and sufficient condition for such fixing to be possible is that the sum of all such s_i that replaces the corresponding s_i^R must not exceed B_i^u when each s_i is selected as the smallest possible bitrate, that is

$$\sum_{(\mathcal{M}_i^R, s_i^R) \in \mathcal{P}_i} \min_{s_i \in \mathcal{S}_i^R} s_i \leq B_i^u, \quad i \in \mathcal{I} \tag{17}$$

  where $\mathcal{S}_i^R = \{s_i : s_i \in \mathcal{S}_i \wedge Res(s_i) = R\}$. If (17) holds, then we can for sure find an optimal policy for each publisher i by replacing one or more $(\mathcal{M}_i^R, s_i^R)$ in $\mathcal{P}_i$ with some $(\mathcal{M}_i^R, s_i)$ satisfying (16), and the algorithm terminates.

- *Solution violates constraints and is not fixable.* If there exists $i \in \mathcal{I}$ such that (17) is violated, then the problem can not be fixed by tuning down some s_i^R to a smaller bitrate s_i satisfying (16). This means that a certain resolution is not feasible and should be excluded from the feasible stream set $\mathcal{S}_i$. Let

$$\tilde{R}_i = \max_{(\mathcal{M}_i^R, s_i^R) \in \mathcal{P}_i} Res(s_i^R) \tag{18}$$

  then the updated feasible stream set is defined as

$$\mathcal{S}_i^{update} = \mathcal{S}_i \setminus \mathcal{S}_i^{\tilde{R}_i} \tag{19}$$

  where,

$$\mathcal{S}_i^{\tilde{R}_i} = \{s_i : s_i \in \mathcal{S}_i \wedge Res(s_i) = \tilde{R}_i\} \tag{20}$$

  Then we go back to *Step1* to start a new iteration given $\mathcal{S}_i^{update}$. It is worth noting that, we should shrink the feasible stream set for one publisher i at a time.

Step3 is also visualized in Fig. 5, where the potential policy for each client obtained after *Step2* is checked against the respective uplink bandwidth constraint. If the check is passed, then we obtain the final solution and terminate the algorithm. Otherwise, if (17) holds, we fix the policy by replacing bitrates with lower ones in the same resolution. Finally, if (17) does not hold, we reduce the stream set (e.g., removing all streams in 720p for A) and start the next iteration.

Convergence of the algorithm: The above algorithm converges because, in each iteration, it would either find a solution and terminate or reduce the feasible bitrate set for some publisher i. Therefore the number of iterations is limited by the number of publishers times the number of resolutions.

Examples: To provide a better idea of how the above algorithm works, we provide three examples in Table. 1. The stream resolutions, corresponding bitrate options, and their QoE utility values are

[5] The algorithm is readily extensible to more than three resolutions.

Table 1: Examples of GSO-Simulcast's control algorithm.

Bitrate levels / QoE				Client	Bandwidth		Subscription		Final solution		
Resolution	Bitrate(Bps)	QoE			Uplink	Downlink	Sub1	Sub2	720P	360P	180P
720P	1.5M	1200		A	5M	1.4M	A-sub-B-360P	A-sub-C-180P	1.5M	400K	
	1.3M	1050	case1	B	5M	3M	B-sub-A-720P	B-sub-C-360P		800K	100K
	1M	750		C	5M	500K	C-sub-B-360P	C-sub-A-760P		800K	300K
360P	800K	700		A	5M	5M	A-sub-B-360P	A-sub-C-180P	1.5M		
	600K	530	case2	B	600K	5M	B-sub-A-720P	B-sub-C-360P		600K	
	500K	440		C	5M	5M	C-sub-B-360P	C-sub-A-760P		800K	300K
	400K	360		A	5M	5M	A-sub-B-360P	A-sub-C-180P	1.5M	400K	
180P	300K	300	case3	B	600K	700K	B-sub-A-720P	B-sub-C-360P		600K	
	100K	100		C	5M	5M	C-sub-B-360P	C-sub-A-720P			300K

situated furthest to the left in the table. The clients, their subscriptions and the bandwidth constraints in the uplink and downlink are listed for each case and shown in the middle. Each case describes a different scenario: in case1, client C's downlink is limited; In case2, client B's uplink is limited; In case3, client B's uplink and downlink are both limited. The final solution that indicates the streams for each client to publish is shown on the right.

4.2 Seizing the global picture

In order to perform the centralized control as discussed in 4.1, GSO-Simulcast needs to capture the global picture of a meeting. Below, we discuss how to collect those types of information, including codec capabilities, subscription relations, and network bandwidths.

- *Subscription information.* There is not much change in the collection of subscription information compared to a meeting that does not employ GSO-Simulcast. As usual, the participants pass their subscription intents to the conference node via signaling channels.
- *Codec capability information.* The codec capability information is collected through the SDP negotiation process, which is carried out before a participant joins a meeting, but with minor modifications. We also send a customized *simulcastInfo* message together with the SDP offer to include additional information so that the conference node is not only able to collect the video codec type and the number of streams supported, but also the stream resolutions and the maximum bitrates with respect to each resolution. In the negotiation, we assign a different synchronization source (SSRC) for each stream resolution to facilitate the feedback control, which will be discussed later.
- *Bandwidth information.* The network information consists of the downlink bandwidths and the uplink bandwidths. In GSO, we rely on sender-side bandwidth estimation, which offers better accuracy than receiver-side estimation. As a result, collecting the downlink network bandwidths is relatively straightforward, as they can be directly reported from accessing nodes to the conference node. On the contrary, the uplink bandwidths are measured at the client-side and are disaggregated. To facilitate timely report of such disaggregated information, we utilize an in-band reporting approach, where each client reports the bandwidth information in an application-defined RTCP packet (type 204 in RFC 3550) [22]. The application-defined RTCP packet is intended for experimental use when new features are developed, which fits our purpose. More specifically, we define the message for sender estimated maximum bitrate (SEMB) following the definition of receiver estimated maximum bitrate (REMB) in [27]. The value of the reported bandwidth B is calculated as $B = Mantissa * 2^{Exp}$, where $Mantissa$ and Exp follow the definition in draft [27].

4.3 Feedback Control

Once GSO-Simulcast's controller has found a new solution, control feedback is sent to configure the streams for each sending participant. The timeliness of such feedback is essential. Hence, similar to the bandwidth report discussed above, the control feedback is also sent out with an in-band approach. We currently reuse the temporary maximum media stream bitrate request (TMMBR) message format for this purpose, which is defined in RFC 5104 [28]. The TMMBR conveys a temporary bandwidth limitation and is sent from an accessing node to a sending participant. TMMBR has been introduced as a transport layer feedback message (RTCP packet type 205). For example, RFC8888 [29] describes how to use TMMBR message to facilitate congestion control. Consequently, there is a potential ambiguity if we reuse TMMBR for a different purpose. In order to eliminate such ambiguity, we send the TMMBR for stream orchestration in an application-defined RTCP packet with type 204. Note that because we assign different SSRCs to streams of different resolutions as discussed in 4.2, the SSRC field in the TMMBR message allows us to specify which stream to configure. When a stream is disabled, the *MxTBR Mantissa* is set to zero. We also need to ensure the reliability of the feedback message. Because a TMMBR message is sent in an RTCP packet, there is no reliability guarantee on the packet's delivery. Therefore, upon receiving a TMMBR, a sending participant sends out a corresponding temporary maximum media stream bitrate notification (TMMBN) message [28]. If the accessing node does not receive the corresponding TMMBN message, it will re-send the TMMBR message, subsequently triggering the transmission of another TMMBN.

4.4 Managing Streams

An actual conference scenario is more complicated as video conferencing service providers today offer various advanced features. Covering each of the possible scenarios is out of the scope of this paper, but we discuss two basic capabilities that are quite fundamental in video conferencing: stream priorities and multi-stream subscriptions from the one publisher.

Stream priority. As pointed out earlier, not all streams are of the same importance in a conference. One nice feature about the

Knapsack problem formulation in *Step1* of Algorithm 4.1.1 is that we can easily incorporate stream priority by assigning different QoE utility weights to streams that come from different participants. For example, we can give the host's or speaker's streams higher QoE weights to ensure they are included in the solution of the Knapsack problem. Also, note that it is crucial to consider the ratio of the QoE utility over the bitrate of a stream. Because when two streams from different clients compete for the same bandwidth resource, we prefer to accommodate both with reduced bitrate than to drop one stream while conceding to the other. Therefore, we want to make sure that small streams have a higher QoE utility vs. bitrate ratio than large streams, so that small streams are protected.

Multi-stream subscriptions from one publisher. The Multi-Choice Knapsack formulation discussed in 4.1.1 is zero-or-one in nature, which permits no more than one stream from each publisher for a given subscriber. However, sometimes a receiving participant needs to subscribe to more than one stream from a sending participant. For example, in screen-share, a subscriber needs to subscribe to a screen stream in addition to a camera view. In another example, when a participant (denoted by X) in a meeting starts to speak loudly, the other participants would want to subscribe to a high-resolution camera view of this speaker in addition to a thumbnail view, a feature called "speaker first". To solve this problem, we add a virtual publisher X′ to the publisher set so that we still address the downlink constraints with the same problem formulation (1) in *Step1* 4.1.1 by treating X and X′ as different publishers. However, at the beginning of *Step2*, we merge X′ with X, so that we treat them again as the same publisher, which then allows us to move forward to the codec capability constraints in *Step2* 4.1.2 and uplink bandwidth constraints in *Step3* 4.1.3[6].

5 EVALUATION

This section presents the evaluation of GSO-Simulcast in four parts. In the first part, we evaluate GSO-Simulcast's control algorithm. The the second part, we study the transient response of GSO-Simulcast. The third part shows the results from various "slow-link" tests, which is an important step before a feature is launched into the product. We show the CPU usage on the client side in the fourth part.

Control algorithm: We start by evaluating the computation time and QoE optimality (accuracy) against the brute-force algorithm. In Fig. 6a, we vary the number of subscribers and publishers, and in Fig. 6b, we vary the number of bitrate levels. Due to the fact that the brute-force algorithm does not scale well when the meeting size is large, the experiment size is controlled in the first two experiments. We show large-scale results later in Fig. 6c without comparing them with the brute-force algorithm. We plot the normalized computation time on the left y-axis (in log-scale) and the QoE optimality on the right y-axis in Fig. 6a and Fig. 6b. The QoE optimality is measured as the ratio of the QoE summation in Eq. (1) of GSO's control algorithm vs. that of the brute-force algorithm.

- In the first experiment shown in Fig. 6a, the computation time of the brute-force approach grows exponentially with the number of participants[7], so it becomes intractable when the number of participants is large. In contrast, GSO's control algorithm is much more efficient. The QoE optimality in all cases is close to one, which verifies the accuracy of the algorithm.
- In the second experiment shown in Fig. 6b, the computation time of the brute-force approach grows exponentially with the number of bitrates, which makes it impossible to use when we want to implement fine-grained bitrate policies. In contrast, GSO's control algorithm scales linearly with the number of bitrates, enabling us to implement fine-grained bitrate policies. Again, in all cases, the QoE optimality is close to one, which shows the effectiveness of the GSO control algorithm.
- In the third experiment shown in Fig. 6c, we measure the computation time of the control algorithm when the meeting size is large. Each tuple is denoted as (# of publishers, # of subscribers, # of bitrates). The figure shows that the proposed control algorithm scales linearly with the number of subscribers and bitrates and quadratically with the number of publishers. Therefore, we can achieve real-time control with fine-grained bitrate policies even for meetings with hundreds of participants.

Transient response in real time: To understand how GSO adapts video bitrates to abrupt network changes in real time, we study the transient video bitrate response in Fig. 7. The x-axis shows the time while the y-axis shows the video bitrate. In the experiment, after 20s, we set downlink bandwidth limit to 750Kbps, 625Kbps, 500Kbps, and 375Kbps, respectively. The bandwidth is later recovered after 57s. As shown in Fig. 7a, GSO-Simulcast is able to quickly adapt to abrupt bandwidth changes. We can also see the benefit of fine-grained bitrates that is enabled with GSO-Simulcast, where in all cases, it perfectly fits the video bitrate just right under the bandwidth limit, resulting in high bandwidth utilization. In contrast, traditional Simulcast (Non-GSO-Simulcast) is not able to fit video bitrate into the bandwidth constraints due to limited number of bitrate levels, as shown in Fig. 7b. For example, when the network is limited to 625Kbps, coarse-grained Non-GSO-Simulcast has to downgrade to a 300Kbps stream while fine-grained GSO-Simulcast can choose to send a 600Kbps stream to fit the bandwidth constraint, and thus, reducing the network and bandwidth mismatch. Note that GSO-Simulcast's ability of using fine-grained bitrate is attributed to the fact that it treats the stream orchestration problem as a automated global optimization, while traditional Simulcast is driven by template-based empirical stream policy and it immediately becomes too complicated to scale to more than a few number of bitrate levels.

Slow-link experiments. Before a feature's deployment, we need to conduct a series of slow-link experiments to make sure it can cover a variety of corner cases. These cases are listed in Table. 2, which reflect various poor network conditions, including high packet losses, limited bandwidths, and significant jitters. The tests are carried out in a small meeting setup with specialized equipment so that we have full control of the network environments

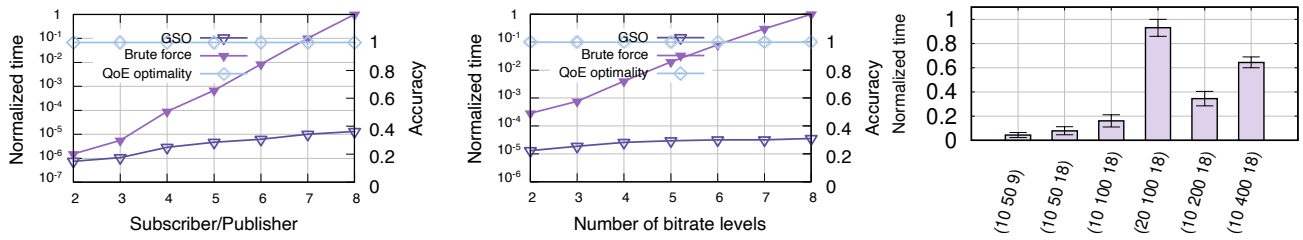
We show the normalized video framerate, video quality [8], as well as video stall rate [9] across different cases in Fig. 8. We compare the

[6] Note that a screen-share video and a camera video have different SSRC and will not be merged.

[7] shown as a straight line in the log-scale plot

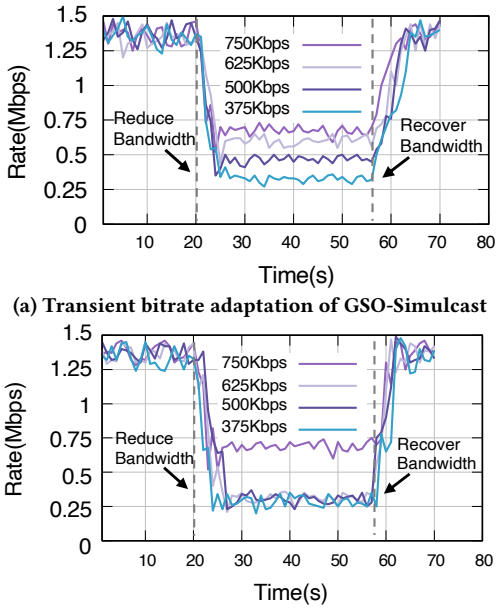
[8] Video quality is measured as VMAF score [30].

[9] Video stall is measured as the percentage of video playback intervals, in which the maximum delay between two consecutive frames is larger than 200ms.



(a) (b) (c)

Figure 6: Performance of GSO control algorithm. (a) and (b): the normalized computation time and accuracy(measured as QoE optimality of GSO vs. the brute force). (c): the normalized computation time when the meeting size and the number of bitrates become large.



(a) Transient bitrate adaptation of GSO-Simulcast

(b) Transient bitrate adaptation of Non-GSO-Simulcast

Figure 7: Transient video bitrate adaptation of (a) GSO-Simulcast, and (b) Non-GSO-Simulcast. After 20s, the down-link's bandwidth is abruptly limited to 750Kbps, 625Kbps, 500Kbps, and 375Kbps, respectively, and is later recovered.

performance of GSO to Non-GSO, and the other two commercial video conferencing apps from top competitors. Fig. 8 shows that GSO adapts to slow-links much more swiftly across all cases, which not only achieves better stability in framerate and video quality but also avoids video stalls. In contrast, Non-GSO and the competitors cannot handle all cases, in many of which they exhibit high video stall, poor visual clarity, and framerate drop.

CPU usage on client side: It is important to make sure that users do not experience high CPU usage when using GSO-Simulcast. In Fig. 9, we show the average CPU utilization of Dingtalk App (GSO version vs. Non-GSO version) in three different application scenarios (video conferencing, audio conferencing, screen sharing) measured on Huawei P30. The use of GSO-Simulcast in video conferencing and screen sharing only slightly increases the CPU usage. On the sender-side, the CPU usage increases less than 1%, and on the receiver-side, the CPU usage increases less than 2%.

Table 2: Network conditions used in slow-link tests.

Direction	Environment	Value	Case
uplink	jitter	50ms	up-50ms
		100ms	up-100ms
	loss	30%	up-30%
		50%	up-50%
	bandwidth-limit	0.5Mbps	up-0.5M
		1Mbps	up-1M
		1.5Mbps	up-1.5M
downlink	jitter	50ms	down-50ms
		100ms	down-100ms
	loss	30%	down-30%
		50%	down-50%
	bandwidth-limit	0.5Mbps	down-0.5M
		1Mbps	down-1M
		1.5Mbps	down-1.5M

The impact on audio conference is negligible, which is expected as pure audio communication is not handled by GSO-Simulcast. The above result shows that the impact GSO-Simulcast on the CPU utilization is minimal and meets our criteria for wide deployment.

6 DEPLOYMENT

To this date, GSO-Simulcast has been deployed at full-scale in Dingtalk and serves all our clients. This section presents the overall statistics of the core metrics [10], including the video stall, voice stall, and video framerate. We started our initial deployment of GSO-Simulcast on Nov. 20th, 2021, gradually increased the deployment coverage until we reached the point of full-scale deployment on Dec. 20th, 2021. We support up to 15 video bitrates in GSO-Simulcast based on the devices' codec capabilities. Due to confidentiality, all reported metrics are normalized against the largest value in the dataset. Our statistical data samples 1 million conferences per day, and the total dataset includes more than 100 million conference samples.

Video stall, voice stall, and video framerate. Fig. 10 reports the normalized average video stall, voice stall and video framerate by

[10] Voice stall is measured as the percentage of audio playback intervals whose audio packet loss is larger than 10%.

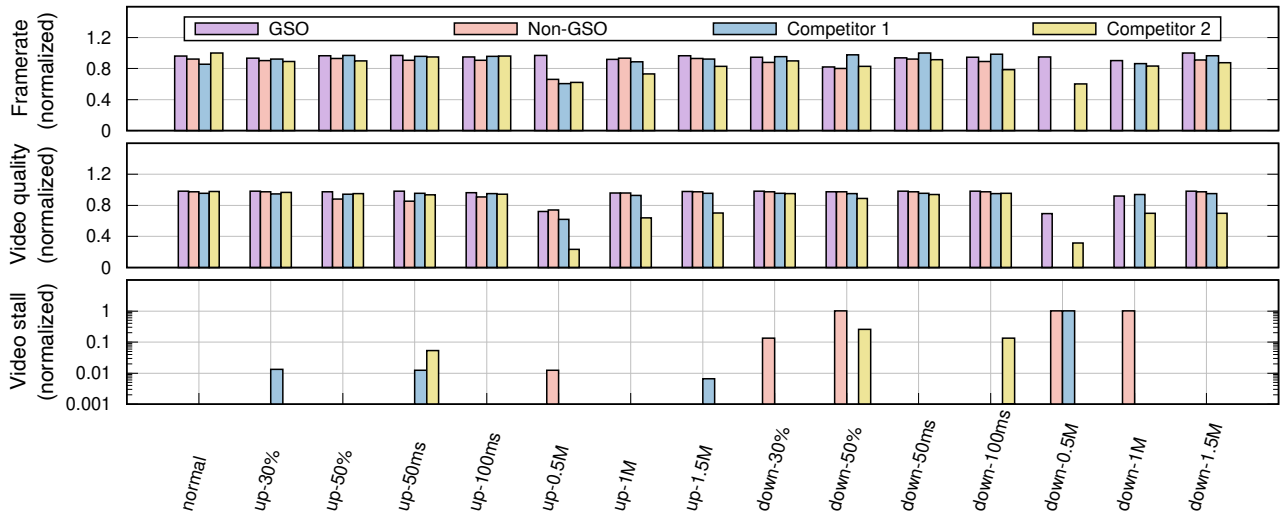


Figure 8: Slow-link test results that include the normalized video framerate, video quality, and video stall.

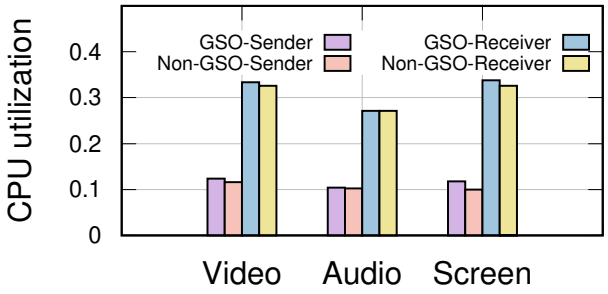


Figure 9: Average CPU utilization of Dingtalk App in different application scenarios measured on Huawei P30 smartphone.

date, which are the most critical metrics for user experience[11]. We make the following observations:

- With the deployment of GSO-Simulcast, all three metrics improved. The improvement became more significant as the deployment scale became larger, indicating strong positive correlations.
- Compared with the metrics before the initial deployment, the average video stall, voice stall, and video framerate after the full-scale deployment improved by 35%, 50%, and 6%, respectively, which are significant for large-scale video conferencing services.
- The improvements in video stall and video framerate were direct outcomes of GSO-Simulcast's stream orchestration. The improvement in voice stall was due to the reduced network congestion as GSO-Simulcast swiftly adapted video to slow-links at a fine-bitrate granularity.
- We observed significant improvement (7.2%) of users' satisfaction score (the percentage of users' positive feedback) as shown in Fig. 11. The user satisfaction score also increased with the deployment of GSO-Simulcast, indicating strong positive correlations.

[11]Because video quality analysis requires specialized tools, we currently do not collect this metric online.

Therefore, we conclude that GSO-Simulcast has significantly enhanced our users' experience and proved its value in the actual deployment.

Orchestration frequency: Fig. 12 plots the CDF of GSO-Simulcast's control algorithm 4.1 call interval, which is the time gap between two consecutive control events. A proper control frequency is key to the success of GSO-Simulcast. A control frequency that is too high control would waste computation power while a control frequency that is too low could not catch up with network changes in real time. In our deployement, GSO-Simulcast orchestrates streams every 1.8s on average. The maximum call interval is 3s, making sure the stream configuration is up to date. The minimum call interval is 1s, avoiding frequent updates that are unnecessary.

7 EXPERIENCE

In this section, we share the experience and lessons learned in deploying GSO-Simulcast.

Avoiding video quality oscillations. Bandwidth fluctuations in slow-links may cause GSO-Simulcast to readjust video bitrate back and forth frequently, thus causing a video quality oscillation that lowers users' visual comfort. Our solution is only to upgrade bitrate when we obtain enough confidence in the bandwidth measurements. In particular, we mark a video stream that has been downgraded, and when the controller later determines that an upgrade is needed, we only allow such an upgrade if the bandwidth increase has surpassed a threshold to filter out the noisy fluctuations in measurements.

Addressing bandwidth over-estimation. Bandwidth estimation plays an important role in delivering the desired performance. In GSO-Simulcast, we use transport-wide congestion control [31] for its flexibility. One problem we encountered was that GCC-like [32] congestion controls tend to over-estimate a link's bandwidth for a small stream as the associated loss rate and latency are typically low in this scenario. An overestimate might mislead GSO-Simulcast to

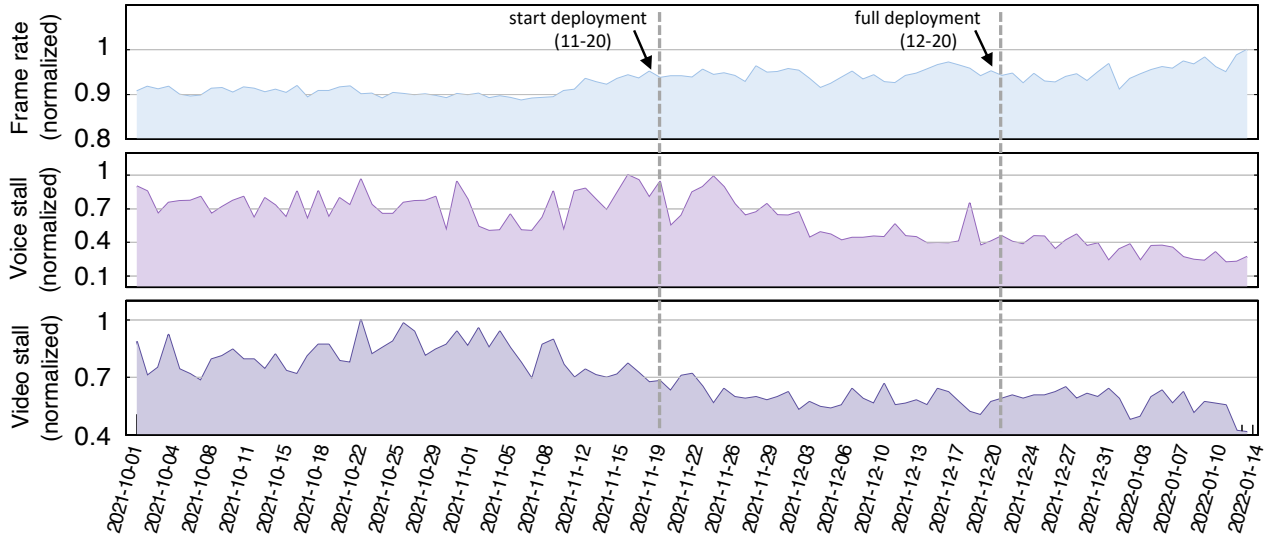


Figure 10: The overall statistics of core Dingtalk video conferencing metrics (average video stall, voice stall, and video framerate), from Oct.1st, 2021, to Jan.14th, 2022. The deployment of GSO-Simulcast started on Nov. 20th, 2021 and reached full scale on Dec. 20th, 2021. All metrics are normalized, sampled from 1 million conferences per-day.

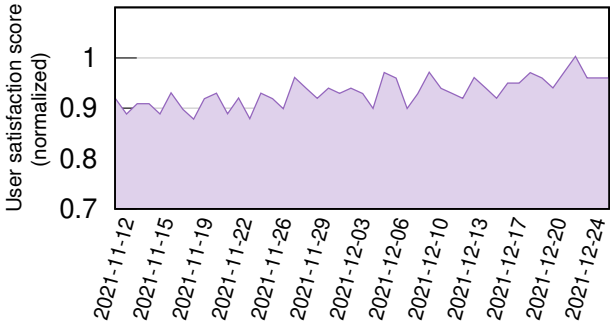


Figure 11: The overall statistics of Dingtalk's user satisfaction score (normalized) from Nov. 12th, 2021 to Dec. 24th, 2021.

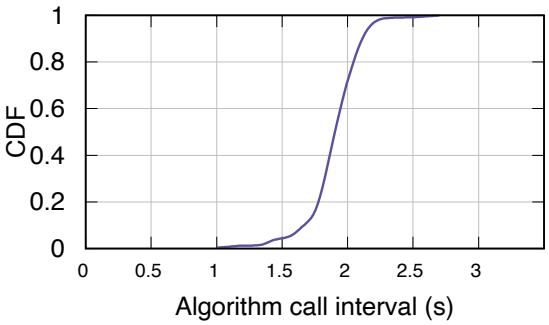


Figure 12: CDF of GSO-Simulcast's control algorithm call interval.

start pushing high-bitrate stream, causing link congestion. To address this issue, we send probing packets in short bursts controlled by a pacer to probe the bandwidth upper bound. It is worth noting that the control of the probing redundancy needs to be carefully adjusted to reduce the traffic overhead.

Reducing message reporting frequency. It is critical to control bandwidth reporting message frequency. Otherwise, we might overwhelm the conference node. We implement both a time trigger and an event trigger. The time trigger periodically updates the measurements while the event trigger is fired to update bandwidth only if its change is significant.

Protecting audios. Even though an audio signal occupies small bandwidth, audio loss is often less tolerable. Therefore, we need to leave sufficient bandwidth room for the audio signal. In GSO-Simulcast, when we obtain a bandwidth measurement, we subtract a "protection" bandwidth from it to further avoid video streams eating the audio stream's bandwidth.

Design for failure. Dingtalk must maintain the service when failure happens. It becomes particularly important for system like GSO-Simulcast where centralized control is involved. On the client side, one typical exception is that while a server instructs a client to send multiple streams, however, only a low bitrate stream is received. In such a scenario, GSO-Simulcast implements a downgrade logic that automatically switches the high-bitrate subscription to a low-bitrate subscription. One the server side, when an exception is raised, GSO-Simulcast would ask clients to fall back to single stream configuration so that the service could continue, however, at the cost of reduced QoE.

8 RELATED WORK

GSO-Simulcast relates to past works in the following areas:

State-of-the-art Simulcast: Simulcast has gained popularity in recent years [11] due to its cost-effectiveness and scalability, and is currently supported by a number of implementations, including, Chrome [13], Amazon Chime [12] and Janus Gateway [33]. However, all these solutions adapt video based on a fragmented network view, and thus, they are limited by a number of issues such as video and network mismatch, poor network utilization, and

link congestion that lead to suboptimal performance. Moreover, they typically use template-based adaptation policies [12]. Even with coarse-grained bitrates, such policies are already complicated. Adding more fine-grained bitrates would make such template-based policies unmanageable. Note that a global flow optimization approach was considered in partitioned-simulcast [34, 35]. However, [34] was limited to theoretical study and only had simulation results up to 6 users, while [35] only considered a one-to-many scenario where a single sender was in the conference. Evaluation in [35] was also limited to a setup of up to 10 receivers. In contrast, GSO-Simulcast is the first deployed Simulcast that uses a global stream control and optimization approach, and overcomes various limitations in the state-of-the-art Simulcast solutions.

P2P-based and MCU-based architectures: In the P2P-based architecture, participants send media streams to each other directly, without the involvement of an intermediary server [36–39]. The P2P-based architecture does not scale well in a multi-party conference [36, 40], where the total number of required P2P connections would increase quadratically with the conference size. Handling a large number of connections not only puts processing burden on the peer, but also causes congestion in the paths. In the MCU-based architecture [41–43], each participant establishes a connection with the MCU server. After receiving the upstream media data, the MCU generates a composite stream containing all of the upstreams received from the participants. Compared to the P2P-based architecture, the MCU-based architecture allows more efficient network usage, but an MCU generally needs to mix a lot of video streams in real-time, and thus, is expensive due to its need for a lot of processing power.

Transcoding and SVC: Transcoding [6–8] is the process of converting a digital video stream from one format to another. With transcoding, a centralized media server can change the bitrate of a media stream via transrating or transsizing. However, transcoding places a significant burden on the media server, which is costly [9]. Scalable video coding (SVC) [10] is a technique that encodes a video stream in multiple layers, either through temporal scalability or spatial scalability. SVC has received a lot of interests [44], as it allows a single video stream to adapt to multiple bitrates. However, the problem with SVC is codec interoperability [45], as it would need every participant in a video conference to support scalable encoding/decoding. Many hardware H264 codecs today do not support scalability [45].

Codec and transport collaboration: There is another type of adaptation in video conferencing, which is through the codec and transport collaboration [46, 47]. Typically, a codec's output rate is adjusted with feedback from congestion controller [32]. Salsify [47] implements a codec that permits frame-by-frame adaptation to enable an even tighter collaboration. Such adaptation is helpful to reduce single link congestion and is also incorporated in our stack, but in a multi-party conference that is not P2P-based, its power is limited as an upstream codec cannot adapt to multiple downstream networks simultaneously. In contrast, GSO-Simulcast overcomes such a limitation with the ability to coordinate upstreams and downstreams.

ML-based approaches: We have noticed that there is a growing trend of applying machine-learning techniques to networking systems in recent years [48, 49]. However, we decide to use a control-theoretical approach in GSO-Simulcast because of the following reasons: first, machine learning approaches require a large amount of high-quality data for training purposes, which is quite expensive to collect in our scenario. Second, we care about the long tail performance, which means that our system needs to perform well even in the cases not adequately represented in the data set. Finally and more importantly, the control algorithm presented in Sec. 4.1 can already achieve near optimal performance. Even though we find the control-theoretical approach more suitable for our system in production for the present, we do think machine-learning based approaches worth more investigation if more sophisticated objectives are involved.

9 CONCLUSION

This paper introduces GSO-Simulcast, the first widely deployed video conferencing system that uses global control stream orchestration in Simulcast and brings significant values to more than 100 million users to the best of our knowledge. Since GSO-Simulcast is decoupled from the underlay infrastructure, it is suitable for multi-cloud deployment. Looking into the future, we believe the techniques of GSO-Simulcast can be applied to a wide range of applications, including real-time communication in the upcoming meta-universe, where it becomes even more challenging to support 3D interactive video conversations.

ACKNOWLEDGMENTS

We are grateful to the our shepherd Haiying Shen and the anonymous reviewers for their insightful comments and feedbacks, which have greatly improved the quality of this paper. We thank all teams at Alibaba that help deploy GSO-Simulcast, especially Zhigang Jiang, Jiong Zhang, Ruixin Li, and Yiyuan Huang from Dingtalk, as well as Jianhua Chen and Shengyang Xu from Alibaba Cloud. We thank Yirui Liu and Xuan Zeng for providing insightful feedback to this paper.

REFERENCES

[1] Zoom. https://zoom.us/, 2022.
[2] Microsoft teams. https://www.microsoft.com/en-us/microsoft-teams/group-chat-software, 2022.
[3] Google meet. https://apps.google.com/meet, 2022.
[4] Amazon chime. https://aws.amazon.com/chime/?nc=sn&loc=0&chime-blog-posts.sort-by=item.additionalFields.createdDate&chime-blog-posts.sort-order=desc, 2022.
[5] Dingtalk. https://www.dingtalk.com/en, 2022.
[6] Ishfaq Ahmad, Xiaohui Wei, Yu Sun, and Ya-Qin Zhang. Video transcoding: an overview of various techniques and research issues. *IEEE Transactions on multimedia*, 7(5):793–804, 2005.
[7] Bo Shen, Wai-Tian Tan, and Frederic Huve. Dynamic video transcoding in mobile environments. *IEEE MultiMedia*, 15(1):42–51, 2008.
[8] G Camarillo, E Burger, H Schulzrinne, and A Van Wijk. Transcoding services invocation in the session initiation protocol (sip) using third party call control (3pcc). *IETF RFC 4117*, 2005.
[9] Transcoder api pricing. https://cloud.google.com/transcoder/pricing, 2022.
[10] Thomas Schierl, Alex Eleftheriadis, Stephan Wenger, and Ye-Kui Wang. RTP Payload Format for Scalable Video Coding. RFC 6190, May 2011.
[11] Simulcast webrtc 1.0. https://www.w3.org/2011/04/webrtc/wiki/images/a/a7/Simulcast_in_WebRTC.pdf, 2021.
[12] Amazon chime sdk. Video simulcast. https://aws.github.io/amazon-chime-sdk-js/modules/simulcast.html, 2021.
[13] Simulcast playground. https://webrtccourse.com/course/webrtc-codelab/module/fiddle-of-the-month/lesson/simulcast-playground/, 2021.
[14] Bo Burman, Magnus Westerlund, Suhas Nandakumar, and Mo Zanaty. Using Simulcast in Session Description Protocol (SDP) and RTP Sessions. RFC 8853, January 2021.
[15] Working with vp8 simulcast. https://www.twilio.com/docs/video/tutorials/working-with-vp8-simulcast, 2022.
[16] Amazon chime sdk. https://aws.amazon.com/chime/chime-sdk/, 2021.
[17] Chromium simulcast. https://source.chromium.org/chromium/chromium/src/+/main:third_party/webrtc/modules/video_coding/utility/simulcast_rate_allocator.cc;bpv=1;bpt=1, 2022.
[18] Ari Keränen, Christer Holmberg, and Jonathan Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal. RFC 8445, July 2018.
[19] Colin Perkins, Mark J. Handley, and Van Jacobson. SDP: Session Description Protocol. RFC 4566, July 2006.
[20] Philip Matthews, Jonathan Rosenberg, Dan Wing, and Rohan Mahy. Session Traversal Utilities for NAT (STUN). RFC 5389, October 2008.
[21] Philip Matthews, Jonathan Rosenberg, and Rohan Mahy. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766, April 2010.
[22] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, and Van Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550, July 2003.
[23] Jonathan Lennox, Magnus Westerlund, Qin Wu, and Colin Perkins. Sending Multiple RTP Streams in a Single RTP Session: Grouping RTP Control Protocol (RTCP) Reception Statistics and Other Feedback. RFC 8861, January 2021.
[24] List of knapsack problems. https://en.wikipedia.org/wiki/List_of_knapsack_problems, 2021.
[25] Nick Gavalas. Solving the multiple choice knapsack problem. https://nickgavalas.com/solving-the-multiple-choice-knapsack-problem/, 2019.
[26] Pseudo-polynomial time. https://en.wikipedia.org/wiki/Pseudo-polynomial_time, 2021.
[27] Harald T. Alvestrand. RTCP message for Receiver Estimated Maximum Bitrate. Internet-Draft draft-alvestrand-rmcat-remb-03, Internet Engineering Task Force, October 2013. Work in Progress.
[28] Bo Burman, Stephan Wenger, Magnus Westerlund, and Umesh Chandra. Codec Control Messages in the RTP Audio-Visual Profile with Feedback (AVPF). RFC 5104, February 2008.
[29] Zaheduzzaman Sarker, Colin Perkins, Varun Singh, and Michael A. Ramalho. RTP Control Protocol (RTCP) Feedback for Congestion Control. RFC 8888, January 2021.
[30] Wikipedia. Video multimethod assessment fusion. https://en.wikipedia.org/wiki/Video_Multimethod_Assessment_Fusion, 2021.
[31] Stefan Holmer, Magnus Flodman, and Erik Sprang. RTP Extensions for Transport-wide Congestion Control. Internet-Draft draft-holmer-rmcat-transport-wide-cc-extensions-01, Internet Engineering Task Force, October 2015. Work in Progress.
[32] Stefan Holmer, Henrik Lundin, Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo. A Google Congestion Control Algorithm for Real-Time Communication. Internet-Draft draft-ietf-rmcat-gcc-02, Internet Engineering Task Force, July 2016. Work in Progress.
[33] The janus gateway. https://github.com/meetecho/janus-gateway/tree/master/html, 2022.
[34] Eymen Kurdoglu, Yong Liu, and Yao Wang. Dealing with user heterogeneity in p2p multi-party video conferencing: Layered distribution versus partitioned simulcast. *IEEE Transactions on Multimedia*, 18(1):90–101, 2015.
[35] Stefano Petrangeli, Dries Pauwels, Jeroen van der Hooft, Tim Wauters, Filip De Turck, and Jürgen Slowack. Improving quality and scalability of webrtc video collaboration applications. In *Proceedings of the 9th ACM Multimedia Systems Conference*, pages 533–536, 2018.
[36] Yang Xu, Chenguang Yu, Jingjiang Li, and Yong Liu. Video telephony for end-consumers: Measurement study of google+, ichat, and skype. In *Proceedings of the 2012 Internet Measurement Conference*, pages 371–384, 2012.
[37] Peer-to-peer communications with webrtc. https://developer.mozilla.org/en-US/docs/Web/Guide/API/WebRTC/Peer-to-peer_communications_with_WebRTC, 2021.
[38] Chao Liang, Miao Zhao, and Yong Liu. Optimal bandwidth sharing in multiswarm multiparty p2p video-conferencing systems. *IEEE/ACM Transactions On Networking*, 19(6):1704–1716, 2011.
[39] Yu Wu, Chuan Wu, Bo Li, and Francis CM Lau. vskyconf: Cloud-assisted multi-party mobile video conferencing. In *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*, pages 33–38, 2013.
[40] Cloud-based and peer-to-peer meetings. https://blog.zoom.us/cloud-based-and-peer-peer-meetings/, 2021.
[41] TrueConf Team. Mcu (video conferencing architecture). https://trueconf.com/blog/wiki/mcu-video-conferencing-architecture, 2019.
[42] Mukund Iyengar. Webrtc architecture basics: P2p, sfu, mcu, and hybrid approaches. https://medium.com/securemeeting/webrtc-architecture-basics-p2p-sfu-mcu-and-hybrid-approaches-6e7d77a46a66, 2021.
[43] Magnus Westerlund and S Wenger. Rtp topologies. Technical report, RFC 5117, January, 2008.
[44] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable video coding extension of the h. 264/avc standard. *IEEE Transactions on circuits and systems for video technology*, 17(9):1103–1120, 2007.
[45] W3C Working Draft. Scalable video coding (svc) extension for webrtc. https://www.w3.org/TR/webrtc-svc/, 2021.
[46] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Analysis and design of the google congestion control for web real-time communication (webrtc). In *Proceedings of the 7th International Conference on Multimedia Systems*, pages 1–12, 2016.
[47] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 267–282, 2018.
[48] Zili Meng, Minhu Wang, Jiasong Bai, Mingwei Xu, Hongzi Mao, and Hongxin Hu. Interpreting deep learning-based networking systems. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 154–171, 2020.
[49] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 197–210, 2017.