Université de Strasbourg

THÈSE

pour obtenir le grade de
DOCTEUR de l'UNIVERSITÉ de STRASBOURG
Spécialité: **Informatique**

Présenté par

Boris GROZEV
dirigée par Emil Ivov et Thomas Noel

Soutenu le **20/12/2019**

Préparée au sein du

Laboratoire ICube de l'Université de Strasbourg et la société **8x8 Londres**,
dans l'Ecole Doctorale Mathématiques, **Sciences de l'Information et de l'Ingénieur**.

Efficient and Scalable Video Conferences With Selective Forwarding Units and WebRTC

Jury de thèse composé de :

Marcelo Dias De Amorim
Directeur de Recherche CNRS, Examinateur

Dr. Emil Ivov
Directeur Developpement Produit, Collaboration Vidéo à 8x8, Encadrant de thèse

Nicolas Montavont
Professeur à l'IMT Atlantique, Rapporteur

Thomas Noel
Professeur à l'Université de Strasbourg, Directeur de thèse

Hervé Rivano
Professeur à l'INSA de Lyon, Rapporteur

Acknowledgements

During the last five years I have been very fortunate to have people around me who supported and helped me in various ways, and I would like to recognize them here.

This work would not have been even started, let alone completed, without the help of Emil Ivov. He encouraged me to pursue a Ph.D. in the first place, and supported me through the whole process as a research supervisor, manager at work, and on a personal level. He often believed in me more than I did myself, and inspired me to set higher goals and work harder. Finding the right balance between engineering and research was one of the hardest challenges I faced, and Emil went to great lengths to find compromises that allowed me to do both. Thank you Emil!

Thomas Noel guided me to the world of academia and publishing, and was instrumental in shaping this thesis. He read early versions of the manuscript and offered invaluable advice, always delivered in a practical and constructive way. He did more than I could ever expect to enable me to complete my work remotely: understanding and translating my broken French, communicating with administrative staff on my behalf, making sure I meet all the deadlines, finding and organizing the jury, and arranging to use video conferencing for the defence at very short notice. Merci Thomas!

I would also like to thank my jury for their efforts: Marcelo Dias De Amorim, Nicolas Montavont and Herve Rivano.

My colleagues Damian Minkov, Brian Baldino, Aaron van Meerten, Saúl Ibarra Corretgé, George Politis, Yana Stamcheva, Pawel Domas, Lyubomir Marinov and everyone else on the Jitsi team have been part of countless technical discussions and contributed with ideas and criticism, which I am grateful for. I learned and continue to learn from them.

George Politis has been my main collaborator for the research papers this thesis is based upon. He contributed with ideas, performing experiments and shaping the text. The other co-authors include Thomas Noel, Emil Ivov, Varun Singh, Pawel Domas, Alexandre Gouaillard and Lyubomir Marinov and I am thankful for the opportunity to work with them.

Varun Singh listened to my ideas and problems, and offered a fresh perspective and advice at some key points in my journey, and I am very grateful for it.

Pierre David participated in my mid-term and pre-defence presentations. Both times he asked insightful and revealing questions and suggested better ways to organize my work.

Erin Cashel meticulously went through the manuscript and made it more understandable (or often just understandable), clear and correct. Thank you!

I am also grateful to the companies that provided employment to me during the last five years: SIP Communicator/BlueJimp, Atlassian, and 8x8.

Yana Stamcheva has been the best manager I could ask for, giving me freedom in my work and making sure I get everything I need, as well as a friend to me. Damian Minkov and Eliza Dikova have been my friends and welcomed me in their home on more than one occasion. Thanks, guys!

Erin Cashel was next to me providing support and encouragement while I was writing this thesis. Thank you for sharing your life with me!

Finally I would like to thank my family back home for always unquestioningly supporting me and being understanding of my decisions. Mom, Dad, Nia, I love you!

Contents

List of Figures

List of Tables

Chapter 1

Résumé

Le concept d'un appareil de communication à distance intégrant à la fois l'image et le son retour à la fin du 19e siècle, peu de temps après l'invention du téléphone.[1]. Mais la technologie pour la rendre d'abord possible et ensuite pratique a mis plusieurs décennies à se développer.

Les premiers systèmes ont été construits à la fin des années 1920 et combinaient les premiers téléviseurs avec une ligne téléphonique bidirectionnelle [56]. Initialement excessivement encombrant et peu pratique (une salle pleine d'équipements), avec le temps est venue la réduction de taille nécessaire pour viabilité commerciale.

Tout au long des années 1960 et 1970, il y avait sur le marché plusieurs appareils téléphoniques avec capacité vidéo, ce qui travaillé avec un signal analogique sur le réseau téléphonique. Cependant, ils n'ont pas gagné en popularité ou en succès commercial.

Les progrès de la compression vidéo dans les années 1980 et 1990 ont conduit à des progrès significatifs. Ils ont permis de transporter un signal numérique sur les réseaux informatiques existants [25]. Cela a conduit au développement de systèmes de téléprésence, généralement une salle dédiée (parfois avec un réseau) équipé de plusieurs écrans, caméras et microphones. Alors que la fidélité augmentait, le prix était encore très élevé (plus de 100 000 $). Malgré le coût élevé, ces systèmes avaient ont enfin trouvé leur place sur le marché: les grandes entreprises ayant besoin de telles pièces et capital pour les acheter.

[1] http://www.terramedia.co.uk/Chronomedia/years/Edison_Telephonoscope.htm

Avec l'amélioration rapide des appareils et des réseaux informatiques au cours des années 1990 et au début des années 2000, il est devenu de plus en plus possible d'utiliser des appareils grand public et Internet public pour l'audio et la communication vidéo. Diverses technologies et normes, notamment SIP [66], ont été développé pour faciliter les appels téléphoniques sur Internet et les interconnecter au réseau téléphonique public, et le terme " Voice over the Internet Protocol ", ou VoIP a été inventé. Bien que des capacités vidéo soient également disponibles, elles gagner immédiatement en popularité.

Les applications de communication vidéo en ligne ont commencé à devenir plus populaires à la fin des années 2000, notamment avec l'essor de Skype. Avec la prolifération de puissants appareils mobiles personnels vers 2008, le matériel requis pour la communication audio et vidéo est devenu accessible à un une partie importante et croissante de la population mondiale. Cela a révélé une multitude de de nouvelles applications potentielles.

Aujourd'hui en 2019, les outils de communication vidéo font partie intégrante de la vie quotidienne de millions de les gens, à la fois personnellement et professionnellement. Il y a encore beaucoup plus de millions avec accès aux appareils capables de visioconférence, c'est-à-dire aux utilisateurs potentiels dans un marché inexploré. La technologie a fait d'énormes progrès, mais c'est toujours un domaine en pleine évolution, et il existe de nombreuses façons de l'améliorer. Ce grand potentiel d'affecter un un nombre considérable de personnes en faisant progresser la technologie est ce qui en fin de compte vaut la peine d'étudier la vidéoconférence.

1.1 Applications de la vidéoconférence

La vidéoconférence devient omniprésente. À mesure que la technologie se développe, la fidélité, la fiabilité, le coût et l'accessibilité s'améliorent, conduisant à de nouvelles applications. Certaines des applications pour lesquelles la vidéoconférence est actuellement utilisée sont:

- *Communication personnelle.* Les personnes qui parlent de manière informelle avec leurs amis ou famille, parfois dans des endroits éloignés. Applications populaires telles que Facebook Messenger et WhatsApp sont souvent utilisés de cette façon.
- *Collaboration à distance*. Des équipes de professionnels dispersés géographiquement collaborer sur le même projet [84]. Cela devient de plus en plus populaire pour les équipes de développement de logiciels, organisant des réunions à distance planifiées ou ad hoc.

- *Télémédecine.* Fournir des services de soins de santé à distance tels que consultation avec des médecins et d'autres médecins professionnels [37], suivi des patients, diagnostics et autres. Cette application a été envisagée il y a longtemps[2], mais ses exigences uniques de fiabilité et de confidentialité la rendent très difficile à adopter dans la pratique. Il a le potentiel de transformer la façon dont les soins médicaux sont à condition de [42].

- *Webinaires.* Présentations interactives en ligne, avec la participation active des participants [90].

- *Interview à distance.* Mener des entretiens d'embauche à distance.

- *Productivité personnelle.* Services visant à augmenter la productivité en associant les participants aux sessions vidéo[3].

- *Service clients.* Fournir un service client et un support à distance.

- *Consultations avec des professionnels.* Mener des consultations à distance avec des experts dans divers domaines.

- *Enseignement à distance.* Fournir l'enseignement et la formation à distance. Les étudiants peuvent participer à une classe en ligne avec un professeur [48].

- *Entretiens d'affaires.* Réunions d'entreprise planifiées entre différents bureaux. Ceci est de plus en plus utilisé à la fois des réunions privées " toutes mains " de l'entreprise, ainsi que des réunions publiques de type " appel aux résultats ".

1.2 Défis et motivations

Aujourd'hui, les gens utilisent une grande variété d'appareils capables de visioconférence. Ceci comprend ordinateurs de bureau et portables personnels, téléphones mobiles et tablettes, smart des appareils de télévision et des salles de conférence dédiées. Ces appareils ont des caractéristiques très différentes telles que la puissance de calcul, la prise en charge du codec matériel, caméras, microphones et tailles d'affichage. Ils sont aussi connecté à divers réseaux allant des liaisons Ethernet à large bande passante et à faible latence, à des réseaux cellulaires limités et instables ou même à des liaisons satellites directes. Cela représente un défi pour les applications qui doivent utiliser le efficacement les ressources disponibles tout en s'adaptant aux conditions changeantes.

[2]L'American Telemedicine Association a été fondée en 1993.
[3]https://www.focusmate.com/

La transmission vidéo en temps réel est largement connue pour nécessiter des calculs et une bande passante considérables. cela est encore plus vrai pour la conférence lorsqu'elle implique plus de deux points de terminaison. Les systèmes de visioconférence traditionnels utilisent beaucoup de ressources, soit sur leur périphériques d'extrémité ou déchargés sur des serveurs distants.

Le développement d'architectures plus efficaces a un double effet. Il sert mieux les utilisateurs car il améliore la qualité de service dans les réseaux non idéaux, élargissant le bassin potentiel de consommateurs. Il est également avantageux pour les propriétaires de services car il réduit les coûts d'exploitation des services.

Les développements récents dans le domaine de la communication vidéo ont conduit à l'introduction d'architectures basées sur Selective Forwarding Units (SFUs [20]). Ces systèmes ont le potentiel de améliorer l'efficacité et offrir de nouvelles fonctionnalités, mais ils présentent leurs propres défis, nécessitant de nouvelles approches et solutions. Cette thèse est dédiée à l'étude des systèmes SFU, avec un focus sur leur réalisation plus efficace, améliorant la qualité de service et réduisant les coûts.

Dans la section suivante, nous décrivons les contributions de cette thèse, avant de poursuivre avec un examen de l'état de l'art dans le chapitre suivant.

1.3 Contributions

Cette section présente les contributions de la thèse et les recherches sur lesquelles elle s'appuie. Pour conserver la description concis, nous supposons une familiarité avec certains des concepts décrits plus loin dans le texte. Lecteurs peu familiers avec la terminologie peut trouver une introduction aux sujets connexes dans le chapitre 3.

Le principal objectif de cette thèse est d'augmenter l'efficacité de la visioconférence systèmes tout en améliorant l'expérience globale de l'utilisateur. On commence par proposer un algorithme efficace qui utilise l'activité vocale pour sélectionner un sous-ensemble de points de terminaison dans une conférence. Transfert de vidéo uniquement pour les points de terminaison dans cet ensemble réduit considérablement la bande passante requise dans une conférence, ce qui la rend plus efficace. Ensuite, nous présentons une évaluation approfondie de l'utilisation de la diffusion simultanée pour la vidéoconférence, ce qui réduit encore la bande passante requise. Ensuite, nous proposons, analysons et évaluons un hybride d'une architecture peer-to-peer et basée sur serveur, qui améliore l'expérience utilisateur et réduit les coûts de service.

Nous explorons également les performances du serveur en termes d'utilisation du processeur et de gestion de la mémoire. Nous proposons des optimisations de gestion de la mémoire visant pour réduire la pression sur le ramasse-miettes. Nous développons une méthodologie pour mesurer les effets des pauses d'application en raison de la collecte des ordures, et évaluer la solution proposée dans différentes configurations. Les résultats sont positifs et découvrir des domaines de recherche supplémentaires.

Ensuite, nous proposons une approche de serveur distribué dans le but d'améliorer la qualité de connexion des utilisateurs en utilisant serveurs à proximité géographique de chaque point de terminaison. Nous présentons une première analyse, une implémentation et une expérience randomisée mesurer la qualité de la connexion. Les résultats montrent que, en ce qui concerne la qualité de la connexion, l'approche offre des améliorations dans certains cas, mais les effets sont négatifs. Nous proposons d'autres recherches pour comprendre comment appliquer la technique de manière optimale.

Un autre objectif de la thèse est d'explorer comment les systèmes basés sur des unités de transfert sélectif peuvent être modifiés pour prendre en charge fonctionnalités disponibles dans d'autres architectures. Nous proposons une extension des spécifications existantes pour des conférences chiffrées de bout en bout, permettant l'utilisation de la diffusion simultanée et de certains mécanismes de réparation de flux. nous mettre en œuvre un système de validation de principe et afficher des frais généraux acceptables.

Cette thèse est en partie basée sur des recherches antérieures de l'auteur, publié dans plusieurs revues et conférences internationales à comité de lecture. Une liste complète des publications est disponible à l'annexe A.

1.4 Structure de la thèse

Ce manuscrit est organisé en neuf chapitres. Ce premier chapitre sert de courte introduction. Chapitre 3 présente les pratiques actuelles et l'état de l'art en vidéoconférence, et présente la pile technologique utilisée tout au long de la thèse.

À partir du chapitre 4, chaque chapitre présente l'une des contributions de cette thèse.

Chapitres 4, 5 et 6 examiner trois techniques différentes pour réduire le besoins en bande passante d'une conférence. Le chapitre 4 présente le schéma LastN pour sélectionner un ensemble de points de terminaison en fonction du niveau audio les indications. Le chapitre 5 détaille les utilisation de la diffusion simultanée pour les vidéoconférences et présente une évaluation expérimentale approfondie. Chapitre 6 propose

un système qui bascule dynamiquement entre une architecture peer-to-peer et basée sur serveur.

Le chapitre 7 examine les performances des Implémentations de l'unité de transfert sélectif en termes d'utilisation du processeur et de pauses du récupérateur de place.

Le chapitre 8 présente l'idée d'utiliser un ensemble de serveurs distribués afin d'améliorer la qualité de connexion des terminaux.

Le chapitre 9 propose des extensions à un système pour les conférences chiffrées de bout en bout afin de soutenir la diffusion simultanée et d'améliorer les mécanismes de réparation des flux.

Le chapitre 10 conclut cette thèse en présentant remarques finales et discussion de certaines perspectives.

La thèse comprend également deux annexes. L'annexe A répertorie les publications, qui sont à la base de cette thèse. Détails de l'annexe B plusieurs brevets issus des recherches de l'auteur.

1.5 Architectures de vidéoconférence: état de l'art

Dans cette thèse, nous nous intéressons à un type de système spécifique pour la vidéo conférence - celles qui permettent à un groupe de personnes de communiquer sur Internet en temps. Nous utilisons les termes " vidéoconférence " ou " conférence " de manière interchangeable pour désigner ce terme. type spécifique de conférence. Par conséquent, les systèmes qui n'autorisent que deux points d'extrémité, ou les systèmes pour le streaming en direct qui ne convient pas à la communication en temps réel ne présente pas d'intérêt immédiat. Plus spécifiquement, nous nous intéressons aux vidéo-conférences avec une architecture particulière - celles utilisant une unité de routage sélective (Selective Forwarding Unit, SFU). Dans la section suivante, nous présentons les différentes topologies qui ont été utilisé pour les conférences multipartites, y compris les SFU, et comparer leurs caractéristiques.

Les systèmes de visioconférence peuvent être classés en fonction de la façon dont leurs points d'extrémité se connectent et le type de traitement multimédia effectué par les composants du serveur (le cas échéant). Dans cette section, nous discutons de la architectures les plus utilisées, en particulier celles basées sur Selective Forwarding Units, qui fait l'objet de cette thèse. Nous prenons un comparatif approche et examiner

les caractéristiques des différentes topologies qui les rendent plus ou moins adaptés à une utilisation dans un système efficace et évolutif.

Il vaut la peine de prendre un moment pour expliquer ce que nous entendons par *efficace* et *évolutif*. Nous appliquons ces termes à deux concepts différents: une conférence individuelle et un service dans son ensemble. Une conférence est évolutive lorsqu'elle peut prendre en charge un grand nombre de points de terminaison. Il est efficace lorsqu'il fournit une expérience donnée aux utilisateurs en utilisant le moins de ressources possible. Un service est évolutif lorsqu'il peut prendre en charge un grand nombre de conférences. Dans certains cas, c'est facile pour rendre un service évolutif, simplement en ajoutant plus de serveurs. Un service est efficace lorsqu'il prend en charge un nombre donné de conférences et d'utilisateurs tout en utilisant le moins de ressources possible. Nous voulons être aussi efficace que possible en considérant le coût global du service, le réseau et l'informatique ressources des clients et des dispositifs de service.

Voir les détails au chapitre 3.

1.5.1 Réseaux Full-mesh

Si nous avons une solution pour la communication audio et vidéo individuelle, une façon de réaliser une conférence multipoint est de connecter chaque paire de points de terminaison directement dans une topologie full-mesh [46]. Ceci est illustré sur la figure 3.2.a.

Voir les détails dans la section 3.1.2.

1.5.2 Unités de contrôle multipoint

La deuxième topologie que nous examinons utilise un serveur central qui mélange le contenu sur le niveau des médias. Ces serveurs sont connus sous le nom d'unités de contrôle multipoint (Multipoint Control Units, MCU [85]). Ceci est illustré sur la figure 3.2.b.

Les MCU interrompent complètement les connexions multimédias avec les points de terminaison. Ils décodent le flux reçus de tous les points d'extrémité, effectuer le mixage audio et la composition vidéo. Pour l'audio, ils créer un mixage séparé pour chaque point de terminaison afin d'exclure l'audio venant du point de terminaison lui-

même. De même pour la vidéo, ils composent et encodent un flux pour chaque point de terminaison.

Voir les détails dans la section 3.1.3.

1.5.3 Unités de routage sélectives

La troisième topologie dont nous discutons utilise également un serveur central avec une topologie en étoile, mais elle utilise routage au lieu de mixage / compositing. La figure 3.2.c l'illustre. L'idée importante qui permet à ces systèmes de fonctionner efficacement est celle de routage *sélectif* [20], ce qui signifie qu'au lieu de transmettre tous les paquets d'un flux vidéo, le serveur peut diviser un flux codé et produire un ensemble de flux avec des caractéristiques différentes *sans avoir besoin de ré-encoder*. Il transmet ensuite uniquement un sous-ensemble sélectionné de ces flux vers un récepteur. Nous utilisons les termes Unité de routage sélective (Selective Forwarding Unit, SFU) ou Selective Forwarding Middlebox [85] pour faire référence à ce type de serveur.

Cette architecture est particulièrement avantageuse. Sans décodage, et donc sans tampon de gigue requis, la latence de bout en bout est faible. Le traitement requis par le serveur est minimal et n'implique aucune manipulation au niveau du support. Cette efficacité accrue des serveurs en ce qui concerne l'utilisation du processeur leur permet de bien évoluer.

Voir les détails dans la section 3.1.4.

1.5.4 Autres topologies

En plus des trois topologies discutées jusqu'à présent, il existe de nombreuses autres qui ont été proposés ou utilisés. Certains modèles hybrides transfèrent la vidéo tout en mixant l'audio. Dans d'autres modèles, certains points finaux reçoivent des flux transférés tandis que d'autres recevoir des flux mixtes / composites. Certains modèles sans serveur utilisent un système centralisé topologie dans laquelle un point final sert de MCU ou SFU (cela a été utilisé Jitsi et Skype [10]).

De plus, il est possible de basculer dynamiquement entre différentes topologies utiliser celui qui convient le mieux à la situation donnée. Dans le chapitre 6, nous présenter un système qui bascule entre un full-mesh et un SFU basé sur taille de la conférence (bien que nous n'utilisions un maillage complet que lorsqu'il y a deux participants).

1.6 WebRTC

Web Real-Time Communications (WebRTC) est un effort pour apporter une haute qualité, multiplateformes, interopérables, capacités audio, vidéo et données en temps réel d'égal à égal sur le Web navigateurs sans l'utilisation de plugins. Il s'agit d'un vaste effort interorganisationnel qui a commencé en 2012 au sein de l'Internet Engineering Task Force (IETF). Il existe deux groupes de normes WebRTC: un ensemble des API JavaScript implémentées par les navigateurs Web et disponibles pour les applications Web, et un ensemble de protocoles réseau couvrant l'établissement de la connexion, les médias transport, sécurité, formats des médias, contrôle de la congestion et autres aspects. Ensemble, ils servir de base aux développeurs pour créer des applications Web riches et interactives avec audio et vidéo.

Voir les détails dans la section 3.2.

1.7 Jitsi Meet

Le projet Jitsi [4](à l'époque connu sous le nom de SIP Communicator) a été lancé en 2003 par Emil Ivov lors de ses études à l'Université Louis Pasteur à Strasbourg [35]. Au départ uniquement un client SIP, il a évolué au fil des ans en un large éventail de composants logiciels pour la communication en temps réel. Les projets sous Jitsi[5] sont des logiciels open source sous licence avec Apache 2 Licence.

En tant que projet open source, des centaines de personnes à travers le monde ont contribué à Jitsi au fil des ans. Cependant, il existe une équipe de développement de base qui maintient toutes les ressources et stimule le développement. En 2015-2018, cette équipe a été employée et soutenu par Atlassian Plc, et depuis fin 2018 il fait partie de la société 8x8; l'auteur est employé à plein temps ingénieur logiciel dans cette équipe.

Voir les détails dans la section 2 3.3.

1.8 Last N

Nous décrivons ici la première étape pour réduire la bande passante requise pour une conférence, en transférant la vidéo à partir d'un sous-ensemble sélectionné de points

[4]Voir la page d'accueil sur https://jitsi.org

[5]Les projets sont désormais hébergés sur github: https://github.com/jitsi

de terminaison. Nous proposons un moyen de rendre cette sélection automatique, en fonction de l'activité du locuteur, tout en gardant le contenu sélectionné pertinent pour les utilisateurs. Nous présentons un algorithme efficace pour effectuer identification du locuteur dominant sans avoir à décoder le flux audio le rendant approprié pour une utilisation dans les SFU. Nous détaillons la mise en œuvre de l'approche proposée dans la SFU Jitsi Videobridge. Enfin, nous effectuons des expériences pour valider l'approche et quantifier les améliorations en termes de ressources réseau et d'utilisation du processeur. Ce travail a été à l'origine publié dans [Pub5].

1.8.1 Sélection du endpoint en fonction de l'activité audio

Conférences de presse déployées dans une topologie pleine étoile ne sont pas bien à l'échelle lorsque le nombre de participants est substantiel, car chaque point de terminaison connecté au serveur de conférence envoie et reçoit des flux de chaque point d'extrémité.

Nous avons identifié trois problèmes pour les points finaux récepteurs. Tout d'abord, les exigences du réseau croître proportionnellement au nombre de participants. Deuxièmement, les besoins en CPU augmentent de la même façon parce que le point de terminaison doit déchiffrer, décoder, mettre à l'échelle et rendre tous les flux séparément. Enfin, l'interface utilisateur ne peut pas afficher efficacement un grand nombre de flux vidéo, en raison de écran immobilier insuffisant.

Le serveur de conférence rencontre également des problèmes d'utilisation du processeur et du réseau. Cependant, cette situation est problématique, car l'utilisation des ressources augmente quadratique avec le nombre de points de terminaison. En effet, les flux de chaque nouveau point de terminaison doit être chiffré et transmis indépendamment à chacun autre point final. Notre évaluation dans la section **??** montre que, en fonction du matériel et des ressources réseau disponibles, l'un ou l'autre (CPU ou réseau) peut être le goulot d'étranglement.

Ici, nous proposons LastN: un schéma général définissant une politique pour le point final sélection basée sur l'activité audio. Il est utilisé par une unité d'expédition sélective de choisir uniquement un sous-ensemble de flux à transmettre à tout moment compte tenu du temps imparti, atténuant les limitations précédemment identifiées des serveur de conférence.

Voir les détails au chapitre 4.

1.8.2 Évaluation des performances

Dans cette section, nous présentons les résultats expérimentaux. Nous avons effectué des expériences contrôlées mesurer l'utilisation des ressources sur les clients et les serveurs. Voir une description détaillée de les expériences et les résultats de la section 4.5.

Dans ce chapitre, nous proposons un schéma (LastN) qui utilise une SFU pour sélectionner flux vidéo à transmettre et à supprimer. Nous présentons une implémentation et l'évaluer en termes d'utilisation du réseau et de ressources informatiques. Nous examinons les effets de la variation du nombre de flux vidéo transmis (N), et la possibilité de désactiver temporairement les flux inutilisés. Nos résultats montrent que les gains de performance attendus sont réalisables dans la pratique. On observe notamment une baisse d'un quadratique à un linéaire (par rapport au nombre total de points de terminaison) croissance du débit binaire sortant utilisé par la SFU. De plus, pour un nombre fixe de points d'extrémité, peaufiner les valeurs de N entraîne une modifications du débit binaire, ce qui rend le système adaptable à différentes situations. Sur la base des résultats, nous nous attendons à ce que LastN soit utile dans la pratique dans deux scénarios: permettre aux petites et moyennes conférences de fonctionner plus efficacement (permettant ainsi à un seul serveur de gérer plus de conférences) et nombre maximal de points d'extrémité dans une conférence, avec les mêmes ressources.

LastN est assez général et permet différentes modifications et améliorations pardessus. En particulier, une modification intéressante, qui nous avons l'intention d'étudier, maintient différentes valeurs de N pour chaque récepteur, tout en conservant la liste L globale pour la conférence. Dans le chapitre 7, nous examinons un autre cas d'utilisation pour LastN: adaptation dynamique de la configuration du serveur à la charge expérimentée.

1.9 Simulcast

Le schéma LastN décrit dans le chapitre précédent réduit considérablement la bande passante requise pour une conférence, mais elle ne résout pas tous les problèmes. Utilisée seule, cette solution présente quelques inconvénients. Tout d'abord, il supprime complètement certains flux, bien que la bande passante disponible soit suffisante pour un flux de qualité inférieure qui bénéficierait toujours à l'utilisateur. Deuxièmement, il est inefficace dans de nombreux cas d'utilisation, car il ne transmet que des flux en pleine résolution, que le client peut réduire en raison de contraintes d'interface utilisateur. Troisièmement, parce qu'il utilise uniquement un débit binaire élevé à haute résolution

flux, il y a peu de flexibilité dans la capacité de la SFU à s'adapter aux changements de la bande passante disponible - des flux entiers doivent être supprimés ou repris.

Les systèmes de conférence utilisent généralement des fenêtres de tailles différentes pour différents points de terminaison, comme le montre la figure 3.4. En utilisant simplement LastN n'est pas optimal dans de tels cas, car le même résultat peut être obtenu avec moins de ressources si les flux affichés dans une petite fenêtre étaient réduit avant de les encoder. En réduisant les flux affichés dans une petite fenêtre nous pouvons atteindre un débit binaire global plus faible; ou de manière équivalente, nous pouvons atteindre le même débit binaire global avec une plus grande valeur de N, ce qui signifie que plus de flux peuvent être affichés.

Mise à l'échelle des flux envoyés sur le réseau pour correspondre aux fenêtres les affichant n'est pas simple pour plusieurs raisons. Premièrement, les tailles des fenêtres peuvent varier d'un point de terminaison au point final. Les points de terminaison peuvent avoir des affichages de taille variable, ils peuvent avoir sélectionné différents flux à afficher dans leur grande fenêtre d'affichage (notamment si tous les points d'extrémité affichent l'actuel haut-parleur, le haut-parleur actif montrera probablement un autre point final), ou ils peuvent utiliser une disposition d'interface utilisateur complètement différente. Ces situations sont particulièrement se produire lorsque la conférence a un mélange de participants utilisant des appareils mobiles et de bureau. Deuxièmement, la configuration peut changer à tout moment, par exemple lorsque les enceintes changent. S'adapter à cela nécessite la SFU pour contrôler le débit de codage des émetteurs, ce qui est complexe et entraîne un retard. De plus, il ne ne tient pas compte des exigences différentes de chaque récepteur.

Une façon de résoudre ces problèmes consiste à utiliser la diffusion simultanée. La diffusion simultanée consiste du codage et de la transmission simultanés du flux vidéo de chaque point d'extrémité en utilisant différentes résolutions et débits. Ce mécanisme permet à une SFU de dynamiquement modifier le débit binaire de sortie, en passant à un codage différent du flux, en réponse aux changements dans les conditions du réseau, et qui permet un contrôle plus souple de la congestion.

Dans ce chapitre, nous effectuons une analyse et une évaluation expérimentale de l'utilisation de la diffusion simultanée dans un système de conférence basé sur WebRTC. Nous évaluons l'utilisation des ressources des périphériques d'envoi et de réception et des composants du serveur, ainsi que la qualité d'image de bout en bout. Nous explorons également le problème du changement de qualité d'image de l'utilisateur expériences lorsqu'un basculement entre différents encodages se produit, et proposer un méthode de réduction du délai avant le basculement.

Voir tous les détails au chapitre 5.

1.10 Une approche hybride: Selective Forwarding Unit et Full-mesh

L'utilisation de LastN et de la diffusion simultanée réduit considérablement la bande passante requise pour un conférence dans un système basé sur SFU. Cependant, il existe deux caractéristiques pour lequel le modèle à maillage complet surpasse toujours le modèle SFU. Le premier est le latence, qui est plus faible car les points d'extrémité peuvent se connecter directement. Le second est le trafic réseau pour les serveurs, car ils ne sont pas toujours utilisés.

De loin, le paramètre le plus important affectant l'utilisation des ressources est de la taille d'une conférence. Les grandes conférences ne sont pas possibles avec un maillage complet, bien que les petits soient et puissent fonctionner mieux qu'un SFU. Donc, un service peut bénéficier si nous choisissons une architecture basée sur la taille de la conférence. La manière la plus simple d'y parvenir est de créer la conférence avec les configuration depuis le début. Cependant, dans la pratique, la taille de la conférence est rarement connue à l'avance. De plus, la taille de la conférence peut changer au fil du temps, donc la meilleure option peut impliquer l'utilisation de différentes architectures au cours différentes parties d'une conférence. Pour ces raisons, nous avons choisi de l'architecture de manière dynamique lors d'une conférence. Plus précisément, nous utilisons un maillage complet architecture pour les conférences de taille P ou moins et passer à une SFU lorsque le la taille atteint $P + 1$. Inversement, nous revenons au maillage complet lorsqu'un point de terminaison feuilles et la taille tombe à P.

Nous avons ajouté la prise en charge du passage à un maillage complet dans le système Jitsi Meet. Parce qu'il ne prend en charge que $P = 2$, nous appelons cette fonctionnalité " peer-to-peer for one-to-one " pour plus de concision, nous utilisons " P2P " au lieu de " peer-to-peer ". La mise en œuvre est détaillée dans la section 6.1. Nous avons ensuite activé la fonctionnalité dans le service *meet.jit.si* et comparé ses performances au mode SFU uniquement en termes de qualité de service pour les utilisateurs et de trafic serveur. Nous présentons les résultats à la section **??**..

Pour ce faire, nous commençons par une connexion SFU et la maintenons à tout moment. nous passer à la connexion P2P de manière opportuniste quand nous savons qu'il y a exactement deux points d'extrémité dans la conférence, et qu'il existe une connexion P2P entre eux. La figure **??** illustre cela. Le démarrage de la conférence avec la connexion SFU nous permet également pour minimiser le temps de démarrage, car dans la majorité des cas la connexion est établi en utilisant des candidats locaux ICE et n'a pas à attendre la collecte des candidats STUN ou TURN.

Pour quantifier les effets du mode P2P, nous effectuons des expériences et des mesures sur le service emph meet.jit.si.

Nos résultats quantifient une réduction significative du coût total de l'infrastructure. De plus, en moyenne, les appels directs ont une latence nettement inférieure L'efficacité du débit est gravement dégradée lorsque l'appel est acheminé via une SFU en raison de l'utilisation de fonctionnalités spécifiques au mode groupe telles que comme diffusion simultanée et terminaison RTCP. Grâce à l'utilisation de l'algorithme GCC, le la perte de paquets est négligeable quelle que soit la topologie. L'établissement d'appel le temps reste pratiquement le même pour les appels directs et relayés par le serveur.

Voir le chapitre 6 pour plus de détails.

1.11 Optimisations des performances pour les SFUs

En ce qui concerne les serveurs de vidéoconférence, les unités de transfert sélectif sont devrait généralement offrir une meilleure efficacité de calcul que les microcontrôleurs car ils ne décodent ni n'encodent l'audio ou vidéo. Cependant, leurs bonnes performances ne doivent pas être prises pour acquises, car elles dépendent de la mise en œuvre logicielle spécifique.

Une SFU est un système logiciel complexe qui doit être conçu avec comme objectif la performance. De mauvaises performances peuvent entraîner une dégradation de l'utilisateur l'expérience, ainsi que l'inefficacité et les coûts d'exploitation des services plus élevés.

Dans ce chapitre, nous explorons les facteurs affectant les performances et proposons des moyens de les améliorer. Nous commençons par décrire la fonction d'un SFU, et le Jitsi Videobridge en particulier dans la section 7.1. Dans la section suivante (7.2, nous proposons un algorithme efficace pour transfert sélectif, et nous évaluons ses effets sur l'utilisation du processeur. Enfin, à l'article 7.3 nous explorons le sujet de la gestion de la mémoire, en particulier dans les systèmes avec collecte automatique des ordures. Nous proposons une technique pour réduire la tension sur le ramasse-miettes, développer une méthodologie pour évaluer ses effets, et effectuer des expériences pour trouver la configuration la plus appropriée à utiliser pour une SFU.

1.11.1 Optimisation des performances de calcul

Au chapitre 4, nous avons observé que, sans diffusion simultanée, l'utilisation du processeur est proportionnelle à débit binaire total. Cependant, avec l'ajout de la diffusion simultanée, il y a un potentiel accru pour le processeur utilisation de ne pas évoluer linéairement. Certains encodages sont reçus et traités (contribuant à l'utilisation du processeur) mais ne sont pas transférés (et ne contribuent pas au réseau bitrate).

S'il n'est pas possible d'éliminer complètement le traitement des encodages, nous avons constaté des améliorations significatives des performances lors de l'application de deux optimisations. Tout d'abord, si un codage n'est sélectionné par aucun point d'extrémité de réception, nous n'effectuons pas le déchiffrement de la charge utile (même si nous faisons toujours l'étape d'authentification). Nous pouvons le faire car les informations dans les en-têtes ne sont pas cryptées. La seconde est l'implémentation de l'algorithme de transfert sélectif (voir section 7.1.2) à la place de l'approche naïve du traitement des paquets.

Voir les détails dans la section 7.2.

1.11.2 Optimiser les performances de la mémoire

Les unités de transfert sélectif ne nécessitent pas une grande quantité de mémoire, mais leur modèle d'allocation de mémoire peut être problématique lorsqu'il est utilisé avec automatique collecte des ordures. Le débit d'allocation élevé entraîne une activation fréquente du GC, qui peut geler l'application pendant un temps significatif avec des algorithmes GC optimisés pour une faible surcharge du processeur, comme les algorithmes par défaut fournis avec le kit de développement Java.

Nous avons proposé d'utiliser un pool de mémoire distinct implémenté dans l'application Java pour réduire la tendre le collecteur d'ordures. Nous avons ensuite développé une méthodologie pour évaluer la performance des différentes configurations du garbage collector et du pool de mémoire. Nous avons effectué des expériences en utilisant l'algorithme GC par défaut dans JDK ainsi que CMS et G1, et quantifié leurs performances pour la Application Jitsi Videobridge.

Nous avons constaté que les meilleures performances sont obtenues en utilisant l'algorithme CMS ensemble avec notre pool de mémoire proposé. Il conduit à des performances acceptables, définies comme meilleures que notre Jitsi Videobridge version 1.0 utilisant la configuration GC par défaut. La plus longue pause d'arrêt du monde (la mesure la plus importante) est passé de 384 à 82 millisecondes. Ce compromis a été

possible avec une baisse de 11% de l'efficacité. Le pool de mémoire amélioré toutes les métriques dans tous les tests.

Voir les détails dans la section 7.3.

1.12 Conférences avec les SFUs en cascade

Jusqu'à présent, nous avons discuté des conférences qui utilisent une unité de transfert sélectif dans un topologie centralisée (une étoile). Dans ce chapitre, nous explorons la possibilité d'utiliser un ensemble de SFU interconnectés dans une conférence. Il y a deux motivations pour ça. Cela pourrait permettre aux conférences de s'adapter à une plus grande taille et d'améliorer la qualité de connexion des utilisateurs s'ils se connectent à un serveur à proximité. Nous nous concentrons sur deuxième objectif, mais nous considérons également le premier pour la conception du système.

Services cloud modernes tels qu'Amazon AWS[6] faciliter le déploiement d'un système comprenant des serveurs dans différentes zones géographiques et fournir des outils bien développés, comme Route53[7], pour acheminer les utilisateurs vers un serveur proche d'eux. Cependant, ils sont principalement conçus pour les les applications Web et le scénario dans lequel un consommateur accède à un certain ressource dans le cloud.

La vidéoconférence a le potentiel de bénéficier grandement de caractéristiques de géolocalisation, car il est de nature interactive, et il est très sensible aux conditions du réseau telles que la largeur de ban, la perte de paquets, le temps d'aller-retour (RTT) et variations de RTT (gigue). Cependant, ce cas n'est pas aussi simple que d'en connecter un consommateur au serveur le plus proche car il nécessite l'interconnexion de deux ou plus de points de terminaison et le problème de la sélection optimale du serveur les emplacements ne sont pas bien explorés.

L'utilisation de SFU en cascade a le potentiel d'améliorer l'expérience utilisateur, comme en témoigne par le RTT réduit à la SFU locale, et le RTT de bout en bout dans la région ouest des États-Unis. Cependant, la stratégie de sélection consistant à connecter chaque point de terminaison à une UFP dans sa région locale a pour effet secondaire non intentionnel de augmentation du délai global de point d'extrémité à point d'extrémité.

Nous proposons un modèle d'utilisation des unités de transfert sélectif en cascade et présentons un analyse et évaluation expérimentale. Voir le chapitre 8 pour les détails.

[6] `https://aws.amazon.com`
[7] `https://aws.amazon.com/route53/`

1.13 Conférences chiffrées de bout en bout avec les SFUs

Les institutions financières ont besoin d'une sécurité renforcée pour protéger leurs actifs. Historiquement, cela a été réalisé grâce à des canaux de communication internes ou des lignes physiques dédiées avec des partenaires de confiance. En interne, haut niveau le chiffrement, couplé à une gestion intelligente des clés de sécurité ajoute la dernière toucher à des systèmes aussi sécurisés que vous le pouvez.

Cette approche présente deux inconvénients. Établir des canaux de communication entre les institutions prend beaucoup de temps et d'argent. En outre, il est difficile, voire impossible, de tirer parti la puissance des infrastructures cloud et Fournisseurs de plate-forme en tant que service (PaaS), car il est inacceptable pour données pour quitter le domaine contrôlé par l'institution.

Dans ce chapitre, nous étudions la faisabilité de la mise en œuvre du Spécifications de double cryptage PERC ([38]) en conjonction avec WebRTC. Nous constatons que le système ne prend pas en charge la diffusion simultanée comme actuellement utilisé dans WebRTC et propose des modifications pour permettre la prise en charge de la diffusion simultanée.

Nous avons implémenté un système de preuve de concept en modifiant la SFU Jitsi Videobridge et le navigateur Chromium. Nous avons utilisé l'application Web Jitsi Meet non modifiée. Le système met en œuvre les propositions décrites dans la section ci-dessus et utilise pré-généré et partagé entre les clés de chiffrement de bout en bout des points d'extrémité. À ce stade, nous évaluons uniquement les caractéristiques de performance et la faisabilité du système " double ", et non l'échange et la vérification des clés aspects de tout système à part entière.

Nous avons effectué des mesures de l'utilisation du processeur et analysé les frais généraux encourus en raison de les extensions d'en-tête supplémentaires que nous proposons. Pour le cas d'utilisation prévu où le chiffrement E2E est nécessaire pour la conformité raisons, la bande passante supplémentaire est tout à fait acceptable. Voir le chapitre 9 pour les détails.

Chapter 2

Introduction

The concept of a remote communication device incorporating both image and sound originated back in the late 19th century, not long after the invention of the telephone[1]. But the technology to make it first possible and then practical took many decades to develop.

The first systems were built in the late 1920s and combined early television technology with a two-way telephone line [56]. Initially excessively bulky and impractical (a room full of equipment), with time came the reduction in size necessary for commercial viability.

Throughout the 1960s and 1970s, there were multiple telephone devices with video capability on the market, which worked with an analog signal over the telephone network. However, they failed to gain popularity or commercial success.

Advancements in video compression in the 1980s and 1990s led to significant progress. They made it possible to transport a digital signal over the existing computer networks [25]. This led to the development of telepresence systems, usually, a dedicated room (sometimes with a dedicated network) equipped with multiple displays, cameras, and microphones. While fidelity increased, the price was still very high (upwards of $100,000). Despite the high cost, these systems had finally found their place in the market: large companies with the needs for such rooms and the capital to purchase them.

With the rapid improvement of computing devices and networks during the 1990s and early 2000s, it became increasingly feasible to utilize consumer-grade devices and public internet for audio and video communication. Various technologies and standards,

[1] http://www.terramedia.co.uk/Chronomedia/years/Edison_Telephonoscope.htm

most notably SIP [66], were developed to facilitate telephone calls over the internet and to cross connect them to the public telephone network, and the term "Voice over the Internet Protocol", or VoIP was coined. While video capabilities were also available, they did not immediately gain popularity.

Applications for video communication online started to become more popular in the late 2000s, most notably with the rise of Skype. With the proliferation of powerful personal mobile devices circa 2008, the hardware required for audio and video communication has become available to a significant and increasing portion of the world's population. This uncovered a host of potential new applications.

Today in 2019, video communication tools are an integral part of the daily lives of millions of people, both personally and professionally. There are still many more millions with access to devices capable of video conferencing, that is potential users in an unexplored market. The technology has made tremendous progress, but it is still an actively evolving area, and there are many ways it can be improved. This great potential to affect a considerable number of people by advancing the technology is what ultimately makes studying video conferencing worthwhile.

2.1 Applications of Video Conferencing

Video conferencing is becoming ubiquitous. As the technology develops, the fidelity, reliability, cost and accessibility improve, leading to new applications. Some of the applications that video conferencing is currently used for are:

- *Personal communication.* People talking informally with their friends or family, sometime in remote locations. Popular applications such as Facebook Messenger and WhatsApp are often used this way.
- *Remote collaboration.* Geographically dispersed teams of professionals collaborating on the same project [84]. This is becoming more and more popular for software development teams, conducting scheduled or ad-hoc remote meetings.
- *Telemedicine.* Providing remote health care services such as consultation with physicians and other medical professionals [37], patient monitoring, diagnostics, and other. This application was envisioned a long time ago[2], but its unique reliability and privacy requirements make it very challenging to adopt in practice. It has the potential to transform the way medical care is provided [42].

[2]The American Telemedicine Association was founded in 1993.

- *Webinars.* Interactive online presentations, with active participation from attendees [90].
- *Remote interviewing.* Conducting job interviews remotely.
- *Personal productivity.* Services aiming to increase productivity by pairing participants in video sessions[3].
- *Customer service.* Providing customer service and support remotely.
- *Consultations with professionals.* Conducting consultations with experts in various fields remotely.
- *Remote education.* Providing teaching and training remotely. Students can participate in an online classroom with a teacher [48].
- *Business meetings.* Scheduled company meetings between different offices. This is increasingly used for both company private "all hands" meetings, as well as public "earnings call" type meetings.

2.2 Challenges and Motivations

Today, people use a wide variety of devices capable of video conferencing. This includes personal desktop and laptop computers, mobile phones and tablets, smart television devices, and dedicated conferencing rooms. These devices have very different characteristics such as computing power, hardware codec support, cameras, microphones, and display sizes. They are also connected to diverse networks ranging from high-bandwidth and low-latency Ethernet links, to limited and unstable cellular networks or even direct satellite links. This presents a challenge for applications that have to use the available resources efficiently while also adapting to changing conditions.

Real-time video transmission is widely known to require considerable computation and bandwidth. this is even more so for conferencing when it involves more than two endpoints. Traditional video conferencing systems use a lot of resources, either on their endpoint devices or offloaded to remote servers.

Developing more efficient architectures has a two-fold effect. It serves users better because it improves the service quality in non-ideal networks, widening the potential pool of consumers. It is also beneficial for service owners as it lowers the service operating cost.

[3] https://www.focusmate.com/

Recent developments in the space of video communication led to the introduction of architectures based on Selective Forwarding Units (SFUs [20]). These systems have the potential to significantly improve efficiency and offer new functionality, but they present challenges of their own, requiring new approaches and solutions. This thesis is dedicated to the study of SFU systems, with a focus on making them more efficient, improving the quality of service, and lowering the costs.

In the following section we outline the contributions of this thesis, before proceeding with a review of the state of the art in the next chapter.

2.3 Contributions

This section outlines the contributions of the thesis and the research it is based on. To keep the description concise, we assume familiarity with some of the concepts described later in the text. Readers unfamiliar with the terminology can find an introduction to the related subjects in Chapter 3.

The main goal of this dissertation is to increase the efficiency of video conferencing systems while also improving the overall user experience. We begin by proposing an efficient algorithm that uses speech activity to select a subset of endpoints in a conference. Forwarding video only for endpoints in this set significantly reduces the required bandwidth in a conference, making it more efficient. Next, we present a thorough evaluation of the use of simulcast for video conferencing, which further reduces the required bandwidth. Then, we propose, analyse, and evaluate a hybrid of a peer-to-peer and server-based architecture, which improves user experience and reduces service cost.

We also explore server performance in terms of CPU usage and memory management. We propose memory management optimizations aiming to reduce the strain on the garbage collector. We develop a methodology to measure the effects of application pauses due to garbage collection, and evaluate the proposed solution in different configurations. The results are positive and uncover areas for further research.

Next, we propose a distributed server approach with the goals of improving user connection quality by using servers in close geographical proximity to each endpoint. We present an initial analysis, an implementation, and a randomized experiment measuring connection quality. The results show that, with regard to connection quality, the approach offers improvements in some cases, but overall the effects are negative. We propose further research to understand how to apply the technique optimally.

Another goal of the thesis is to explore how systems based on Selective Forwarding Units can be modified to support features available in other architectures. We propose an extension of the existing specifications for end-to-end encrypted conferences, enabling the use of simulcast and certain stream repair mechanisms. We implement a proof-of-concept system and show acceptable overhead.

This thesis is in part based on previous research by the author, published in several peer-reviewed international journals and conferences. A full list of publications is available in Appendix A.

2.4 Thesis Structure

This manuscript is organized into nine chapters. This first chapter serves as a short introduction. Chapter 3 presents the current practices and state of the art in video conferencing, and introduces the technology stack used throughout the thesis.

Beginning with Chapter 4, each chapter presents one of the contributions of this thesis.

Chapters 4, 5, and 6 examine three different techniques for lowering the bandwidth requirements of a conference. Chapter 4 introduces the LastN scheme for selecting a set of endpoints based on audio level indications. Chapter 5 details the use of simulcast for video conferences and presents a thorough experimental evaluation. Chapter 6 proposes a system that dynamically switches between a peer-to-peer and server-based architecture.

Chapter 7 examines the performance of Selective Forwarding Unit implementations in terms of CPU usage and garbage collector pauses.

Chapter 8 introduces the idea of using a set of geographically distributed servers in order to improve endpoints' connection quality.

Chapter 9 proposes extensions to a system for end-to-end encrypted conferences in order to support simulcast and to improve the mechanisms for stream repair.

Chapter 10 concludes this thesis by presenting concluding remarks and discussing some perspectives.

The thesis also includes two appendices. Appendix A lists the author's publications, which are the basis of this thesis. Appendix B details several patents resulting from the author's research.

Chapter 3

Video Conferencing Architectures: State of the Art

In this thesis, we are interested in a specific type of system for video conferencing — those which allow a group of people to communicate over the internet in real- time. We use the terms "video conference" or "conference" interchangeably to refer to this specific type of conference. Therefore, systems that only allow for two endpoints, or systems for live-streaming which are unsuitable for real-time communication are not of immediate interest. More specifically, we are interested in video conferences with a particular architecture — those using a Selective Forwarding Unit (SFU). In the next section we introduce the different topologies that have been used for multiparty conferencing, including SFUs, and compare their characteristics.

3.1 Video Conference Topologies

Video conferencing systems can be categorized by the way their endpoints connect and the type of media processing performed by server components (if any). In this section we discuss the most commonly used architectures, particularly those based on Selective Forwarding Units, which is the focus of this thesis. We take a comparative approach and examine the characteristics of the different topologies that make them more or less suitable for use in an efficient and scalable system.

It is worth taking a moment to explain what we mean by *efficient* and *scalable*. We apply those terms to two different concepts: an individual conference and a service as a whole. A conference is scalable when it can support a large number of endpoints.

It is efficient when it provides a given experience to the users using as little resources as possible. A service is scalable when it can support a large number of conferences. In some cases it is easy to make a service scalable, simply by adding more servers. A service is efficient when it supports a given number of conferences and users while using as little resources as possible. We aim to be as efficient as possible by considering the overall cost of the service, and the network and computing resources of both client and service devices.

When comparing different topologies we consider the following:

- The latency, by which we mean the end-to-end delay that the people experience when they use the system. Latency is critically important; if it is too high users experience problems communicating. For conversational speech International Telecommunication Union recommends not to exceed a one-way delay of 400ms, however some users are dissatisfied with a one-way delay of 300ms [32].
- The amount of resources required for a conference. There are *computing resources* and *network resources* and we look at them separately for servers and for client endpoints.
- The flexibility of receivers to customize the user interface (UI). For example controlling the volume of individual audio streams, rendering interactive UI elements on top of individual video streams such as audio level indications, tooltips with additional information, etc. Also, the ability to control which remote participants are visible as well as the size of their viewports. For brevity, we refer to this concept as *UI flexibility*.
- The possibility of end-to-end encrypted conferences. Encrypting all multimedia in such a way that only the participating endpoints, but not any intermediary servers, can decrypt it is another topic of interest of this thesis. Chapter 9 explores this topic.

In the next section, we introduce the way real-time video communication works in the point-to-point case, which is a building block for the multipoint (or conference) case. In the following sections, we introduce the main topologies used for conferencing and discuss their characteristics.

3.1.1 One-to-one Communication

In a system for two-way communication between two endpoints, both endpoints perform simultaneously the functions of a receiver (receiving and interpreting a signal from

the remote endpoint) and a sender (sending a signal to the remote endpoint). However, it is often useful to consider the two roles separately, because this reduces the complexity. Here we consider the one-way communication between a sending endpoint and a receiving endpoint, with the understanding that the complete system also uses the same ideas with the roles reversed to allow for two-way communication. We utilize this convention throughout the thesis without explicitly stating so.

Virtually all modern solutions for audio and/or video communication use the following components [24] : a media source, an encoder, a network layer, a jitter buffer, a decoder, and a renderer (see figure 3.1).

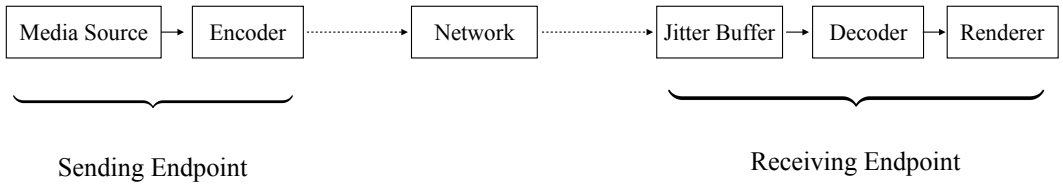


Figure 3.1: *The components of an audio or video real-time communication pipeline.*

The media source is usually a camera or a microphone. It produces the digital input signal for the encoder in the form of a stream of raw (uncompressed) frames [1]. This can also be a virtual device, combining input from other sources, or applying filters or other transformations. For our purposes we consider any pre-encoding transformations, for example de-noise filters and rescaling, to be part of the media source.

The stream of packets coming from the source has a very high bitrate[2], making it unsuitable for transfer over a network uncompressed. This is where the encoder comes in.

[1] For video, a frame constitutes a single image, while for audio a frame is a group of consecutive samples with a given duration.

[2] A typical video stream with 1280x720 pixels at 30 frames per second in YUV420 format takes 166Mbps, while a typical audio stream with a sample rate of $48\,\mathrm{kHz}$ in PCM format with 16-bit samples takes $768\,\mathrm{Kbps}$.

The encoder takes as input a stream of raw frames and produces a stream of compressed frames. There are many different encoding standards, such as the VP8 [9] and H264 [34] codecs[3] for video, and Opus [80] and G711 [33] for audio.

In general there are two types of codecs – lossless and lossy. With lossless compression, the decoder can reconstruct an exact copy of the original signal. In this case there is a trade-off between two variables – the encoding complexity (i.e., how much computation the encoding process takes), and the compression rate (i.e., the ratio of the uncompressed size to the compressed size). This trade-off varies between different codecs, and most encoders can be configured with different complexity settings, but in general more computation yields a higher compression rate.

With lossy compression the decoder is not able to reconstruct the original signal, but only an approximation of it. This allows audio and video codecs to achieve a much higher compression ratio at a cost of increased error (the difference between the original and reconstructed signal), adding a third variable to the trade-off. Lossy audio and video codecs are designed to maximize the perceptual quality (that is, how a human observer perceives the difference), at given complexity and compression rate. Most widely used codecs can be configured with different complexity settings and a specific target bitrate for the stream which effectively controls the compression rate. Lossy codecs can achieve *perceptually lossless* compression at much lower bitrate than truly lossless codecs [87].

Real-time communication systems use lossy codecs[4], because they generally produce a lower bitrate and the target bitrate can be re-configured dynamically during a session.

The process of encoding tends to be very CPU intensive, especially for video. This remains true even with the extremely powerful processors of today. This is due in part to ever increasing resolutions and frame rates, and because modern video codecs are specifically designed to take advantage of the available computing resources to achieve higher perceived quality and compression ratio [13].

The network layer transports the encoded stream from the sending endpoint to the receiving endpoint. It has all of the usual limitations of computer networks: limited bandwidth, latency, packet-to-packet jitter and packet loss. Any of these conditions may change at any time during a session due to changes in the load of the network or in the routing configuration.

[3]The word codec is a portmanteau of coder-decoder.

[4]There are rare exceptions [22].

Reliable transport protocols such as TCP [59] are sometimes used, though not preferred, because unless the round-trip time (RTT) is very low the latency they introduce hinders real-time communication [14]. State-of-the-art systems use protocols on top of UDP [58] and specialized algorithms for stream repair based on explicit retransmission requests for some packets and forward error correction [30]. For congestion control they use different mechanisms to produce an estimation of the available bandwidth, which is then used to configure an appropriate target bitrate for the encoder.

The jitter buffer is used to deal with the packet-to-packet jitter (i.e. difference in transit time of individual packets) introduced in the network layer in order to ensure smooth playout. We consider the jitter buffer a separate component instead of part of the network layer because it is closely associated with the decoding process and only needed when decoding is performed. Some jitter buffers use knowledge about the specific codec to decide whether to wait until a missing frame is complete (all packets are received), drop a missing frame because it is not strictly needed, or indicate to the decoder that Packet Loss Concealment (PLC) needs to be used [89].

To accommodate packets which were lost in the network, then reported lost by the receiver, and then retransmitted by the sender, the jitter buffer may need to be longer than one RTT. In order to not always incur this extra latency, state-of-the-art systems use adaptive jitter buffers, which change their size according to network conditions [24]. Still, the jitter buffer always adds some extra delay [7].

The decoder essentially undoes what the encoder does. It takes as input a stream of compressed frames and produces a stream of raw frames. With lossy codecs (and because the jitter buffer may drop some of the encoded frames), the decoded stream is not exactly the same as the pre-encoded stream.

The process of decoding is usually much faster that that of encoding [13], but depending on the receiving endpoint's characteristics it may be significant. For instance, even if the endpoint's processor is fast enough to decode the stream, doing so may lead to higher energy use which is undesirable especially for devices running on battery power.

The renderer consumes the raw stream. This may mean displaying video on a screen or part of a screen, or playing audio through a speaker.

Each of these components has some characteristics which are important for the design of a conference system. In summary, the most important things to consider are:

- The encoding process is costly and we need to make sure that encoding devices are not overloaded.
- The network has limited and changing bandwidth, and the system needs to adapt to it and use it efficiently.
- All components, particularly the network and jitter buffer, add to the end-to-end latency. We should always strive to minimize latency.

Now that we've covered the basics of how one-to-one communication works, in the next section we introduce the first of the multi-point topologies: the full-mesh.

3.1.2 Full-mesh Networks

If we have a solution for one-to-one audio and video communication, one way to achieve a multi-point conference is to connect each pair of endpoints directly in a full-mesh topology [46]. This is illustrated in figure 3.2.a.

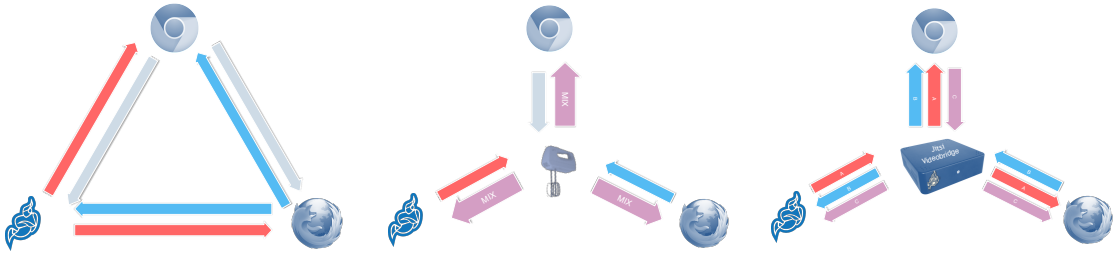
This model has some important advantages. First, since there are no intermediary servers between the endpoints, the end-to-end latency is minimized. Second, since no servers are necessary for handling multimedia in the conference[5], this makes it both easier to develop such a solution and cheaper to deploy it in production.

Since each receiver receives individual streams from the remote endpoints, it is free to customize the user interface. Also, since the multimedia streams do not pass through a server, end-to-end encryption is relatively easy to achieve [3].

However, there are some considerable disadvantages of this model when it comes to scale. First, if a sending endpoint is to use the available bandwidth efficiently it has to encode its stream multiple times, which is extremely CPU-intensive. Second, it has to send its stream to multiple other endpoints. This requires a lot of bandwidth, which is often a very limited resource for end users on a network [17, 2]. Third, even with sufficient bandwidth, competing flows to multiple destinations makes congestion control harder [73] and produces inconsistent quality for the receivers [74].

[5]Of course, some servers are needed for signaling, that is to help the clients establish the connections with each other.

The effects of these problems increase dramatically with conference size. Small conferences of up to about 5 participants may perform well given proper hardware and good network conditions, even though the resource usage is far from optimal. This is one of the reasons some services use this model. However, for larger conferences the model just does not work[6].



(a) *The full-mesh topology connects endpoints directly without a central server.*

(b) *The Multipoint Control Unit topology uses a mixing server.*

(c) *The Selective Forwarding Unit topology uses a routing server.*

Figure 3.2: *A three-endpoint conference using the most common topologies: full-mesh, MCU, and SFU.*

3.1.3 Multipoint Control Units

The second topology we look at uses a central server which mixes content on the media level. Such servers are known as Multipoint Control Units (MCU [85]). This is illustrated in figure 3.2.b.

MCUs completely terminate the media connections with endpoints. They decode the streams received from all endpoints, perform audio mixing and video composing. For audio, they create a separate mix for each endpoint in order to exclude the audio coming from the endpoint itself. Similarly for video, they compose and encode a stream for each endpoint.

This model has the advantage of simplicity and efficiency for the endpoints. Sending endpoints only have to encode and send a single stream (technically one audio and one video, though we do not make this distinction every time) to one destination (the MCU). Receiving endpoints receive a single stream, and this is all they need to handle. In fact, endpoints need no more functionality than they need for one-to-one communication.

[6]There is a joke in the field that as the size increases the topology changes from full-mesh to a full mess.

Congestion control also functions like in the one-to-one case, because the server controls the encoder and can configure the desired bitrate.

However, what facilitates this efficiency for the endpoints is extra processing on the server, which has to perform multiple transcoding operations. This is extremely CPU intensive, and because of it MCUs typically run on powerful dedicated machines, and are able to host only a small number of conferences.

The simplicity for the receiving endpoints comes at the cost of flexibility. They receive a single pre-mixed audio stream, and a single pre-composed video stream. Therefore they are not able to control the volume per-participant, render interactive UI elements, or directly control the remote endpoints they wish to display. Video re-encoding can be reduced if the MCU produces a single encoded video stream, common for everyone in the conference, but this is a shortcoming from a usability perspective and prevents the above-mentioned use of congestion control per-endpoint.

Critically, the additional decoding process on the server necessitates the use of a jitter buffer, increasing latency.

Finally, end-to-end encryption is simply not possible with an MCU, because it has to decode all incoming streams.

Multipoint control units have been used with great success in online services such as BlueJeans[7] and Cisco's Webex[8]. There are also dedicated hardware MCU devices on the market, such as those offered by Avaya[9] and Polycom[10], as well as software solutions such as TrueConf[11] and IBM's Sametime MCU[12].

3.1.4 Selective Forwarding Units

The third topology we discuss also uses a central server with a star topology, but it uses routing instead of mixing/compositing. Figure 3.2.c illustrates it. The important idea which allows such systems to work efficiently is that of *selective* forwarding [20], which means that instead of forwarding all packets of a video stream, the server can split an encoded stream and produce a set of streams with different characteristics *without the need to re-encode*. It then forwards only a selected subset of these streams to a

[7] https://www.bluejeans.com/
[8] https://webex.com
[9] https://avaya.com
[10] https://polycom.com
[11] https://trueconf.com/
[12] https://www.ibm.com/support/knowledgecenter/en/SSKTXQ_9.0.0

receiver. We use the terms Selective Forwarding Unit (SFU) or Selective Forwarding Middlebox [85] to refer to this type of server.

This architecture is distinctively advantageous. With no need to decode, and therefore no jitter buffer required, end-to-end latency is low. The processing required by the server is minimal and does not involve manipulation on the media level. This increased efficiency of servers in regards to CPU usage allows them to scale well.

Endpoints receive individual streams, which allows them to use a personalized layout. However, this comes with a higher bitrate and an increased cost for decoding. There is also increased complexity for the clients, because they need to handle multiple streams.

Implementing end-to-end encryption is challenging, but it is possible. To achieve selective forwarding, the SFU needs to access, and in some cases modify, some of the stream's meta-information. Chapter 9 of this thesis discusses this problem and shows how advanced selectivity features can be implemented with end-to-end encrypted media.

One of the main disadvantages of SFUs is the increased demands they place on network resources. Without any selectivity, as the conference size increases, so too does the amount of bandwidth required (it grows as N^2). Some form of selectivity is essential, but it is also not trivial. This is one of the main topics in this thesis and we explore it in the following chapters.

Congestion control is also challenging with an SFU because it does not directly control the encoders and subsequently cannot control the encoding bitrate. When adapting to changes in available bandwidth, it must enable/disable entire streams. This does not allow for fine control over the bitrate; in fact, the bitrate of the forwarded streams may also change in time.

Selective Forwarding Units were introduced relatively recently [20], but they have gained popularity and are successfully being used in many popular products including Vydio[13], Google Hangouts and Google Meet[14], Facebook Messanger, and 8x8 Virtual Office[15]. Additionally there are multiple open-source SFU implementations (or frame-

[13]https://vidyo.com
[14]https://meet.google.com
[15]https://8x8.com

works which include an SFU) including Janus[16], Kurento[17], Mediasoup[18], Medooze[19] and Jitsi[20]. The author is employed by 8x8 and is one of the authors of the Jitsi Videobridge SFU.

3.1.5 Other Topologies

In addition to the three topologies discussed thus far, there are many more which have been proposed or used. Some hybrid models forward video while mixing audio. In other models, some endpoints receive forwarded streams while others receive mixed/composited streams. Some serverless models utilize a centralized topology in which one endpoint serves as an MCU or SFU (this has been used Jitsi and Skype [10]).

Additionally, it is possible to switch between different topologies dynamically using whichever is most appropriate for the given situation. In chapter 6, we present a system that switches between a full-mesh and an SFU based on conference size (though we only use a full-mesh when there are two participants).

Some models use multiple servers. For example, in a single conference, multiple SFUs can be used to decrease end-to-end latency. We examine one such system in chapter 8 of this thesis.

3.1.6 Summary

There are various topologies for multi-endpoint conferences. In the sections above, we examined the three most common. We compared them based on characteristics that impact the cost and quality of a conference service. Table 3.3 summarizes the results.

We can see that the SFU model does not have any of the unavoidable disadvantages of the MCU model including inherent latency, high cost of server processing, inflexibility of the UI, and inability to support end-to-end encryption. It also does not have the problem of high resource use and congestion control on the sending endpoints which full-mesh conferences have at scale. The disadvantages of the SFU model are in managing the network use on the server and endpoints, and the complexity of end-to-end encryption. While the SFU model is not without its challenges, if they are approached

[16]https://janus.conf.meetecho.com/
[17]https://kurento.org
[18]https://mediasoup.org
[19]https://github.com/medooze
[20]https://jitsi.org

	Latency	Server Computing Resources	Server Network Resources	Endpoint Resources	UI Flexibility	E2E Encryption
Full Mesh	Very Low	N/A	N/A	Very High	Good	Easier
MCU	High	Very High	Low	Low	Bad	Not possible
SFU	Low	Low	High	High	Good	Hard
Cascaded SFUs	Potentially Very Low	Low	High	High	Good	Hard

Figure 3.3: *A summary of the characteristics of the different topologies. SFUs and Cascaded SFUs have the potential to be very efficient if we can manage the network resources at the server and endpoints.*

and addressed sufficiently well, it has the potential to be superior in service cost and quality. For this reason, we have chosen to study the SFU model in detail in this thesis.

Now that we have introduced the SFU architecture, which is the central topic of this thesis, we move on to present the technology stack we have chosen to use: WebRTC.

3.2 WebRTC

Web Real-Time Communications (WebRTC) is an effort to bring high quality, cross-platform, interoperable, peer-to-peer real-time audio, video, and data capabilities to web browsers without the use of plugins. It is a large cross-organization effort which started in 2012 at the Internet Engineering Task Force (IETF). There are two groups of WebRTC standards: a set of JavaScript APIs implemented by web browsers and available to web applications, and a set of network protocols covering connection establishment, media transport, security, media formats, congestion control, and other aspects. Together they serve as the basis for developers to create rich, interactive web applications with audio and video.

There is also an open-source project[21], led by Google, which has become the de-facto reference implementation for the network protocols. To avoid confusion between the standards and the software, we refer to the software as *webrtc.org*.

The JavaScript APIs [12] are developed within the World Wide Web Consortium (W3C)[22]. They allow for media acquisition [15] (gaining access to the user's microphone, camera or the contents of their screen), establishing a secure connection with a remote peer (another browser or a non-browser endpoint), and sending and receiving audio, video, or application-defined data through the peer connection.

In order for two endpoints to connect, they first need to exchange some information about the session (such as available IP addresses). This is commonly referred to as "signaling", and it necessarily involves a server. The APIs describe this information using Session Description Protocol (SDP) [26, 54]. The standards do not specify how to signal the SDP block, and different web applications use different signaling protocols such as SIP [66], XMPP [68], or custom protocols.

More important for this thesis are the underlying network protocols WebRTC uses. Developed within the RTCWEB working group of the IETF[23], they specify that Interactive Connectivity Establishment (ICE [64]) is used to establish the connection, and that the Secure Real-time Transport Protocol (SRTP [70, 11]) is used to transport the encoded streams. There are mandatory to implement codecs for audio [79] (Opus [80, 75] and G.711 [33]), and video [86] (VP8 [9, 86], and the H.264 Constraint Baseline profile [34]), but additional codecs are optionally supported through a negotiation process [54].

After more than 7 years, the RTCWEB working group recently closed after completing its charter in August 2019[24]. At the time of this writing all unpublished drafts are in the editor's queue to be published as official Requests For Comments (RFCs). The specifications are relatively mature, and significant changes are unlikely.

When the video element was introduced in the HTML5 standard [31], streaming technology had already been developed and available through third-party plugins in a non-standards based way [73]. The standardization of the video element and its adoption by browsers facilitated the growth of media streaming, allowing it to become ubiquitous. WebRTC has the potential to do the same for real-time communication. Natively supported by most major web browsers [25], it is the base for immensely popular and suc-

[21] https://webrtc.org
[22] W3C: https://www.w3.org/
[23] https://tools.ietf.org/wg/rtcweb/
[24] https://mailarchive.ietf.org/arch/msg/rtcweb/4cj95edGFtfjZkUjozTybOJiMcA
[25] http://iswebrtcreadyyet.com/

cessful services such as Google Hangouts and Google Meet, Facebook Messenger and Discord [26]. In 2016 Google announced that the Chrome browser is installed on over two billion mobile and desktop devices[27], and since then Chrome's market share has continued to grow[28]. It is hard to quantifiably measure the use of the WebRTC technology as a whole, because no single entity controls it. However, we do know that millions of people already use it, and there are multiple billions of people with access to it via their web browser. That is to say, there are billions of potential users.

The WebRTC standards and implementations provide a modern, state-of-the-art stack for real-time communication. They allow for many different systems to be built on top of them, including systems for video conferencing. This feature, as well as the great potential to develop and improve systems that can be relatively quickly released to and utilized by such a vast number of users, is why we chose the WebRTC technology for this thesis. While most of our results are generic, in some cases we restrict our solutions to those that can be built with WebRTC and run on modern browsers, with most experiments performed on WebRTC systems. This allows us to stay practical and remain relevant.

In the next section, we limit our discussion to a specific video conferencing system.

3.3 Jitsi Meet

In this final section of the introduction, we present a specific system for video conferencing which is used for study and experimentation for the remainder of the thesis: Jitsi Meet.

The Jitsi project[29] (at the time known as SIP Communicator) was started in 2003 by Emil Ivov during his studies in Louis Pasteur University in Strasbourg [35]. Initially only a SIP client, it evolved over the years into a broad set of software components for real-time communication. The projects under Jitsi[30] are open-source software licensed with the Apache 2 license [1].

As an open-source project, hundreds of people around the world have contributed to Jitsi over the years. However, there is a core development team that maintains all resources and drives development. During 2015-2018 this team was employed and sup-

[26] https://bloggeek.me/massive-applications-using-webrtc/
[27] https://www.youtube.com/watch?v=eI3B6x0fw9s
[28] https://gs.statcounter.com/browser-market-share#monthly-201602-201909
[29] See the homepage at https://jitsi.org
[30] The projects are now hosted on github: https://github.com/jitsi

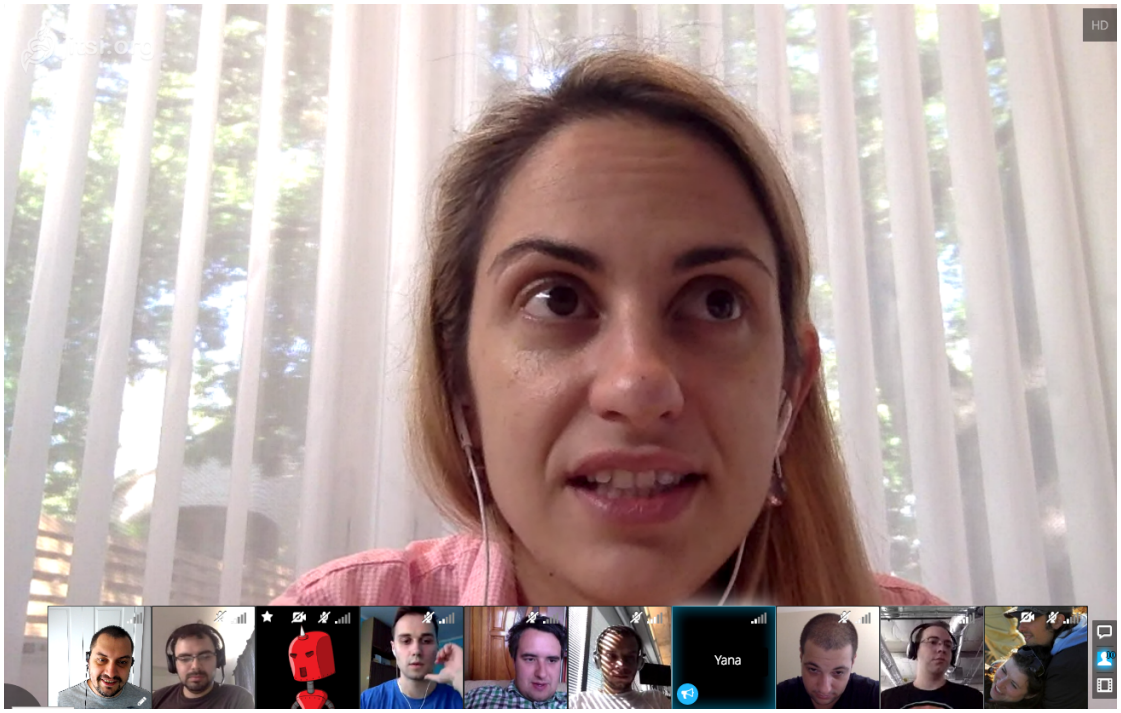


Figure 3.4: *A screen shot from a 10-participant conference in the Jitsi Meet application. Most of the space is occupied by the active speaker, while the remaining speakers are displayed in "thumbnail" size. Some speakers have disabled their video, so an avatar is displayed instead.*

ported by Atlassian Plc, and since the end of 2018 it is part of the 8x8 company; the author is employed as a full-time software engineer on this team.

Jitsi Meet is a conferencing system built for WebRTC. It consists of client applications for both web browsers and mobile devices, as well as various server components. Notably, it uses the Jitsi Videobridge[31] Selective Forwarding Unit, which is used in many of the experiments in later chapters. Figure 3.4 shows what a typical video conference looks like in the web browser interface.

The system is designed to be feature complete, but extensible and customizable, scalable, and interoperable with SIP and PSTN networks. It is in active development, but is already used in multiple commercial products, including 8x8's Virtual Office,

[31] See https://github.com/jitsi/jitsi-videobridge

the Rendez-Vous service by Renater[32] Rocket Chat, Matrix[33] and Symphony's video conferencing solutions[34].

In addition to maintaining the software, the 8x8 team runs a free to use video conferencing service that is open to the public. It is available on *https://meet.jit.si* and we refer to it as the *meet.jit.si* service. It requires no registration and supports both desktop browsers and mobile devices. It has been steadily gaining popularity, and at the time of this writing, hosts approximately 20000 conferences per week. It is separate from the company's commercial products, and one of the main reasons for its existence is to test new experimental features.

We have chosen to use this particular conferencing system for most of the experiments for a few reasons. First, as one of the principal authors of the Jitsi Videobridge SFU, the author is intricately familiar with its workings saving time when modifications are needed. Second, the project is open-source, which allows us to share our contributions with the community not only as research but as running code. Lastly, the access to the *meet.jit.si* service the author has through his position at 8x8 allows us to perform some of the experiments at scale in a production service. The service has proven an invaluable tool for research.

This concludes the introduction of the state of the art. The next section discusses the authors contributions and detail the structure of the remaining of the thesis.

[32]https://www.renater.fr/RENDEZ-VOUS
[33]https://matrix.org
[34]https://symphony.com

Chapter 4

Last N: Relevance-Based Selectivity

In this chapter, we describe the first step in reducing the required bandwidth for a conference, by forwarding video from a selected subset of endpoints. We propose a way to make this selection automatic, based on speaker activity, while keeping the selected content relevant for the users. We present an efficient algorithm for performing dominant speaker identification without the need to decode the audio streams making it appropriate for use in SFUs. We detail the implementation of the proposed approached in the Jitsi Videobridge SFU. Finally, we perform experiments to validate the approach and quantify the improvements in terms of network resource and CPU usage. This work was originally published in [Pub5].

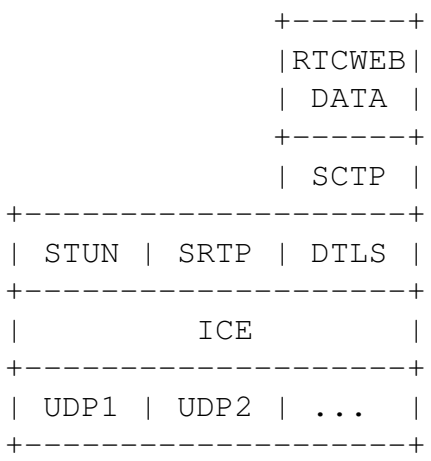
In the next section, we discuss the transport protocol used in WebRTC in more detail.

4.1 Real-time Transport Protocol

WebRTC uses Interactive Connectivity Establishment (ICE [64]) protocol to establish the connection between endpoints. This protocol is designed for Network Address Translation (NAT [76]) traversal. Each endpoint generates a list of local *candidates*, which represent a IP address and port which could possibly be used to reach the endpoint. The two endpoints exchange their candidates and generate a list of pairs of local and remote candidates. Then they systematically attempt to establish a connection using each pair. ICE uses STUN [65] in an attempt to connect the endpoints directly even in the presence of a router performing NAT. As a fallback it uses a relay server using TURN [50]. It supports both UDP and TCP (using additional framing [43], since ICE is packet-based) as the underlying transport, but UDP is preferred.

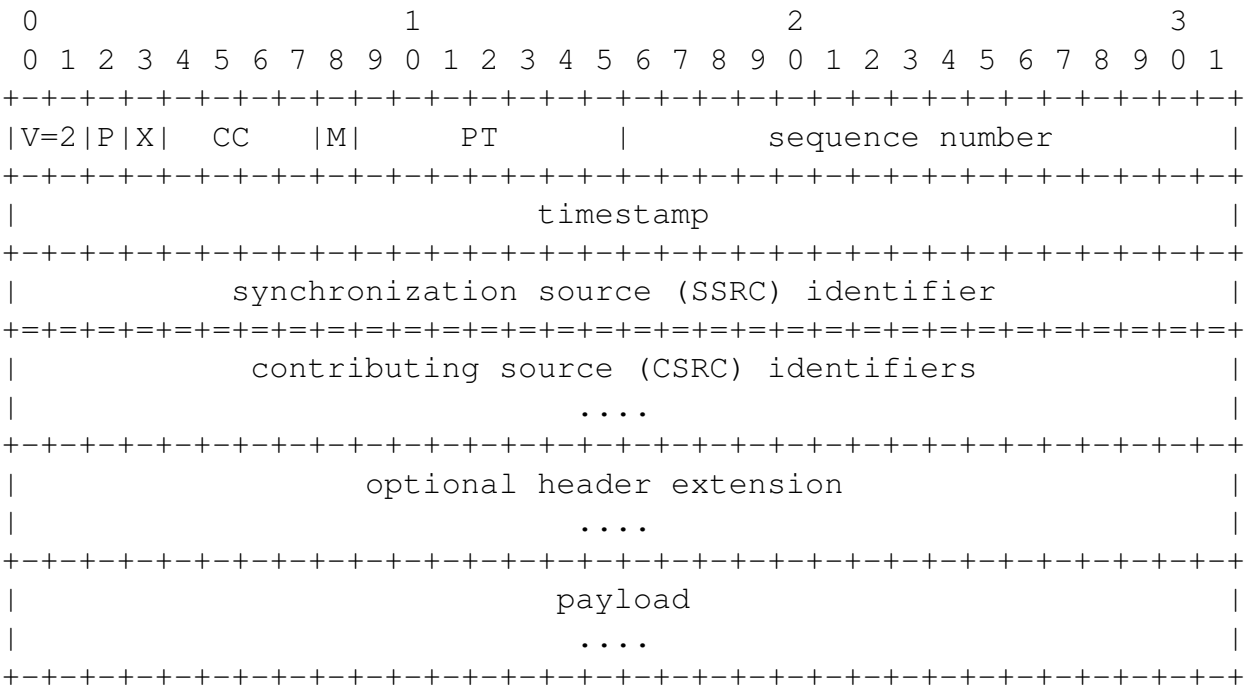
After a connection is established using ICE, the two endpoints start a Datagram Transport Layer Security (DTLS [61]) session. This secure session is used to exchange the security context needed for audio and video, as well as to transport the abstract data.

The actual audio and video is transported using Secure Real-time Transport Protocol (SRTP [27, 70]). The following diagram illustrates the protocol stack used in WebRTC (see [40]):



The Real-time Transport Protocol (RTP [70]) is primarily designed to satisfy the needs of multi-participant multimedia conferences. It is typically used on top of UDP and makes use of its multiplexing and checksum services. It does not guarantee delivery or prevent out-of-order delivery, nor does it assume that the underlying network is reliable and delivers packets in sequence.

It is designed to provide the common functionality required for any of the intended use cases, including audio and video conferencing, storage of continuous data, and interactive control and measurement applications. The structure of an RTP packet consists of a fixed-size header, followed by an optional extended header, followed by the payload:



The protocol allows for multiplexing of different streams via the synchronization source (SSRC) field. Support for different media formats is available via the Payload Type (PT) field. The Sequence Number (SEQ) and Timestamp (TS) fields enable the correct ordering and timely playback of received packets.

RTP is designed to be used in conjunction with an additional "profile" specification, which defines among other things the payload type mapping and the available header extensions. WebRTC uses the Extended Profile with Real-time Transport Control Protocol (RTCP)-Based Feedback (RTP/AVPF [57]). This profile is designed specifically for audio and video. It defines defines a set of constant PT numbers, and supports additional media formats negotiated per-session. It uses RTP's companion control protocol (RTCP [70]), whose primary function is to provide feedback on the quality of the data distribution. This is used for control of adaptive encodings, i.e., setting the desired encoding bitrate. Additionally, it allows for inter-stream synchronization for streams coming from the same source by providing a mapping from the timestamp space of each stream to a common source clock. This is typically used to achieve synchronous playback of audio and video (lipsync).

WebRTC also utilizes the Generic Mechanism for RTP Header Extensions [72]. This allows for a set of extensions to be negotiated for a particular session, and adds a set of optional variable length extension headers to RTP packets. This can be used for things like specifying additional payload information like the type of a video stream, screen

orientation, using a common inter-stream sequence number space [29], and transporting an audio level indication [44].

There is a secure version of the protocol: Secure RTP (SRTP [11]). It is mandatory for WebRTC [60] and provides encryption and authentication of the payload of RTP packets.

There are a few types of RTCP messages used in WebRTC. Receiver Reports (RRs) are periodically sent by receiving endpoints and include information about the packet loss, interarrival jitter, and a timestamp allowing computation of the round-trip time between the sender and receiver. Sender Reports include the number of packets and bytes sent, and a pair of timestamps facilitating inter-stream synchronization. Negative Acknowledgement (NACK [57]) packets can be used to explicitly indicate that a packet or set of packets have not been received (it is up to the application to optionally retransmit some of the referenced packets).

Full Intra Request (FIR [83]) and Picture Loss Indication (PLI [83]) packets are used for video to indicate that there is a need for the sender to produce a refresh point in the stream. We refer to these are keyframe requests, because with the codecs we are interested in a refresh point is equivalent to a keyframe.

Finally, there are RTCP messages related to congestion control. Receiver-Estimated Maximum Bitrate (REMB [5]) feedback packets signal to a sender the maximum bitrate a receiver wishes to receive [73]. Transport-wide Congestion Control (TCC) feedback packets [29] are used to provide detailed packet-by-packet reception information from a receiver to the sender.

In the next section we'll discuss how these protocols function with SFUs, and in our implementation in particular.

4.2 The Selective Forwarding Unit Implementation

Below we describe the architecture and features implemented in our conferencing system and the SFU in particular.

General overview: Jitsi Videobridge is an SFU implementation, mainly targeted at serving multi-party conferences with WebRTC-enabled endpoints. Presently it is deployed in conjunction with Jitsi Meet, a JavaScript WebRTC application for real-time

multi-party audio and video conferencing. The use-cases range from casual personal meetings to business meetings with shared remote presentations.

The system uses XMPP/Jingle [49] for signaling, with a separate entity acting as conference focus [67] and initiating a Jingle session with all endpoints. The focus allocates channels and resources to each participant and also controls the SFU through the COnferences with LIghtweight BRIdging (COLIBRI) protocol [36]. In addition to instances of Jitsi Meet running in WebRTC-capable browsers, Jitsi Videobridge can interact with other full-fledged client software, gateways to legacy networks (e.g., using SIP), and provides enhanced server-side services (e.g., recording the participants in a conference).

In addition to establishing channels for audio and video, each endpoint maintains a control channel to the SFU. Based on either WebRTC's data channels or a separate WebSocket [23] connection, this bi-directional and reliable transport channel allows for notification and control messages in cases where RTCP cannot be used. This is necessary because the WebRTC APIs do not provide a mechanism for the JavaScript application to directly send or receive RTCP messages; thus, protocol extensions which are not directly supported by the browser cannot be be implemented as RTCP. The messages themselves have a custom JSON-based format. Relevant to the experiments in this chapter are the "dominant speaker" notifications. They are sent from the SFU to endpoints to indicate that the main speaker in the conference has changed. There are also "start" and "stop" messages which the SFU can use to instruct an endpoint to start or stop its video stream (see section 4.3).

From a protocol standpoint, implementation of a Multipoint Control Unit is simple. It acts like any other client, completely terminating all functions of the RTP protocol (similar to how back-to-back user agents (B2BUA [66]) work for SIP). In contrast, Selective Forwarding Units cannot do the same. They need special handling for both RTP and RTCP.

RTP Header changes For the features discussed in this chapter, the SFU only needs to changes one field in the RTP headers: the abs-send-time RTP header extension [5] for video packets. It replaces the 24-bit timestamp in each outgoing RTP packet with a locally generated timestamp. This corrects the receiver's packet inter-arrival time calculation, so that only the path from the SFU to the endpoint is taken into account (but not the path from the sending endpoint to the SFU).

In later chapters, we discuss other RTP header modifications that an SFU may perform for more advanced features.

RTCP Termination The SFU terminates RTCP and generates RTCP packets for the RTP session. Incoming RTCP packets are handled according to their type. The SFU does not propagate Receiver Reports (RRs), generating new RRs for each incoming RTP stream instead. It propagates Sender Reports (SRs) to all endpoints (in later chapters we discuss how SRs need to be handled if the streams are modified).

Negative Acknowledgements are terminated, and if the specified packets are available in the SFUs cache they are retransmitted.

If Receiver Estimated Maximum Bitrate is used, then feedback packets need to be terminated. Otherwise, the senders receive contradictory reports from different receivers, leading to rate instability [74]. The SFU consumes incoming REMB packets for its bandwidth estimation and produces REMB packets for each sender.

Other types of RTCP packets are passed through including RTCP Extended Reports, and keyframe requests (PLI, FIR).

Security: Although the SFU does not change the payload of RTP packets in any way, it needs to change the RTP headers and generate RTCP packets. For this reason, it is not currently possible for the endpoints to exchange end-to-end encrypted media without the SFU being able to decrypt it (we discuss solutions to this problem in chapter 9). WebRTC mandates that all media is transported over Secure Real-time Transport Protocol (SRTP), with session keys exchanged over Datagram Transport Layer Security (DTLS) [60]. This forces the SFU to establish a separate SRTP context with each endpoint, necessitating the re-encryption of evert RTP packet individually.

A different scheme discussed for use with SFUs is Encrypted Key Transport (EKT) [52]. With EKT, the SFU and all endpoints share a single SRTP context, therefore, the SFU is not required to re-encrypt RTP packets (unless the headers were modified). As a result, the proposals discussed in this paper can be applied without any modifications.

In the following sections, we discuss our proposals for endpoint selection based on speaker order, and for identifying the dominant speaker(s) based on audio-level information.

4.3 Endpoint Selection Based on Audio Activity

Media conferences deployed in a full-star topology do not scale well when the number of participants is substantial, as each endpoint connected to the conferencing server sends and receives streams from each other endpoint.

We identified three issues for the receiving endpoints. First, network requirements grow in proportion to the number of participants. Second, CPU requirements grow the same way because the endpoint has to decrypt, decode, scale, and render all streams separately. Finally, the user interface cannot display a large number of video streams efficiently, due to insufficient screen real-estate.

The conferencing server experiences issues with CPU and network usage as well. However, this situation is problematic, because the resource use grows quadratically with the number of endpoints. This is because the streams from each new endpoint have to be encrypted and forwarded independently to each other endpoint. Our evaluation in section 4.5 shows that, depending on the hardware and the available network resources, either of these (CPU or network) can be the bottleneck.

Here we propose LastN: a general scheme defining a policy for endpoint selection based on audio activity. It is used by a Selective Forwarding Unit to choose only a subset of streams to forward at any given time, alleviating the previously identified limitations of the endpoints and conferencing server.

Last N: The SFU only forwards a fixed number of video streams (N) to each endpoint, changing the set of forwarded streams dynamically according to audio activity. Additionally, the receiving endpoint continue receiving streams that may have been chosen by the participant but are currently outside of the LastN set (a set of "pinned" endpoints). The LastN scheme only applies to forwarding video streams and not to audio; audio from all endpoints is always forwarded.

The integer N is a constant configured for the whole conference. We denote the total number of endpoints in a conference as K. Then, the SFU sends $K \times N$ video streams instead of the $K \times (K-1)$ streams sent in the case of a full-star topology. This enables the SFU to scale down the requirements on network capacity and CPU, allowing for more efficient handling of large conferences compared to the full-star topology.

From a user experience perspective, video from only a subset of all endpoints is displayed, but as soon as a participant of the conference starts to speak, their video is displayed automatically. To implement this scheme, the SFU relies heavily on identifying the dominant speaker. We discuss the details involved with this in the next section. The SFU maintains a list (L) with all endpoints currently in the conference, ordered by

the last time that an endpoint was identified as the dominant speaker; thus, the endpoint currently identified as the dominant speaker is always at the top of the list.

Suppose that an endpoint E has selected a set P of endpoints that it wants to permanently receive (P could be empty). Then the SFU forwards to E the selected endpoints and the first of the remaining endpoints, up to a total of at most N endpoints. That is, E receives P and the first (at most) $N - |P|$ endpoints from the list $L \setminus P \setminus \{E\}$.

Pausing video streams: When a stream from an endpoint is not forwarded by the SFU to any other participant, it is unnecessary for the endpoint to send the stream to the SFU (the SFU is effectively ignoring the stream). The SFU sends control messages to endpoints instructing them to temporarily pause, or to resume transmitting their video stream. These messages can be sent over the control channel or encoded as an RTCP extension report. Since WebRTC does not currently support such RTCP extensions, we implement them over the control channel.

This approach can be used with any algorithm the SFU might use to change its forwarding policy, as long as the SFU keeps track of whether a stream is being forwarded to at least one endpoint or not. With LastN and no selected endpoints, it is easy to see which streams are being forwarded because they are listed in the order they last spoke (L). All streams in L after stream $N + 1$ are not forwarded and can be paused. Hence, there are always $K - N - 1$ streams paused. The performance evaluation and results are further discussed in section 4.5.

Keyframes: When a paused stream is resumed, the endpoints require a keyframe (an I-frame) in order to begin decoding. Typically, when endpoints do not have a valid keyframe, they generate an RTCP Full Intra Request (FIR) message; senders generate a keyframe in response to such messages. Instead of waiting for the receivers to generate a FIR request, the SFU preemptively sends a FIR message when it instructs an endpoint to resume the video stream. Additionally, when an endpoint is elected dominant speaker and was not in the LastN set, its stream starts to be forwarded to all receivers. This allows a single FIR message to be sent after a dominant speaker change, even though there is often more than one receiver, effectively reducing the time to correctly decode and render a stream by one RTT (i.e., instead of waiting for the receivers to send FIRs).

Further Improvements: The LastN scheme proposed here can be easily extended with additional policies, providing more complex solutions suitable for some use-cases, while preserving many of the features. One example which may prove useful in remote presentations, is to have a global set of streams that are always forwarded (note that this is not the same as the endpoint-selected streams, which are per-endpoint). This can be implemented simply by reordering L and keeping the desired streams at the

top. Another is allowing N to vary per-endpoint, perhaps dynamically, according to the device and network conditions of the endpoint.

For remote presentations, always forwarding a given endpoint's video may be desirable. We can accomplish this by keeping the endpoint (or endpoints) at the top of the list L.

Participants in the conference would have the ability to individually select a set of endpoints to always receive video from. This would require ordering L differently for each receiver.

In the next section, we discuss how an SFU detects which endpoint is the active speaker, an integral part of the LastN scheme.

4.4 Dominant Speaker Identification

An essential feature for a conferencing system from a user experience perspective is the ability to dynamically switch focus to the currently speaking participant of the conference; further as an optimisation, the receiving endpoint may render not just the current dominant speaker but also a list of LastN speakers and/or set of selected speakers. SFUs enabling the LastN scheme perform Dominant Speaker Identification (DSI). Traditionally DSI is performed using raw audio streams [81], but this implementation is very inefficient for an SFU because it has to decode all incoming streams.

The Client-to-Mixer Audio Level Indication [44] RTP extension defines a way for audio senders to include the audio level of the audio sample carried in the RTP packet, encoded as a 7-bit value. This value is available to the SFU without the need to decode the packet.

Here we present an algorithm that performs DSI using solely the audio level (7 bits per packet) instead of the raw audio samples (typically 960 16-bit samples per packet). It is an adaptation of the state-of-the-art algorithm by Volfin [81].

The desired behavior of a dominant speaker identification algorithm is as follows:

- No false switching should occur during a dominant speech burst. Both transient noise occurrences and single words said in response to or in agreement with the dominant speaker are considered transient occurrences. These should not cause a speaker switch.

- A speaker switch event cannot occur during a break in speech between two dominant speakers. It must be triggered by the beginning of a speech burst.
- A tolerable delay in the transition from one speaker to another in a speaker switch event is up to one second.
- When simultaneous speech occurs on more than one channel, whoever spoke first is considered the dominant speaker.
- The relative loudness of a speaker's voice should not influence their chance of being identified as the dominant speaker.

A speech burst is defined as a speech event composed of three sequential phases: *initiation*, *steady-state* and *termination*. In the initiation phase, speech activity builds up. During the steady-state, speech activity is mostly high, though it may include breaks in activity due to pauses in speech. Finally, in the termination phase, speech activity declines and then stops. Typically, a dominant speech activity is composed of one or more consecutive speech bursts. We refer to the point where a change in dominant speaker occurs as a speaker switch event.

The algorithm implemented for dominant speaker identification uses speech activity information from time intervals of varying lengths. It consists of two stages: a local processing and a global decision. In the first stage, each participant's audio signal is processed independently while speech activity scores are evaluated for the immediate, medium, and long time-intervals. The lengths of the time intervals correspond to and allow capturing basic speech events such as a few phonemes, a word or two, or a short sentence. In the second stage, the dominant speaker is identified based on the speech activity scores obtained in the first stage, and speaker switch events are detected. We assume that a speaker switch event can be inferred from a rise in the three speech activity scores on a certain channel, relative to scores of the dominant channel.

Sequences and combinations of basic speech events may indicate either the presence or absence of dominant speech activity. This method distinguishes between isolated transient audio occurrences and those occurring within a speech burst.

We use long term information to determine whether speech is present in a currently observed time-frame since dominant speech activity in a given time-frame is better inferred from a preceding time interval than from any instantaneous signal property.

Local Processing: In this stage, the signal in each channel is processed separately to place each signal frame into a broader context than its instantaneous audio activity.

This is accomplished by processing the currently observed frame by itself in addition to a medium- length preceding time interval and in addition to a long time interval that precedes it. Thus, each time we move up to a longer time interval, the speech activity obtained in the previous step is analysed again in a broader context.

We consider each time interval as composed of smaller sub-units. We determine the speech activity in each time interval according to the number of active sub-units by attributing a speech activity score to this number. The score is obtained from the likelihood ratio between hypothesis of speech presence and hypothesis of speech absence. The speech activity evaluation process consists of three sequential steps, referred to as immediate, medium, and long. The input into each step is a sequence of the number of active sub-units acquired in the previous step. A thresholding approach allows measurement of the amount of speech activity while suppressing isolated high-energy noise spikes.

For the step of immediate speech activity evaluation, we use the client-to-mixer audio level indication of the frame [44] rather than a frequency representation of the frame. This is made possible by thresholding as long as the replacement input presents a comparatively similar speech activity estimation to the original frequency sub-band test.

Audio levels are expressed in dBov (with values from -127 to 0), which is the level, in decibels, relative to the overload point of the system, i.e., the highest-intensity signal. In order to reuse the heuristically derived parameters of the original algorithm, such as the thresholds, we divide the audio level range into the same number of sub-bands as the number of frequency sub-bands analysed by the original algorithm [81].

Additionally, the algorithm accounts for differences in the relative loudness of the background noise present in the audio signal of a participant. We apply a separate, individual initial thresholding of the input based on the history of minimum audio level indication reported by the participant[51]. The history maintains the minimum by considering its recency for the purposes of reflecting medium- and long-term changes in the level of the background noise.

Global Decision: The objective of this stage is to identify the dominant speaker. This stage is activated in time steps of a certain interval, which is referred to as the decision-interval. It utilizes scores obtained in the local processing stage for dominant speaker identification. We approach this stage by detecting speaker switch events, rather than selecting a dominant speaker in every decision-interval. Once a dominant speaker is identified, he remains dominant until the speech activity on one of the other channels

justifies a speaker switch. We refer to non-dominant channels as competing (for dominance).

4.5 Performance Evaluation

In this section, we present the experimental results. We performed controlled experiments measuring the resource use on both clients and servers. The following paragraphs describe the testing environment, the experimental setup, and the results from all experiments performed.

4.5.1 Testbed

The SFU is running on a dedicated machine with a quad-core Intel Xeon E5-1620 v2 @ $3.70GHz$ processor. All tests comprise of a single conference with a varying number of participants. One endpoint in the conference is a Chrome instance running *Jitsi Meet*, with the microphone and camera disabled. The rest of the endpoints come from a Jitsi Hammer[1] instance (also see [Pub13]). This is an application developed specifically for load-testing the SFU. It creates multiple endpoints, each streaming one audio and one video stream from pre-recorded files. The two streams have a combined average bitrate of about $515\,\mathrm{Kbps}$. These endpoints do not implement any congestion control mechanism and their RTCP support is limited. They only emit Sender Reports and do not react to REMB, NACK, or FIR messages. The streaming video includes keyframes every 1.3 seconds on average, which allows decoders to relatively quickly regain decoding state, thus, the state of the conference can be monitored without the need to handle FIR or NACK requests.

To trigger changes to the dominant speaker in the conference, the participating endpoints (from Jitsi Hammer) add audio-levels in the format used by the DSI algorithm [44]. At any point in time, one endpoint sends levels associated with the human speech, while the others send silence. The endpoint that sends speech levels changes every 4 seconds, with the new one being chosen at random. Note that we change only the audio-level field and not the RTP payload of the audio streams. The "start" and "stop" messages are implemented with RTCP packets.

We denote the number of load-generating endpoints K, so, including the Chrome instance, there are always $K + 1$ endpoints in the conference. We denote the LastN

[1] https://github.com/jitsi/jitsi-hammer

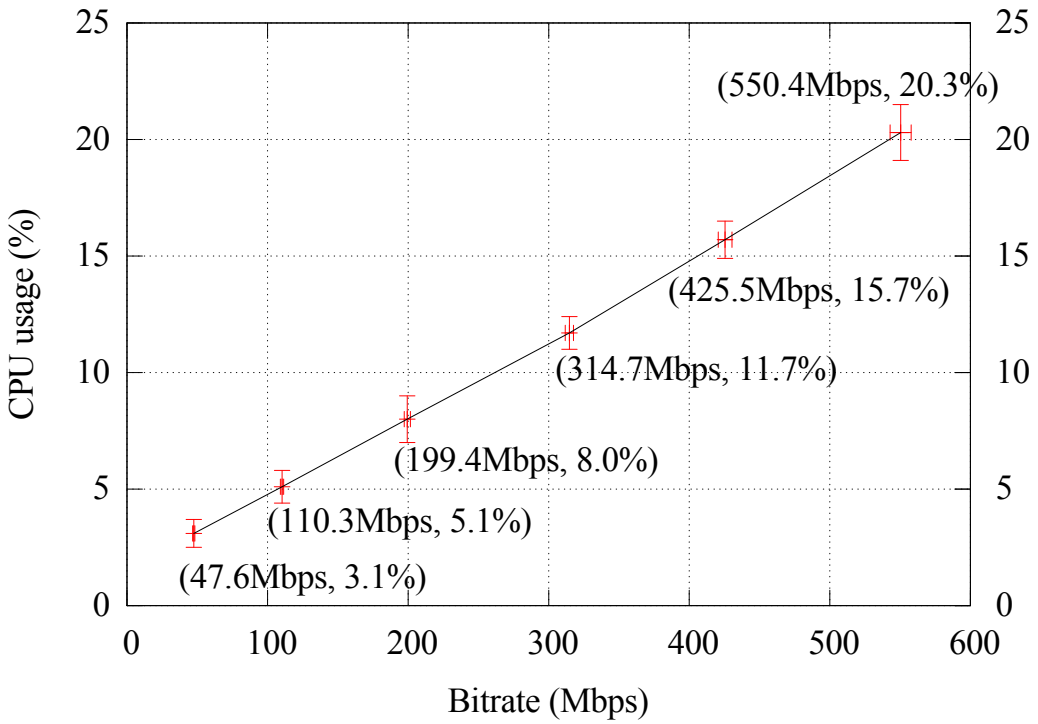


Figure 4.1: *The CPU usage as it relates to total bitrate in a full-star conference. The error bars represent one standard deviation. A clear linear correlation.*

value N, so each endpoint receives (at most) N video streams. With $N = -1$ we denote that LastN is not used, and all streams are forwarded to everyone. The number of video streams sent by the SFU is either $(K+1) \times N$, or $(K+1) \times (K-1)$ (full-star, $N = -1$). The number of streams received is either K or $N+1$, depending on whether video pausing is used.

All data is gathered in intervals of one second. On all graphs, the values are means over an interval of at least 120 seconds, and the error bars indicate one standard deviation of the sample. The measured CPU usage constitutes the fraction of the last 1-second interval which the CPU spent in either `User`, `Nice`, `System` or `IOWait` state (what the `top` command shows on the "CPU(s)" line), and 100% indicates that all 8 logical cores were in use.

4.5.2 CPU Usage in a Full-star Conference

In this test, we measured the CPU and bandwidth used in a conference with a full-star topology ($N = -1$). We varied the number of endpoints K (10, 15, 20, 25, 29, and 33). Figure 4.1 shows the variation in CPU usage compared to the aggregate bitrate. We observe linear growth, with the best linear fit having $R^2 > 0.999$. This is to be expected because the most computationally intensive part of the SFU's operation is encryption and decryption, which is a very predictable process and depends almost entirely on the amount of data processed.

Based on these values, we conclude that in reasonable practical scenarios, both the computing and the network resources might constitute a bottleneck. For example, an average machine with a $100Mbps$ link may hit the network bottleneck first, while a less powerful machine cannot cope with $1Gbps$ of traffic. Assuming that about $500\,\mathrm{Kbps}$ is a typical bitrate for a video stream in a conference, we observe that, as expected, the required resources increase rapidly: a conference with 33 participants requires on the order of 500 Mbps capacity for the SFU.

4.5.3 Full-star vs LastN

In this test we measured the output bitrate of the SFU with $K = 10, 15, 20, 25, 30$, with four configurations: full-star, and LastN with $N = 3, 5, 8$. Video pausing was enabled. Figure 4.2 shows that when LastN is used, the bitrate grows linearly with K, with a coefficient depending on N, while with no LastN it grows quadratically. This constitutes a significant improvement when LastN is used. For $K = 30$, the gain is 72%, 82%, and 89% for N 8, 5, and 3, respectively. Counting the aggregate bitrate the gain is almost the same: 72%, 82%, and 88%. We also see that different compromises between bitrate and number of visible videos can be achieved by simply varying the value of N.

4.5.4 Video Pausing

In this test, we observed the differences in the total bitrate and CPU usage between the two modes of LastN: with video pausing on and off. We fixed $K = 30$ and changed N to *25, 20, 15, 10, 8, 5, 3, 2, 1*. The graphs also show values for $N = 30$; for them LastN is disabled. We plot them as $N = 30$, because it is effectively the same (i.e., in both cases everything is being forwarded). Figure 4.3 shows variation in the bitrate, and figure 4.4 shows the variation in CPU usage. We observe that for high values of N,

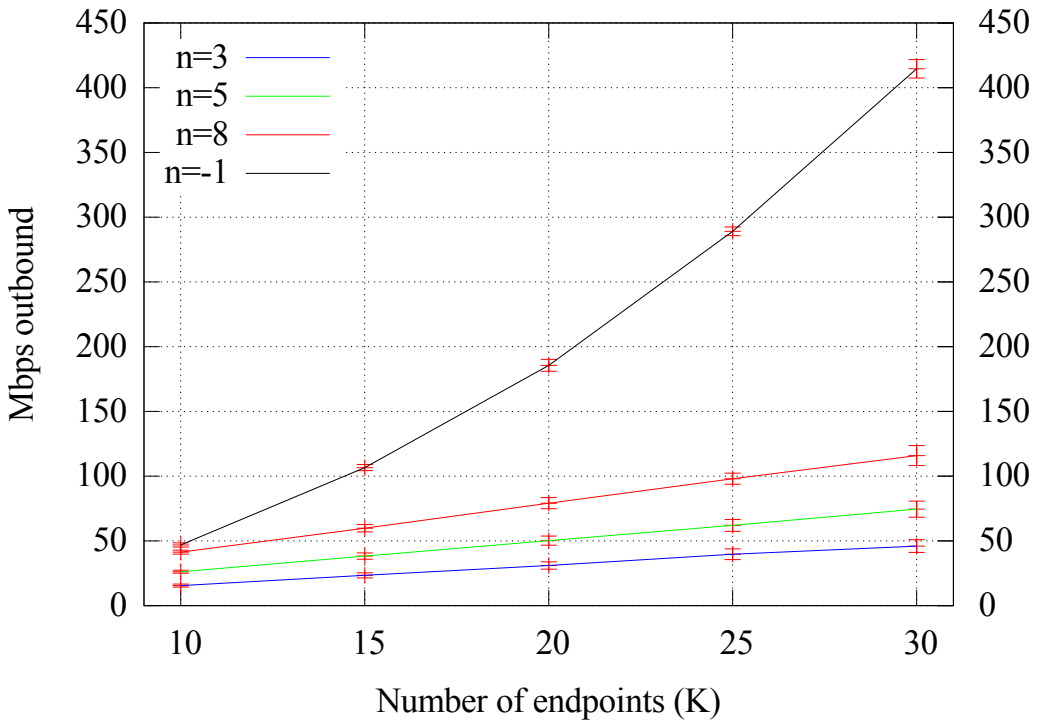


Figure 4.2: *The outbound bitrate at the SFU, as the number of endpoints grows. Shown are four LastN configurations. The error bars represent one standard deviation.*

there is little or no gain, but it increases when N is lower. This is expected because 1) the total bitrate is lower, and 2) there are more endpoints with paused video. When N is set to 8,[2] we see an 8% decrease in bitrate and a 16% decrease in CPU usage. We expect that pausing video streams are beneficial in practice because it has the additional advantage of decreasing CPU usage for the endpoints with paused video by eliminating the need to encode video.

4.6 Conclusion

In this chapter, we propose a scheme (LastN) that uses an SFU to select which video streams to forward and which to drop. We present an implementation and evaluate it in terms of network usage and computing resources. We examine the effects of varying the number of forwarded video streams (N), and the possibility of temporarily turning unused streams off. Our results show that the expected performance gains are achiev-

[2]We assume this to be a reasonable value for practical use, because of user interface considerations.

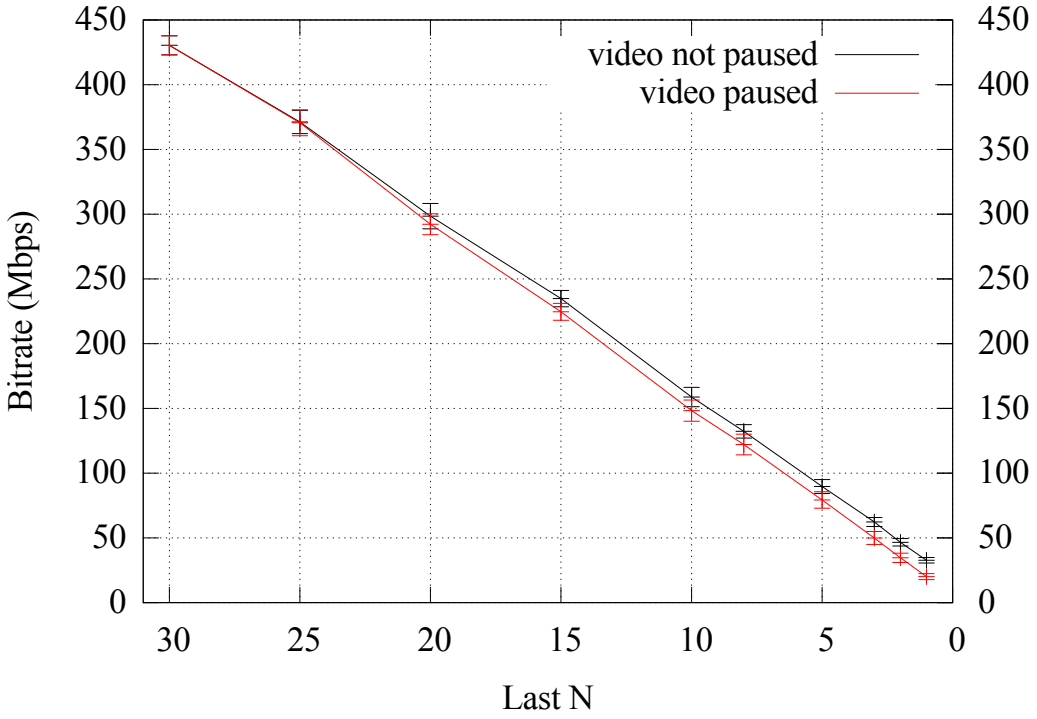


Figure 4.3: *The aggregate bitrate at the SFU for a fixed number of endpoints (30) and varying value of N. Compared are the two modes of LastN: with and without video pausing.*

able in practice. Notably, we observe a drop from a quadratic to a linear (with respect to the total number of endpoints) growth of the outbound bitrate used by the SFU. Additionally, for a fixed number of endpoints, tweaking the values of N results in gradual changes to the bitrate, making the system adaptable to different situations. Based on the results, we expect that LastN is useful in practice in two scenarios: allowing for small-to-medium conferences to work more efficiently (thus allowing a single server to handle more conferences), and increasing the maximum number of endpoints in a conference, given the same resources.

LastN is quite general and allows for different modifications and improvements on top of it. In particular, one interesting modification, which we intend to study, is maintaining different values of N for each receiver, while keeping the list L global for the conference. In chapter 7, we look at another use-case for LastN: dynamically adapting server configuration to the experienced load.

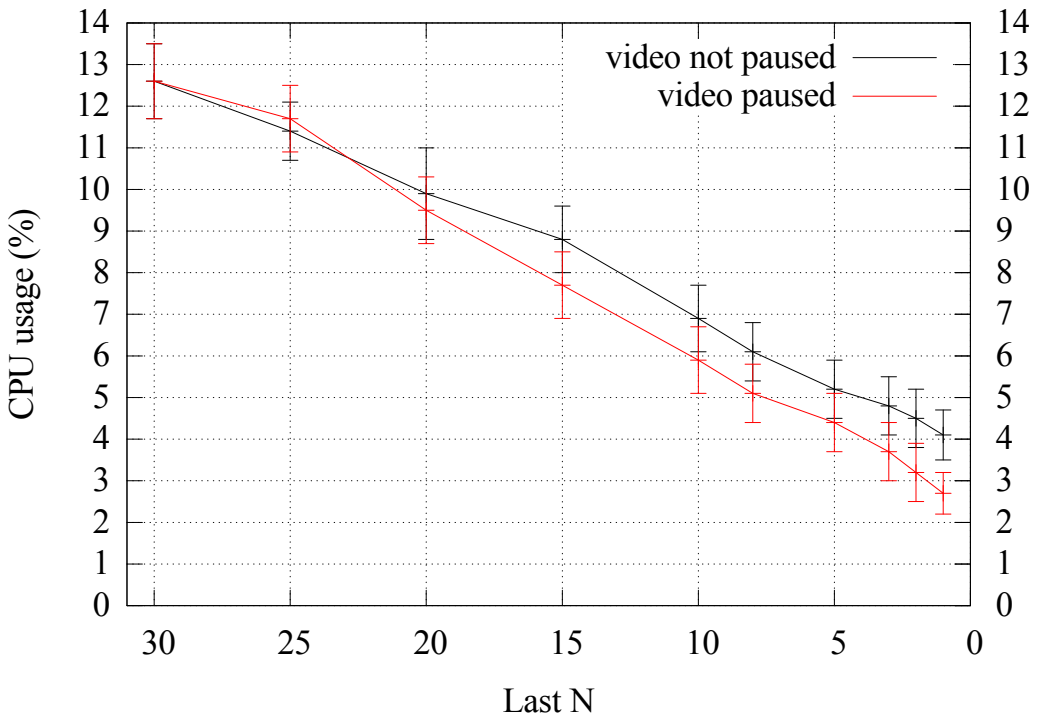


Figure 4.4: *The CPU usage at the SFU for a fixed number of endpoints (30) and varying value of N. Compared are the two modes of LastN: with and without video pausing.*

In the next chapter we introduce another technique which can be used to further decrease the resources required for a conference.

Chapter 5

Simulcast

The LastN scheme described in the previous chapter significantly reduces the bandwidth required for a conference, but it does not solve all problems. Used by itself, this solution has a few disadvantages. First, it completely drops certain streams, despite the available bandwidth being sufficient for a lower quality stream that would still benefit the user. Second, it is inefficient in many use cases, because it forwards only full-resolution streams, which the client may render down-scaled because of user interface constraints. Third, because it only uses full-resulution high-bitrate streams, there is little flexibility in the SFU's ability to adapt to changes in the available bandwidth – entire streams must be either dropped or resumed.

Conferencing systems commonly use viewports of varying sizes for the different endpoints, as depicted in figure 3.4. Simply using LastN is not optimal in such cases, because the same result can be achieved with fewer resources if the streams displayed in a small viewport were scaled down before they are encoded. By scaling down the streams displayed in a small viewport we can achieve lower overall bitrate; or equivalently, we can achieve the same overall bitrate with a larger value of N, which means that more streams can be displayed.

Scaling the streams sent over the network to match the viewports displaying them is not straightforward for a couple of reasons. Firstly, the viewports' sizes may vary from endpoint to endpoint. Endpoints may have displays of varying size, they may have selected different streams to display in their large viewport (notably, if all endpoints show the current active speaker, the active speaker likely show another endpoint), or they may use a different user interface layout altogether. These situations are especially likely to occur when the conference has a mix of participants using mobile and desktop devices. Secondly, the configuration may change at any time, such as when the speakers

change. Adapting to this requires the SFU to control the encoding bitrate of senders, which is complex and incurs a delay. Additionally, it does not account for the different requirements of each receiver.

One way to address these problems is with the use of simulcast. Simulcast consists of the simultaneous encoding and transmission of each endpoint's video stream using different resolutions and bitrates. This mechanism allows an SFU to dynamically change the output bitrate, by switching to a different encoding of the stream, in response to changes in the network conditions, and which allows for more flexible congestion control.

In this chapter, we perform an analysis and an experimental evaluation on the use of simulcast in a WebRTC based conferencing system. We evaluate the resource usage of sending and receiving devices and server components, as well as the end-to-end image quality. We also explore the problem of the image quality change the user experiences when a switch between different encodings occurs, and propose a method of reducing the delay before the switch happens.

The remainder of this chapter is structured as follows: Section 5.1 describes how simulcast is used in WebRTC, and particularly in the Jitsi Meet system. Section 5.2 describes the metrics that we are interested in obtaining. Section 5.3 describes one improvement we propose, and section 5.4 presents our experimental evaluation. Finally, section 5.5 concludes the discussion and presents ideas for further research.

5.1 Simulcast in WebRTC

Google Chrome was the first major browser to support simulcast. It works through a limited and non-standard JavaScript API, and it was only available for the VP8 codec. The technique requires modifying the SDP blobs passed on to the PeerConnection API. In the local description, an "ssrc-group" [45] with semantics "SIM" is added, which contains two or three SSRCs, depending on the desired number of resolutions to send. In the remote description, the special media-line level attribute "x-conference-flag" is added. With this configuration, the sender produces and transmits additional encodings for its video stream. Each additional encoding halves the resolution of the previous in each dimension, and uses a lower encoding target bitrate. For example, if the main stream is 1280x720 and there are two additional encodings, with resolutions of 640x360 and 320x180.

On the RTP level, each encoding uses a separate SSRC (see figure 5.1), and the encodings are entirely independent. That is, each encoding represents a complete stream, which can be decoded on its own.

5.1.1 Standardization

Simulcast is one of the less mature parts of the WebRTC standards, though there has been recent work addressing some of its aspects. At the signaling level, the Javascript Session Establishment Protocol (JSEP) internet draft [78] includes support for RTP Payload Format Constraints (RID) [77, 63], a framework for identifying RTP streams within an RTP session, as well as restricting their payload format parameters. RID adds to the JSEP specification the expressive power required for signaling simulcast and temporal and spatial scalability. When using JSEP to signal multiple encodings, the "m=" section for the RTP sender includes an "a=simulcast" attribute with a "send" simulcast stream description that lists each desired encoding, and an "a=rid" attribute for each encoding. The use of RID identifiers allows for disambiguation of the individual encodings, even when they are all part of the same "m=" section.

At the transport level, the Frame Marking RTP Header Extension draft [19] describes a payload-type agnostic mechanism for conveying information about video frames, which may be necessary for the operation of SFUs (e.g. the ID of the frame's temporal layer). The draft has not been adopted by major implementations, meaning that current SFUs need to be payload-type aware and extract the information they need from the RTP payload.

As of March 2017, the W3C WebRTC 1.0 editor's draft [12] has been extended with support for sending multiple RTP encodings from a single `RTCRtpSender`, which naturally enables sending simulcast in the objectified parts of the WebRTC API.

At the time of this writing, one gray area in all of the standardization efforts is receipt of simulcast. A simulcast sender transmits the different versions of its video source as separate RTP streams, each with a unique Synchronization Source (SSRC) identifier. There is more than one way to handle this at the receiver side. One option is to make the receiver simulcast-aware and expose it to multiple independent incoming streams (any of which may be paused or resumed throughout the session). It then needs to decide which stream to render at any time. However, neither the JSEP draft nor the objectified parts of the WebRTC API provide an API to configure receipt of simulcast. This means that if simulcast is offered by the remote endpoint, the answer generated by a JSEP endpoint does not indicate support for receipt of simulcast, resulting in the remote endpoint only sending a single encoding per "m=" section. Meaning, with the currently

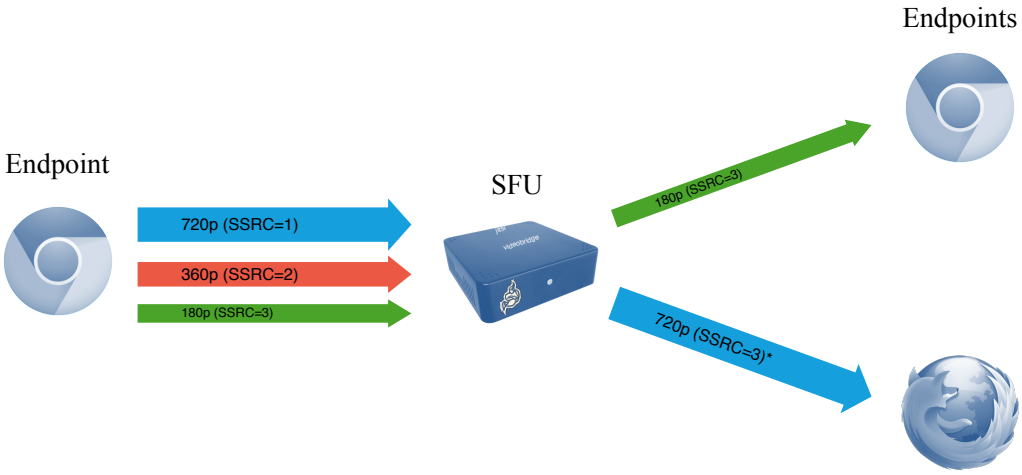


Figure 5.1: *An illustration of the use of simulcast with an SFU. Note that the SFU rewrites the SSRC in order to produce a single continuous stream with SSRC=3 while switching between the input streams.*

available standards and browser implementations, the implementation has to be done in the JavaScript layer, and this requires having multiple HTML "video" elements and switching the one being displayed. Since the JavaScript code has very limited control over what is being rendered (e.g., it does not have access to the frames that a "video" element renders and their timestamps), it is not clear whether this solution can be implemented reliably and without noticeable glitches when a video element is changed.

Another option is to move this logic to the SFU, producing a single RTP stream for each sender-receiver pair, effectively making simulcast transparent to the receiver. To correctly construct an RTP stream out of the multiple streams that it receives, the SFU modifies the SSRC, RTP sequence number, and RTP timestamp fields in RTP packets. This is illustrated in figure 5.1.

We have chosen to use the second option, as it allows us to guarantee that the implementation works well for all clients (including mobile), while keeping the simulcast-related logic in one place.

5.1.2 Simulcast in Jitsi Meet

The Jitsi Meet interface shows one remote video in full size (we refer to the associated endpoint as "on-stage" or "main"), and all other remote videos scaled down to thumbnail size in a "filmstrip" overlayed on top of the main video. This is clearly demonstrated in figure 3.4.

This layout works very well with simulcast because most streams can be scaled down to a lower resolution. Subject to bandwidth restrictions, the SFU sends the on-stage video with the highest available resolution and the rest with the lowest available resolution.

There are two methods of selecting the on-stage endpoint, and different endpoints in a conference can use either of them. The default is to track the current active speaker (with the active speaker itself showing the previous active speaker on-stage). The second is actively selecting a specific endpoint to be selected. This user can initiate this by clicking on one of the thumbnail videos. Both selection strategies are client-driven. That is, the client makes a decision for the desired on-stage video and sends a message to the SFU notifying it of any changes.

In a typical case, the source streams are 1280x720 (720p) pixels at 30 frames per second (fps). With simulcast this results in two additional encodings with resolutions of 640x360 (360p) and 320x180 (180p) both at $30\,\mathrm{fps}$. The encoding bitrates for the streams are limited to $2.5\,\mathrm{Mbps}$, $0.5\,\mathrm{Mbps}$, and $0.15\,\mathrm{Mbps}$, respectively. As the available bandwidth drops, senders first reduce the target bitrate of the large stream, and then turn it off completely (and they do the same with the middle stream if the available bandwidth drops further).

The video codec in use is VP8, because as of this writing it is the most widely supported video codec by WebRTC-enabled browsers (supported by Google Chrome, Mozilla Firefox, Opera, as well as Internet Explorer and Safari via third party plugins), and because *webrtc.org* includes an efficient VP8 simulcast implementation.

The system also supports a non-simulcast mode, and it is normally used for browsers which do no support sending simulcast. In the evaluation below we intentionally use this mode with the Chrome browser to make a comparison. Without simulcast, clients typically send a VP8-encoded 720p stream. The exact target bitrate depends on the network conditions and is limited to at most $2.5\,\mathrm{Mbps}$, the same as the the high resolution simulcast stream.

5.2 Metrics

Ultimately, we want to understand the effect of simulcast on user experience. In this study, we examine certain related variables under controlled conditions, and we compare the simulcast case with the non-simulcast case. We look at the following metrics: objective image quality, stream-switching delay, rendered frame rate, CPU utilization and network usage (on both a client and a server). The rest of this section discusses these metrics in further detail.

Image quality Peak Signal-to-Noise Ratio (PSNR [82]) is a direct measure of the difference between two images. When applied end-to-end (comparing the image before encoding at the sender and the rendered image at the receiver) it gives an objective metric of the image quality. We also experimented with the Structural Similarity (SSIM [82]) index, but we found that the results, when normalized to fit in the same range, are nearly identical to PSNR. Based on the results of Dosselmann et al. [18], these results were expected.

Client resource usage Client-side CPU usage is of critical concern. If system resources are overused it can negatively impact the user in many ways: the application may be slow to respond; other applications may be slow to respond; degraded encoding: for example, the browser may choose to limit the encoding resolution if it detects high CPU load; video streams may fail to render all frames, resulting in freezing or "choppy" video. It is hard to determine precisely when these effects may manifest; it depends on factors such as hardware and applications running in the background, among others.

End-users of a service often have limited network bandwidth. The uplink restrictions are usually stricter [17, 2], but as participants in a conference receive multiple video streams, they may reach their downlink limits first. Therefore, a conferencing system's ability to optimize network bandwidth usage in both directions is vitally important.

Rendered frame rate One metric reflective of degraded user experience is the rendering frame rate for remote videos. If a receiver's rendering rate is less than the rate of the received streams, it may indicate the overuse of the receiver's resources.

Server resource usage In regards to the server, reducing CPU usage makes it possible to host more conferences per SFU, decreasing the infrastructure costs of running an

SFU-based conferencing service. Reducing the network usage further decreases these costs.

Stream switching latency In our implementation, the client decides which participant to show on-stage and then notifies the SFU of their choice This means that there is a non-nill time interval between the moment a client selects to promote a new on-stage participant, and the moment the high-quality stream for this participant becomes available and ready to render. We refer to the length of this period as the "switching latency"

During the switching period, clients display the low resolution (180p) stream scaled up to the available space. If the latency is low, the transition from low resolution to high resolution is not visible, because the switch is accompanied by a fade-in effect. When the latency gets higher, there is a visible flicker as the resolution changes, which may be distracting for the user. Our observations determine that this phenomenon becomes noticeable when the switching latency approaches 500 milliseconds.

This effect may occur every time that the active speaker of the conference changes, and data from the *meet.jit.si* service shows that these changes happen frequently: once every 22 seconds on average. It also occurs when the user manually selects another endpoint. Because of all this we strive to minimize the switching latency.

5.3 Preemptive Keyframe Requests

To minimize the stream switching latency, we propose using logic in the SFU to anticipate clients' switching behaviour and initiate a possible switch earlier.

When the SFU switches from forwarding one version of a stream to another (for example, from the 180p stream to the 720p stream), it needs to make sure that the first frame from the new stream is independently decodable (in the case of the VP8 codec this means that it is a keyframe [9]). When a key frame is necessary, the SFU requests one from the sender with an RTCP Full Intra Request (FIR) [83] message, and must wait for at least one RTT before the frame arrives. This means that there is always a certain delay before the SFU can switch to another stream.

The decision of which stream to forward in high quality is driven by the receiver. When it elects a new "on stage" participant, it signals this to the SFU (via a message over WebRTC data channels), initiating a switch to the high quality stream. The switch requires a keyframe, so the SFU requests one from the sender. Meaning there is a delay

of at least two round trip times (one receive-to-SFU and one sender-to-SFU) between when the receiver requests the high-quality stream and the stream being available.

The SFU performs dominant speaker identification (see chapter 4) and notifies clients (again, via a data channel message) about the change. When a speaker change occurs, we can expect that some of the receiving clients may want to soon switch their selection. This allows us to request a keyframe earlier.

We propose the following: suppose that the dominant speaker in a conference changes, and let us refer to the new dominant speaker as "sender". Let d_s be the round trip time for the sender, and d_i be the round trip time for the i-th receiver. Then the SFU schedules the sending of an FIR after a delay $d = max_i(d_i) - d_s + e$ (or no delay, if $d <= 0$).

This scheme aims to have the keyframe arrive soon after the last receiver indicates a switch to the new stream (hence $max_i(d_i)$). If the keyframe arrives too early for a particular receiver, it is discarded by the SFU and a new request has to be sent. Since the *webrtc.org* engine generates a keyframe at most once every $300\,\mathrm{ms}$, this significantly increases the delay in switching to the new stream.

The term e is an additional delay added to reduce the probability of the keyframe arriving too early (incurring the $300\,\mathrm{ms}$ penalty) due to e.g., network jitter. We use a value of $10\,\mathrm{ms}$.

In the section below, we evaluate the effectiveness of the approach by measuring the stream switching delay directly.

5.4 Results

5.4.1 Experimental Testbed

We set up a testbed consisting of seven physical machines connected to the same LAN via $1\,\mathrm{Gbps}$ Ethernet. We use one of the machines as a server (for both signaling and media forwarding). It runs Ubuntu 15.10 on an Intel® Core™ i7-5557U dual-core processor running at $3.1\,\mathrm{GHz}$.

Another machine is used for the client-side measurements. It runs Ubuntu 16.04 on an Intel® Core™ i7-4712HQ quad-code processor running at $2.3\,\mathrm{GHz}$. Both of these machines have CPU frequency scaling effectively disabled by using the *performance* governor for *cpufreq*.

Figure 5.2: *One of the frames from the sequence with a unique identifier encoded as a QR code in the upper left corner.*

The remaining machines are used solely for load-generation, with no measurements taken there, apart from some sanity checks on the load and output bitrate. Each of them runs up to four conference participants, each within a separate Chrome tab.

CPU usage measurements are taken using the *iostat* utility. Network usage measurements are taken using *ifstat* with a window of 1 second.

To measure end-to-end PSNR, we encode a frame number identifier as a QR code and overlay it in the input sequence (see figure 5.2). On the receiver side, we render frames in an HTML *canvas* element and capture its contents in *png* format with lossless compression. We then scan the QR code in the rendered frame, find the corresponding input frame, and compute the PSNR using the *compare* utility from the *imagemagick* package. Obviously this kind of processing has effects on the receiver performance, so these measurements are performed in a separate experiment.

We use the well-known *Four People* sequence[1], with the original 1280x720 resolution and downscaled to 30 frames per second. The structure and motion of this particular video sequence are similar to a typical video-conferencing scene, making it a good fit for our scenario. Each frame has an embedded QR code used to match output frames to input frames. The QR code is in the top-left corner and is 350x350 pixels in size. The

[1] https://media.xiph.org/video/derf/

reason for the relatively large size is that it needs to be recognizable after scaling down to 320x180 and encoding. The resulting sequence has 302 frames and is just over 10 seconds long.

Given a fixed target bitrate, a VP8 encoder (and likely any other video encoder) produces encoded frames whose PSNR from the input depends significantly on the input, so we take care to always compare PSNR from corresponding frames of the input sequence (or from the exact same sequence of frames, where we show mean values).

We use version 51 of the Google Chrome web browser as the client, and the VP8 codec.

For the non-simulcast tests, the senders are sending a 1280x720 stream, encoded with a target bitrate (t.b.) of 2.5 Mbps. For simulcast, they are sending one 1280x720 stream with a t.b. 2.5 Mbps, one 640x360 stream with a t.b. of 0.5 Mbps, and one 320x180 stream with a t.b. of 0.15 Mbps.

We use Chrome's *webrtc-internals* tool to measure the received and rendered frame rates.

We use two simple scenarios in this evaluation. The first, used for measuring the effects on the senders, has one receive-only endpoint and one isolated sender endpoint. The second scenario, used for all other measurements, has a single conference with a variable number of participants. Each participant is both a sender and a receiver, and one of the receivers runs on a dedicated machine where the effects on the receiver are measured. We were able to test conferences with up to 16 participants with simulcast, but only up to 10 without simulcast, due the client machines' inability to handle the load with 11 participants.

5.4.2 Peak Signal-to-Noise Ratio (PSNR)

Figure 5.3 shows the frame-by-frame PSNR score for the duration of one input sequence. We observe consistent long term variations, which are explained by features of the input stream. We also see that the temporal scalability feature results in high frame-by-frame variation. This is because when it is enabled different temporal layers are allocated different bitrate budgets. Since we do not make use of temporal scalability in the SFU, and since the mean values were the same, we disabled this feature in the rest of the tests.

More importantly, we see that with simulcast enabled, the encoder produces a consistently lower PSNR. This is explained by the motion vector reuse in the simulcast

encoder in *libvpx* [2]. This feature allows the motion vectors calculated by one of the encoders to be reused by the other encoders, thus providing gains in terms of CPU usage. We verified this by disabling the feature, resulting in matching PSNR between the simulcast and non-simulcast encoders. However, in all other tests we have left the feature enabled.

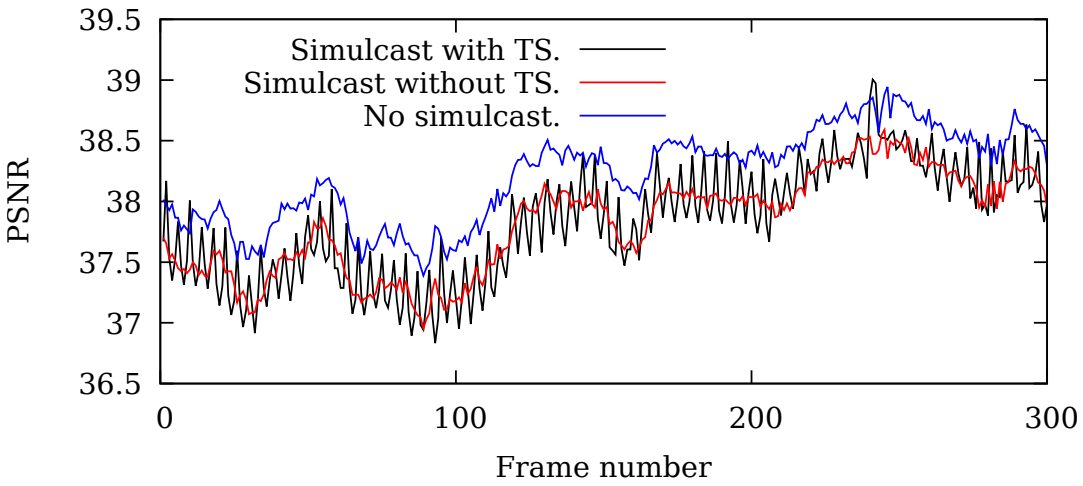


Figure 5.3: *PSNR on stage for the duration of one input sequence (*10 s*,* 302 *frames). The large-scale variation is due to the specific input. Simulcast produces lower PSNR due to motion vector reuse.*

To better quantify the effect size, we measure the average PSNR across all frames of two full loops of the input sequence. In the full resolution (1280x720) video we observe an average PSNR of 37.83 dB with simulcast, and 38.21 dB without simulcast.

This difference of 0.38 dB corresponds to a difference in bitrate of about 300 Kbps or about 12% of the total bitrate. That is, in experiments with the same setup, we observe a PSNR of 37.79 (very close to the result of simulcast above) with a non-simulcast encoder with a target bitrate of 2.2 Mbps.

For the thumbnail streams, we compare the low quality simulcast stream (320x180, encoded at 0.15 Mbps) against the full non-simulcast stream (1280x720, encoded at 2.5 Mbps) but down-scaled and rendered in a 320x180 canvas. Doing so ensures we compare exactly what the user sees in the interface of our application. We observe an average PSNR of 23.12 dB with simulcast, and 23.62 dB without simulcast.

[2] We would like to thank Mo Zanaty (Cisco) and Marco Paniconi (Google) for this particular insight.

5.4.3 Frame Rate

We observe no difference between the received and rendered frame rate for either simulcast or non-simulcast. Even in the largest conferences, the receiver machine is able to render at the full transmitted rate. However, with simulcast, senders use a frame rate of 15 frames per second for the low-quality stream. This allows them to achieve the relatively high PSNR using only $0.15\,\mathrm{Mbps}$. All full-resolution streams use $30\,\mathrm{fps}$.

5.4.4 Stream Switching

To evaluate the preemptive keyframe request mechanism, we measure the stream switching delay directly on the receiving client. Specifically, we measure the time between when the request for the high quality stream is sent, and when the HTML "video" element changes its resolution.

The length of this delay depends on many variables, including the round trip time between the SFU, receivers, and senders; the time needed for the sender to generate a keyframe; the length of the receiver's jitter buffer. We measure the delay directly from clients running in a production environment[3]. We took measurements over the course of about one week, once immediately prior to introducing preemptive keyframe requests, and once immediately after.

Figure 5.4 shows the cumulative distribution of the delay. The reason that the graphs plane out so low (below 80%) is that the data contains events that were significantly delayed by the sender failing to transmit a high quality stream when the receiver requests it. In this case, the SFU continues sending the available low- or mid-quality stream until the high-quality stream becomes available. We observe that the median delay is lower when preemptive keyframe requests are used ($435\,\mathrm{ms}$ vs. $502\,\mathrm{ms}$).

5.4.5 Sender Load

As expected because of the configuration of the streams, with simulcast enabled, the sender transmits 26% more data than without simulcast (at $3.15\,\mathrm{Mbps}$ vs. $2.5\,\mathrm{Mbps}$).

Next, we observe the sender as it participates in a conference with one more receive-only endpoint, and examine the CPU usage. For this experiment, we identify two pa-

[3] https://meet.jit.si

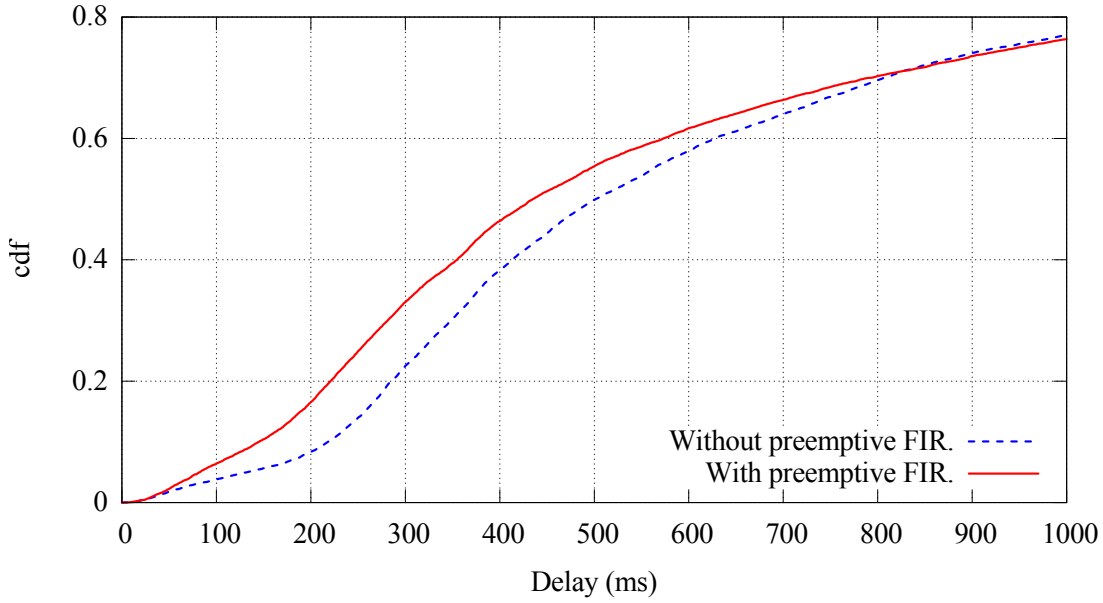


Figure 5.4: *Cumulative distribution function of the stream switching delay measured on the client.*

rameters: simulcast (on and off) and rendering of the local video (enabled or disabled). We sampled the CPU usage once a second for 30 seconds.

A summary of the results is in Figure 5.5. When the local video is not rendered, the CPU usage for the simulcast case is very close to the non-simulcast case. The means are 11.7% for non-simulcast and 12.2% for simulcast. Rendering the local video significantly increases the CPU usage in both cases. The means for the rendered video are 15.7% for non-simulcast and 17.9% for simulcast.

These results, while highly variable, show that the added cost due to simulcast is not significant. In fact, any overhead incurred by using simulcast is likely less than the cost of rendering the local video in a browser. One of the reasons for this low overhead is the motion vector reuse feature of the VP8 simulcast encoder in *libvpx*.

5.4.6 Receiver Load

We examine the incoming bitrate on a receiver in a conference when varying the number of senders. As expected based on the configuration of the streams, we observe that the bitrate grows linearly with the number of participants for both simulcast and non-simulcast, but at a significantly lower rate when simulcast is enabled. The benefits of

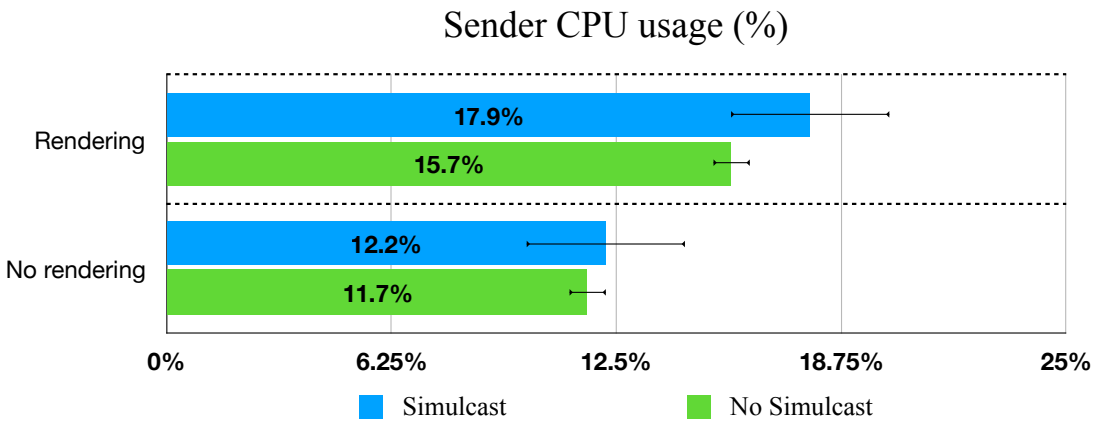


Figure 5.5: *Sender CPU usage. We have two parameters in this experiment: simulcast and rendering the local video. The error bars represent one standard deviation.*

simulcast are already substantial in a 3-way conference. In particular, without simulcast the receiver needs to be able to handle $5\,\text{Mbps}$, compared to $2.8\,\text{Mbps}$ with simulcast, which is 45% less. For a conference of 15 participants, without simulcast the receiver needs to be able to handle $37.5\,\text{Mbps}$ and only $4.8\,\text{Mbps}$ with simulcast.

Next, we examine the CPU usage on the receiver. The summarized results are shown in figure 5.6. We observe that, when simulcast is disabled, CPU usage at the receiver grows by approximately 5.5% per participant. With simulcast enabled, it only grows by 1% per participant, resulting in 50% less CPU usage for a conference of 9.

This becomes important when the receiver is also a media sender, because the *webrtc.org* engine's CPU budgeting can limit the encoding resolution when it detects the system is overloaded.

5.4.7 Server Load

We examine both the incoming and outgoing bitrate on the SFU. Figure 5.7 summarizes the results. We see that as expected simulcast adds to the incoming bitrate, due to the extra streams each sender transmits. This overhead is equal to roughly $0.65\,\text{Mbps}$ per sender, or 26%.

For the outgoing bitrate, simulcast offers a significant advantage. In both cases the bitrate grows quadratically with the number of participants, but with simulcast, the quadratic coefficient is the bitrate of the lower quality streams ($0.15\,\text{Mbps}$) instead of

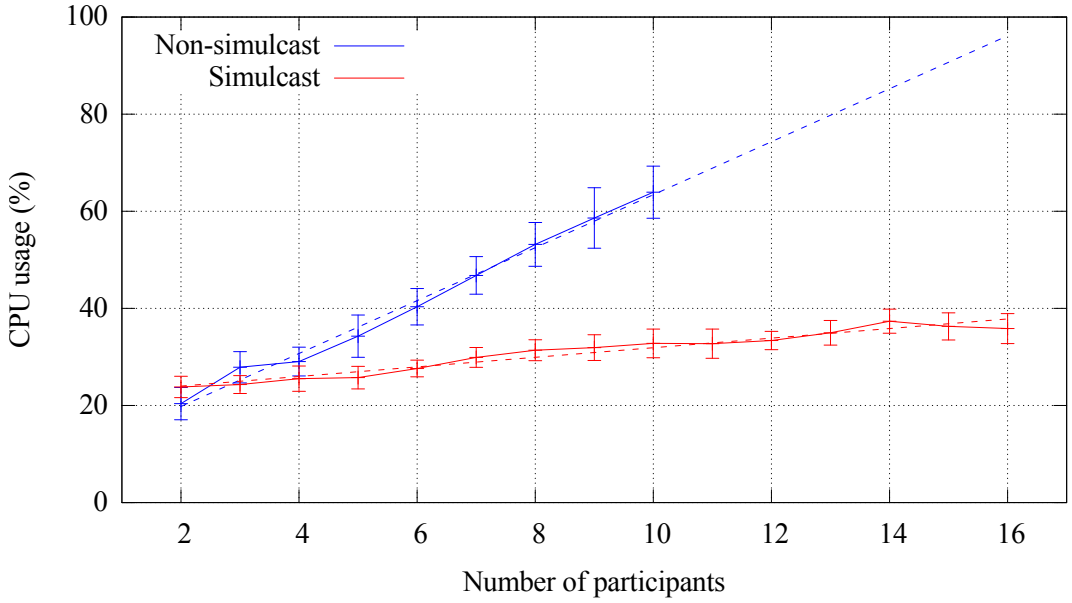


Figure 5.6: *CPU usage at the receiver as it grows with the number of participants. The dashed lines show the best linear fit, and the error bars represent one standard deviation.*

the high-quality streams (2.5 Mbps). As a result, for a conference of 10 participants (the most we could achieve without simulcast), the value drops from 228 Mbps to 39 Mbps with the use of simulcast, a decrease of 83%.

For a conference with two participants, the combined network usage is higher for the simulcast mode (11.5 Mbps vs. 10 Mbps), because of the overhead of the unused streams. In this case, simulcast offers no advantages and should not be used. For three participants the combined network usage is lower with simulcast (18.2 Mbps vs. 23.4 Mbps), and the difference grows with the conference size.

Figure 5.8 shows the CPU usage results for the non-simulcast mode and the simulcast mode. The values are the average of 60 samples taken over a one minute period during a conference with the given size. For a conference with 10 participants, the simulcast mode uses 50% less CPU than non-simulcast.

When taking both CPU usage and bitrate into account, we observe that simulcast has a lower efficiency: 8.5 vs 14.9 Mbps per $1\%CPU$. This means that while the total number of conferences that a single server instance can handle is increased with simulcast, the total bitrate that it can handle is decreased.

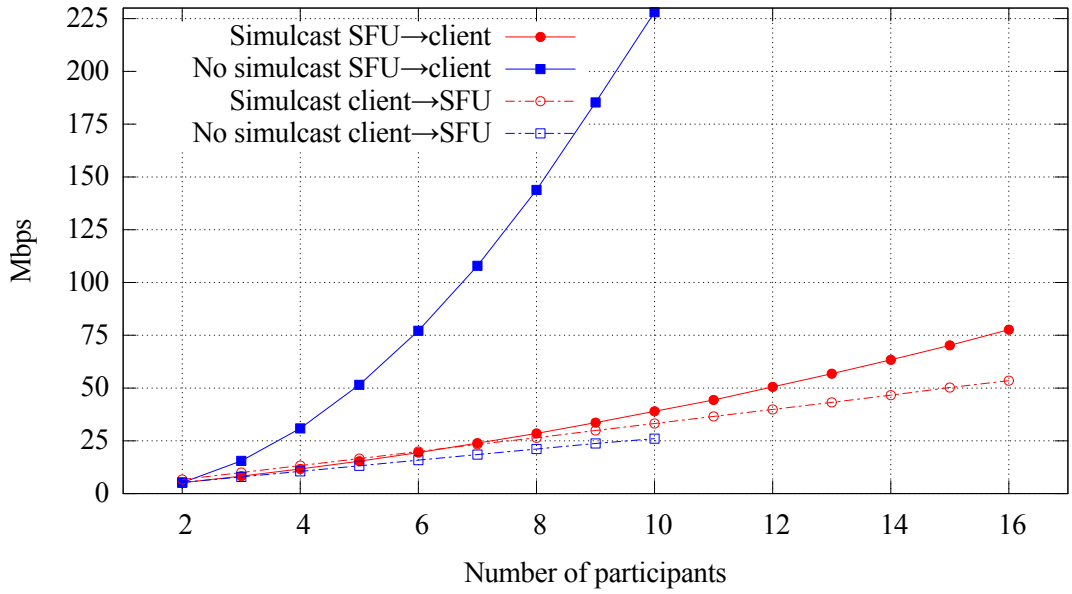


Figure 5.7: *Network usage at the server as it grows with the number of participants. The dashed lines show the incoming bitrate, and the solid line shows the outgoing bitrate. The outgoing bitrate with simulcast grows quadratically, but with a small coefficient.*

5.5 Conclusion

In this chapter, we performed various experiments to evaluate different aspects of a sample simulcast implementation and its impact on senders, servers and receivers in a WebRTC-based video conferencing environment. Our results show that, for senders, simulcast causes an average increase of 26% in outgoing bitrate, which was expected due to the additional encodings that are transmitted. However, we also found that due to the various optimizations implemented in the *libvpx* encoder, the CPU overhead for senders is insignificant.

We also showed that when endpoints receive and render a stream from a simulcast sender, its quality is consistently worse than that of an equivalent resolution non-simulcast stream (i.e., on average simulcast PSNR scores are $0.38\,\mathrm{dB}$ lower than those for non-simulcast). The difference is due to the optimizations mentioned in the previous paragraph and we estimate that, on average, one can compensate for it by increasing the target bitrate at the sender by roughly 12%; alternatively, one can disable the optimizations and achieve the same PSNR values at the cost of increased CPU usage. From a bandwidth utilization perspective, using simulcast greatly reduces the amount of incoming traffic on the receiver, almost halving it in a three-way conference and decreasing

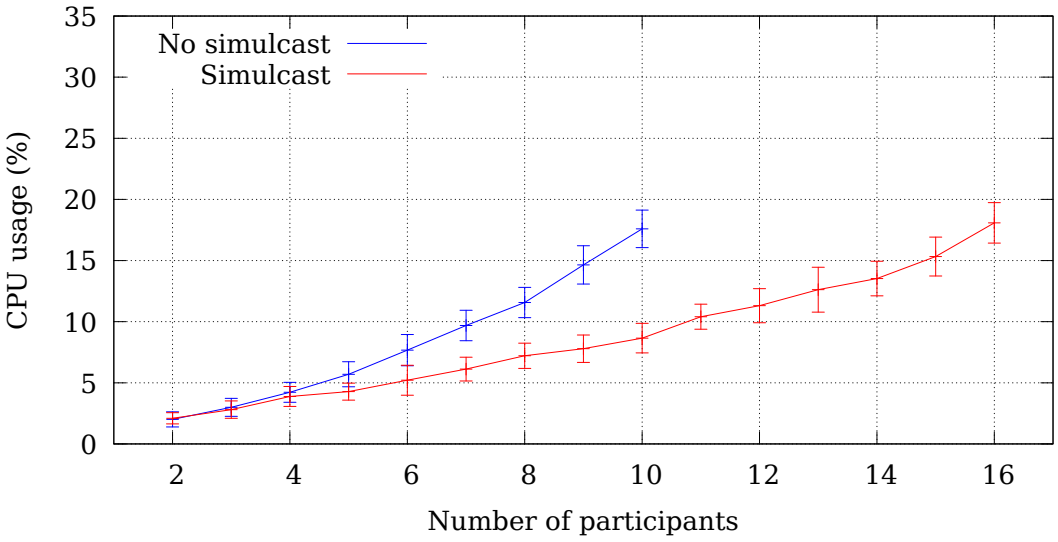


Figure 5.8: *CPU usage on the server as the number of participants in the conference increases. The error bars represent one standard deviation.*

it by more than 80% in a ten-participant conference. In addition to encoding specifics, we found that the overall receiver-side experience can be impacted by the process of switching between different streams, as it requires a refresh of the decoding context (a keyframe). We demonstrated how to mitigate this effect by tracking dominant speaker changes and using them to anticipate stream switches.

Similarly, we found significant gains on the SFU servers themselves when using simulcast. The experiments show that, when using simulcast, the SFU consumes less bandwidth for roughly equivalent quality in conferences with three or more participants. The gain is negligible for three participants, but becomes very significant for larger conference sizes: it reaches 83% for a conference with 10 participants. On the flip side we found that simulcast increases incoming bandwidth utilisation by 26% regardless of conference size.

Incidentally, we outlined one pitfall we consider likely to impact simulcast implementations. Specifically, we showed that simulcast's positive performance impact on SFUs is significantly lower if the decision to not forward certain streams is made only after they have gone through decryption, data copying, encryption, and the rest of the usual SFU processing.

Future Work

Another technique which can be used to allow an SFU to access different encodings of a stream is Scalable Video Coding (SVC [28, 20]). With SVC, the sender produces a single stream with multiple interdependent layers, which can be combined additively to increase different quality aspects. The various layers depend on each other, forming a hierarchy. The lowest layer, i.e., the layer that does not depend on any other layer, is called the base layer and offers the lowest quality level. Each additional layer improves the quality of the signal in any one of the three dimensions: spatial, temporal, or signal-to-noise ratio. When used for conferencing, the SFU can select which layers to include in the output stream for each receiver.

SVC is part of the MPEG-DASH specification [4] and is widely used for adaptive streaming of live and recorded content. One major advantage that it offers is efficient storage of multiple quality levels, which is not directly relevant to the conferencing case.

Simulcast and SVC have been compared for use in IPTV [8] and adaptive streaming [41], but not for real-time conferencing, though this topic certainly warrants further exploration. Simulcast obviously has overhead in terms of the bitrate between the sender and the SFU, but it is not propagated to receivers because the streams are independent. With SVC, the encoding overhead [21] is incurred in the SFU-to-receiver links as well, so the overall overhead varies with conference size. Performing such a comparison is be challenging because of codec support. Simulcast can be used with any codec, although the low encoding complexity overhead observed in our experiments depends on techniques specific to VP8. Scalable Video Coding must be supported by the codec. Currently, there are no complete implementations that support both simulcast and SVC.

Another topic worthy of further exploration is SFU-driven endpoint selection, that is choosing which endpoint(s) to forward in high quality at the SFU instead of the receiving endpoint. This eliminates the need for the preemptive keyframe requests discussed earlier and virtually eliminates the stream switching latency. A hybrid approach must be taken to also support user-initiated selection.

This concludes our discussion on simulcast. In the next chapter, we look at reducing the total server traffic in a service by using peer-to-peer connections whenever possible.

Chapter 6

A Hybrid Approach: Selective Forwarding Unit and Full-mesh

The use of LastN and simulcast greatly reduce the required bandwidth for a conference in an SFU based system. However, there are two characteristics for which the full-mesh model still outperforms the SFU model. The first is the latency, which is lower because endpoints can connect directly. The second is the network traffic for the servers, because they are not always in use.

By far, the most significant parameter affecting resource use is conference size. Large conferences are not possible with a full-mesh, though small ones are and might perform better than an SFU. Thus, a service can benefit if we choose an architecture based on the size of the conference. The simples way of achieving this is to create the conference with the appropriate configuration from the start. However, in practice, the conference size is rarely known in advance. Additionally, conference size can change over time, so the best option may involve using different architectures during different parts of a conference. For these reasons, we have chosen to switch the architecture dynamically during a conference. Specifically, we use a full-mesh architecture for conferences of size P or less and switch to an SFU when the size reaches $P+1$. Conversely, we switch back to full-mesh when an endpoint leaves and the size drops down to P. This leaves us with a choice for the parameter P.

We chose to use $P = 2$; that is, we only use full-mesh for the trivial case of two endpoints in a conference. Other services, such as Google Hangouts, do the same. There are a few reasons for this decision. First, we analyzed the conference size on the *meet.jit.si* service and found that over 77% of the time is spent in one-to-one conferences, meaning that $P = 2$ should be sufficient to gain most of the benefits of reduced traffic.

Second, since we started with an SFU system, we wanted to ensure that introducing the full-mesh mode does not have any adverse effects in any situation. We cannot guarantee this when $P >= 3$. For example, in situations with limited upstream bandwidth, an endpoint benefits from an SFU because it only needs to send one copy of its stream. Additionally, if a direct connection is not possible between a pair of endpoints, they need to use a relay server. In this case, the full-mesh solution uses two or more relay servers, leading to strictly increased traffic as compared to an SFU.

The third reason is the need for seamless switching between the two modes, and the practical desire to keep the overall complexity of the system low. Users should not experience any disruption when the conference changes from an SFU to full-mesh or vice versa. All endpoints must agree on which mode to use, and do the switch at nearly the same time, otherwise, users experience a disruption. With two endpoints, we found that we can achieve seamless transition by switching to the direct connection based on the ready-state of the transport layer connection (ICE). With three endpoints this is insufficient; endpoint A must first consider the ready-state for the connection between endpoints B and C before switching to direct connections.

We added support for switching to a full-mesh in the Jitsi Meet system. Because it only supports $P = 2$, we refer to this feature as "peer-to-peer for one-to-one" for brevity we use "P2P" instead of "peer-to-peer". The implementation is detailed in Section 6.1. We then enabled the feature in the *meet.jit.si* service and compared its perfomance to the SFU-only mode in terms of quality of service for users and server traffic. We present the results in Section 6.2.

6.1 Implementation

6.1.1 Topology

The Interactive Connectivity Establishment (ICE [64]) protocol WebRTC uses is designed for NAT traversal, that is establishing a direct connection between two endpoints even in the presence of Network Address Translation (NAT). Endpoints contact a STUN [65] server to determine their translated address, and advertise it as a potential address candidate. However, in the presence of firewalls or certain types of NATs a direct connection is not possible. For such cases, ICE uses a tunnel through a TURN [50] relay server. A direct connection is preferred whenever possible: TURN candidates have lower priority in the ICE check list.

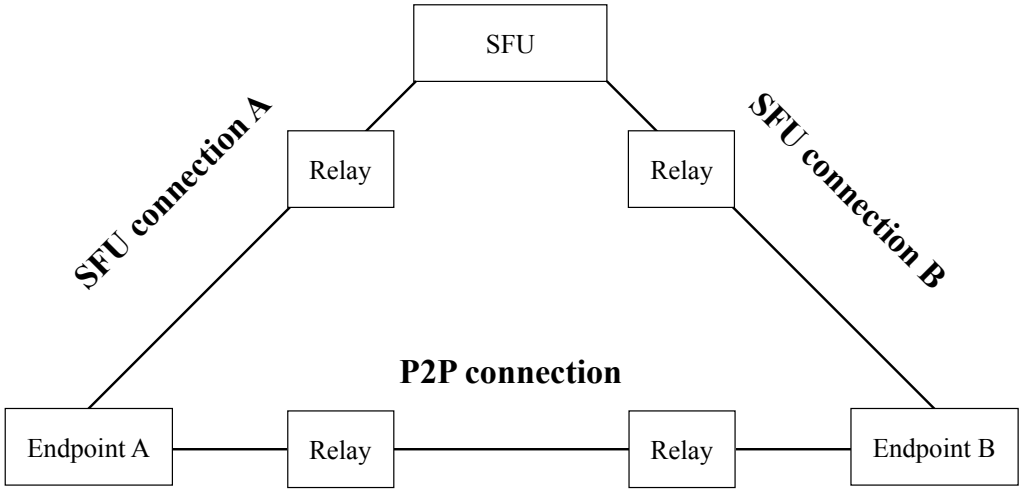


Figure 6.1: *The topology used for the SFU and peer-to-peer modes. The relays are optional and only used when a direct connection fails. The SFU connection is always maintained, while the P2P connection is only established when there are two endpoints.*

WebRTC does not make a distinction between a server and another browser for the remote peer; it uses ICE in both cases. Since we control the SFU server, we can ensure it is not behind a restrictive NAT making a direct UDP connection possible in the vast majority of cases. Some restrictive firewalls block UDP traffic; to support endpoints in such networks we provide TURN relay servers. Endpoints establish a TLS connection to the relay server, bypassing most firewalls because it is indistinguishable from regular HTTPS traffic, and tunnel the ICE and multimedia traffic through the relay. The full topology for the SFU mode is shown in figure 6.1.a. The relay server can be co-hosted on the same machine as the SFU, but we chose to use separate servers as it allows us to deploy them in different geographical regions. We observe that in normal operation only 0.2% of endpoints using *meet.jit.si* connect to the SFU via a relay, the rest connecting directly over UDP.

For peer-to-peer connections the endpoints also use ICE, but the probability of failure to connect directly is higher because there may be NATs on both sides. They use the same TURN relay servers with the additional option to create the tunnel over UDP. In rare cases two relays may be necessary. Figure 6.1.b shows the topology.

6.1.2 Features

With the SFU system we use LastN and simulcast as described in chapters 4 and 5. Senders transmit three encodings of their stream to the SFU. Typically, these are $720p$, $360p$ and $180p$ versions encoded at up to $2.5Mbps$, $0.5Mbps$, and $0.15Mbps$, respectively. Simulcast requires the VP8 codec, as at the time of this writing it is the only codec supported by WebRTC clients.

For the peer-to-peer mode simulcast is counter-productive. Since the receiver only makes use of one of the encodings, it is best to allocate all available bandwidth to it. Therefore, we disable simulcast for the peer-to-peer connection, allowing the use of any codec supported by both endpoints, regardless of simulcast support. We chose to use H264 because of the wider hardware decoding and encoding support in various devices. Regardless of the selected codec, senders transmit a single $720p$ at up to $2.5Mbps$.

Another difference between the two modes is that we use Forward Error Correction [47, 30] for the peer-to-peer connection, while the SFU does not currently offer support for FEC.

6.1.3 Dynamic Switching

Our goal is to make the switch between the SFU connection and the direct connection seamless. That is, the user should not experience any disruption when the switch happens. Ideally the user should not even notice this event.

To accomplish this we start with an SFU connection, and maintain it at all times. We switch to the P2P connection opportunistically when we know that there are exactly two endpoints in the conference, and that a P2P connection exists between them. Figure 6.1 illustrates this. Starting the conference with the SFU connection also allows us to minimize the start-up time, because in the majority of cases the connection is established using local *host* [64] candidates and does not have to wait for gathering of STUN or TURN candidates.

Switching to P2P Mode

In WebRTC terms, we use separate `PeerConnection`s for SFU and P2P mode. A conference starts as soon as there are at least two participants. First the signaling server initiates an XMPP/Jingle session [68, 49] with each enddpoint. Each endpoint then creates a `PeerConnection` for the SFU connection. The typical WebRTC offer/answer

takes place over the signaling channel so that the ICE connectivity process starts. Each participant has all of its `MediaStream` [15] objects added to the `PeerConnection`, and the media direction is set to `sendrecv` [78] allowing media playback to begin as soon as the connection is established.

If at any time we transition to a state with exactly two endpoints in the conference (i.e., when the second endpoint joined or the third endpoint left), we create a new `PeerConnection` and initiate a session with the remote endpoint. The session also uses XMPP/Jingle, but this time directly between the two endpoints, bypassing the centralized signaling server. The two endpoints compare their IDs to determine which one acts as the Jingle `initiator`, the other one taking the `responder` role. We add all local `MediaStream` objects to this peer connection as well, and we set the media direction to `sendrecv`.

After the two connections start their connectivity checks, things can happen in two possible flows. If the SFU connection goes to ICE `connected` [12] state then we add the remote SFU `MediaStream` objects to the user interface and a normal conference starts with audio and video in both directions. As soon as the P2P connection succeeds, provided there are still exactly two endpoints in the conference, we replace the remote SFU `MediaStream` objects with the P2P objects and set the media direction for the SFU objects to `inactive`. Doing so keeps the ICE connection running, and as no media is exchanged bandwidth is saved. The user interface now renders audio and video from the P2P connection.

On the other hand, if the P2P connection is established first, we add its remote `MediaStream` objects to the user interface immediately and set the media direction of the SFU connection to `inactive`. When the SFU `MediaStream` connection is established we take no action; that is, we keep it as inactive and do not display any of its streams.

Switching to SFU Mode

We switch back to the SFU connection in two cases: when a third participant joins the conference and when the ICE connection for the P2P mode enters the `failed` state [12]. First, we adjust the media direction of the SFU connection to `sendrecv`, then we replace the P2P `MediaStream` objects in the user interface with the corresponding objects from the SFU connection. Finally, we close the P2P `PeerConnection` (if we need to switch to P2P again later, we create a new one).

To prevent unnecessary switching between the two modes when there are frequent changes in conference sizes[1], we take a threshold approach. Attempts to re-establish the P2P connection only happen after a delay above a threshold of 5 seconds from the last conference size change event (e.g., from three endpoints to two).

6.2 Experimental Evaluation

To quantify the effects of the P2P mode we perform experiments and measurements on the *meet.jit.si* service.

The service runs on the Amazon Web Services [2] (AWS) cloud and is distributed across four regions: N. Virginia (us-east-1), Oregon (us-west-2), Ireland (eu-west-1), and Australia (ap-southeast-2). In each region, we have a TURN server and a fixed number of "shards" used for failover. Each shard contains a signaling server and an autoscaling group of at least two SFUs for load balancing. If there is a surge of conferences on a specific shard, we increase the number of SFUs in that shard. If an SFU fails, we move the ongoing conferences to another SFU in the same region, and if a whole shard fails, we move the conferences to another healthy shard.

Each SFU and TURN server runs on an AWS `c4.xlarge` instance[3] with 4 vCPUs, 7.5 GiB of RAM, and high network performance. The infrastructure is behind a cluster of geographically distributed HAProxy[4] instances. When a client attempts to join a conference room, it contacts a HAProxy, which checks whether or not this is an existing room. If it is a new room, the HAProxy assigns it to the shard closest to the geographic location of the user. If the room already exists, it retrieves the shard hosting the room and provides the client with the appropriate configuration to connect to that shard.

6.2.1 Infrastructure Resource Use

One of the main benefits of using direct connections for one-on-one conversations is that traffic is offloaded from the server-side infrastructure. We use the following metrics to quantify this:

- Average per-participant bitrate

[1] This commonly happens when one endpoint refreshes the page in the browser, resulting in a re-join.
[2] https://aws.amazon.com/
[3] https://aws.amazon.com/ec2/instance-types/
[4] http://www.haproxy.org

- Total data transferred
- CPU usage and system load
- Thread count

Note that the effects of introducing the dynamic switching extension depend significantly on the fraction of one-on-one conferences on the service. Obviously, if there were no one-on-one calls, there would be no effect.

Because we wanted to measure the impact on the whole service, it was not possible to enable the feature for only a subset of the conferences. Instead, we compare metrics for a period of two weeks just prior to the introduction with the same metrics taken in the two weeks following the introduction of the feature. During the first period all calls, including one-on-one, go through the SFU. During the second period one-on-one calls go peer-to-peer if possible, otherwise going through the SFU; TURN relays were disabled in order to observe the net effect of peer-to-peer offloading. Calls with three or more endpoints always go through the SFU.

We chose a period of two weeks because it is long enough to produce sufficient data while remaining unaffected by daily and weekly patterns, and short enough to not be affected by other changes in the environment (like other features being introduced and the growth of the service itself). During the test period the load of the machines never exceeded the threshold needed to scale it up; therefore, the number of servers for both tests was always the same.

The first period includes 23470 conferences with a combined duration of 689273 minutes, and the second one includes 23320 conferences with a combined duration of 696311 minutes. The differences between the two periods are within 1% ($+0.6\%$ in number of conferences and -1.0% in combined duration), so we compare the rest of the metrics directly without applying normalization.

The distribution of conference sizes is was follows: $77.8\%$with size 2 (one-on-one), $12.2\%$with size 3, $5.4\%$with size 4, $2.0\%$with size 5, and the remaining $2.6\%$with size 6 or more (see 8.4 above). There are no significant differences between the two periods. We derive this distribution by sampling the active conferences once per minute.

The total data transferred towards the servers during the two periods is $7.1TB$ and $3.6TB$ respectively, a reduction of 49%. From servers to clients it is $6.0TB$ and $2.7TB$ respectively, a reduction of 55%. The combined reduction is 52%. Given the number of participants, this translates to an average combined bitrate of $1.1Mbps$ and $0.56Mbps$ per participant, respectively for the two periods.

Note that this reduction of 52% does not match the $77.8\%$of one-on-one conferences. This is in part due to failure of some endpoints to connect directly, and in part because the resource consumption of conferences grows faster than linear with the conference size. That is, a single conference of size N is not equivalent in terms of bandwidth to $N/2$ conferences of size 2.

The average CPU usage on the SFUs is 4.7% and 3.2% respectively during the two periods; a reduction of 32%. The average memory usage is $3740MB$ and $2420MB$; a reduction of 35%.

The average number of Java Virtual Machine (JVM) threads used by the SFU process is 392 and 378 respectively, hardly any reduction. This is because clients are connected to the SFU at all times and most of the necessary threads on the SFU are already allocated (though not used as much, as we can see from the CPU usage result). The average system load as reported by the Linux `top` command is 0.28 and 0.15 respectively, a reduction of 46%.

The average round trip time measured by the SFUs is $110ms$ and $114ms$, respectively, for the two periods. This shows that the decreased load and network traffic on the servers does not result in a reduced RTT, at least not at the low load levels we observed.

6.2.2 Quality of Service

Another potential benefit of using direct connections for one-on-one calls is higher quality of service (QoS). To quantify this we measure RTT, call establishment time, packet loss, and download bitrate. We collected the data from the endpoints via the WebRTC statistics API [6].

We took measurements from the *meet.jit.si* service for five weeks, from a total of 58489 conferences. During this period, TURN was enabled to compare the performance of TURN-relayed calls to SFU-routed calls. This period did not coincide with the infrastructure measurements sampling period, to avoid skewing of the infrastructure results due to having TURN enabled.

We only examine UDP-based calls because bandwidth estimations over TCP are problematic as TCP runs its own congestion control. Additionally, TCP connections only work with the SFU, since browsers do not support receiving incoming TCP connections. Therefore, TCP is not relevant in our comparative evaluation.

We examine the us-east-1 and eu-west-1 regions separately because we found that different regions have very different characteristics, partly due to differences in the ISP

peering agreements [55]. We only consider same-region calls, i.e., calls where all of the participants are in the same region. We chose these two regions because they host the majority of conferences. The other regions were not included in our analysis due to the small number of calls.

RTT and Call Establishment Time

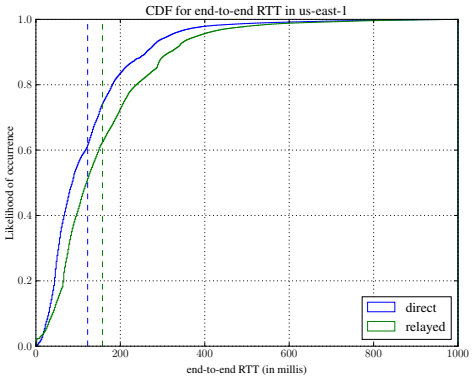
For the RTT and call establishment time measurements, we distinguish two groups: direct and server-relayed calls, i.e., calls that are either relayed through TURN or routed through the SFU.

Packet loss is around 0.5%, and is independent of the group. These low levels of packet loss are a consequence of using the Google Congestion Control (GCC [16]) algorithm, which uses delay measurements to reduce the sending rate before an actual congestion and packet loss occur.

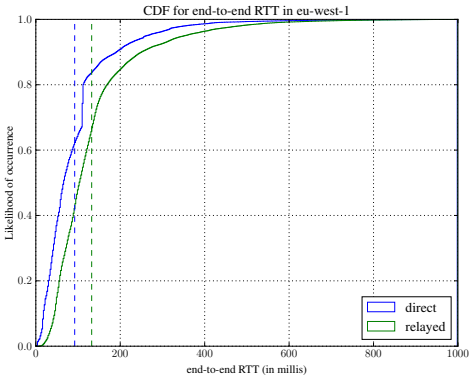
Round-trip time Next we look at the RTT for relayed and direct calls. Figure 6.2 summarizes the results. On average, calls in the US have $27.5ms$ longer RTT than EU calls. This difference may be due to the longer average distances between endpoints in the US, and differences in the network infrastructure.

Direct calls in the EU have an average RTT of $92ms$ while the average for relayed calls is $132ms$. This $40\,\mathrm{ms}$ difference may be due to an increased number of hops between the peers, due to the use of a server. In the US, direct calls have an average RTT of $122ms$ vs. $157\,\mathrm{ms}$ ($35\,\mathrm{ms}$ longer) for relayed calls.

Connection establishment time We define the connection establishment time as the duration from when the gathering of ICE candidates begins until ICE transitions to the `connected` state. The values that the two endpoints measure differ based on which endpoint initiates the session, and which endpoint has the ICE `controling` role (that is, it *nominates* the selected candidate pair). In our case, the initiator always assumes the *controling* role, so we group the results by initiator/responder. Figure 6.3 summarizes the results. The average connection establishment duration in the EU is $1700\,\mathrm{ms}$ and $1950\,\mathrm{ms}$ in the US. This 12% increase fits our observations about the generally longer RTT in the US. When comparing direct vs. relayed calls, we observe a slight decrease in the establishment time for direct connections.

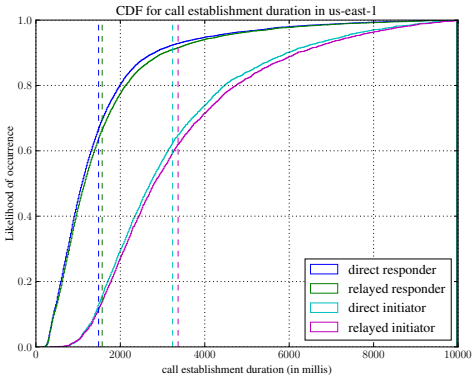


(a) Users in the US (the us-east-1 region).

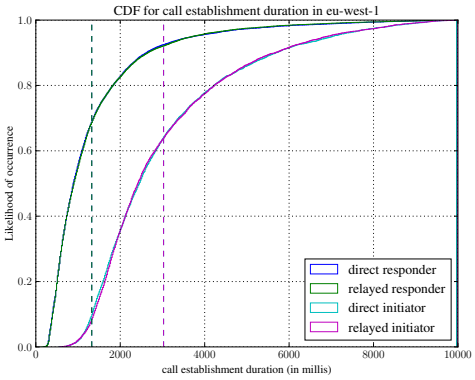


(b) Users in Europe (the eu-west-1 region)

Figure 6.2: *The cumulative distribution of round-trip-time for direct vs. relayed connections. The dashed lines indicate the means. Direct calls have an overall lower RTT than relayed calls.*



(a) Users in the US (the us-east-1 region).



(b) Users in Europe (the eu-west-1 region)

Figure 6.3: *The cumulative distribution of connection establishment time for direct vs. relayed connections, further split by the endpoints' initiator/responder role. The dashed lines indicate the means. For all cases, direct calls have a slightly lower establishment time than relayed calls.*

6.3 Conclusion

In this chapter, we described an extension of an SFU-based system, which switches to a full-mesh topology when a conference has only two endpoints. We detailed the im-

plementation using WebRTC. We then introduced the extension to the *meet.jit.si* service and evaluated the proposed system in terms of infrastructure cost and user experience.

Our results quantify a significant reduction on the total infrastructure costs. Furthermore, on average, direct calls have significantly lower latency Throughput efficiency is severely degraded when the call is routed through an SFU because of the usage of group-mode specific features such as simulcast and RTCP termination. As a result of using the GCC algorithm, the packet loss is negligible regardless of the topology. The call establishment time remains practically the same for direct and server-relayed calls.

Chapter 7

Performance Optimizations for Selective Forwarding Units

When it comes to video conferencing servers, Selective Forwarding Units are usually expected to offer better computation efficiency than MCUs because they do not decode or encode audio or video. However, their good performance should not be taken for granted, as it depends on the specific software implementation.

An SFU is a complex software system which has to be engineered with performance as a goal. Poor performance can lead to degraded user experience, as well as inefficiency and higher service operating costs.

In this chapter, we explore the factors affecting performance and propose ways to improve them. We start by describing the function of an SFU, and the Jitsi Videobridge implementation in particular in Section 7.1. In the next section (7.2, we propose an efficient algorithm for selective forwarding, and we evaluate its effects on CPU usage. Finally, in Section 7.3 we explore the topic of memory management, specifically in systems with automatic garbage collection. We propose a technique to reduce the strain on the garbage collector, develop a methodology to evaluate its effects, and perform experiments to find the most suitable configuration to use for an SFU.

7.1 Introduction to Selective Video Routing

An SFU's primary task is to route RTP packets, and it is what constitutes most of the workload. The task consists of the following steps:

1. Receive a packet from a source endpoint via a network socket.
2. Process the packet in a context specific to the source endpoint (the receive pipeline[1]).
3. Select the set of endpoints to receive the packet, and for each of them:
   (a) Process the packet in a context specific to the destination endpoint (the send pipeline).
   (b) Send the packet to the destination endpoint via a socket.

This process is illustrated in figure 7.1. Both the send and receive pipelines may modify some of the header fields in order to produce a continuous stream for each destination, so it may be necessary to create copies of the packet.

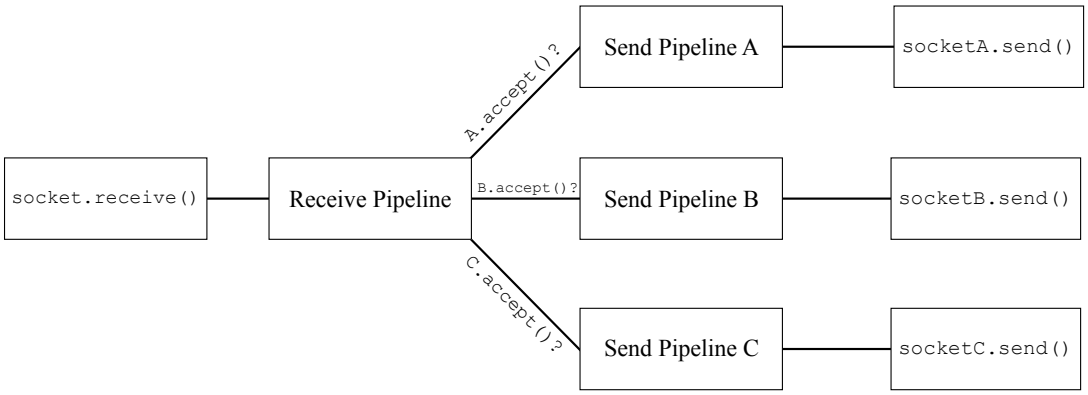


Figure 7.1: *The primary task of an SFU is to route packets from a source endpoint to a set of destination endpoints. The* `accept()` *check implements* selective *forwarding.*

There are numerous secondary tasks that SFU implementations perform, but they usually do not contribute significantly to their performance. These include dominant speaker identification, connectivity establishment (ICE and DTLS), signaling and control logic, keeping track of stream statistics and overall statistics, self-health monitoring, bandwidth estimation, and logging.

[1]The "receive" and "send" pipelines are named from the point of view of the SFU.

Jitsi Videobridge[2] is written mostly in *Java*, with certain components written in *C* specifically to improve performance. Some of the underlying libraries use *Kotlin*, however as it is translated to JVM bytecode their characteristics are similar to *Java*.

It uses the *ice4j* library[3]: an implementation of the Interactive Connectivity Establishment protocol [64] that offers an abstraction layer with a virtual `DatagramSocket` interface for receiving packets from an endpoint. Packets are received using a synchronous `receive()` call into a buffer (a Java `byte[]`) managed by the application.

7.1.1 Receive Pipeline

RTP packets enter the application through an *ice4j* socket. Once received from a socket, a packet passes through the receive pipeline associated with the remote endpoint. The pipeline consists of a series of steps, each performing a specific action and possibly modifying the packet. It consists of the following steps:

1. Basic statistics tracking. Here we count the combined number of packets and bytes received.
2. RTP/RTCP pre-parsing. Here we recognize the type of the encrypted packet: RTP or RTCP, and read of some basic fields (SSRC, sequence number) required for the next step.
3. SRTP or SRTCP [11] authentication and decryption. Here we perform the SRTP "unprotect" procedure. First, we compute a hash for the packet's contents and compare it to the packet's authentication tag. If the results do not match, or if we have already received a packet with this sequence number, we discard the packet. Otherwise, we perform decryption of the packet's payload. This step modifies the contents of the packet's payload.
4. Audio level extraction. This step applies only for audio packets. We read the RTP header extension for the audio level [44] and fire an event. The audio level is then used for dominant speaker identification.
5. Per-stream stats tracking. Here we calculate the statistics for individual streams, including packet rate, bitrate, packet loss, and jitter.

[2]https://github.com/jitsi/jitsi-videobridge
[3]https://github.com/jitsi/ice4j

6. Silence discarding. Here we discard audio packets that contain only silence and repair the resulting stream by adjusting the packet's sequence number and timestamps to hide previously discarded packets. This step modifies the RTP headers.

7. RTX [62] and padding [70] termination. Here we repair the stream by removing the RTX encapsulation of re-transmitted packets and discard packets containing only padding (and no payload). This step modifies the RTP payload and headers.

8. Video parsing and bitrate calculation. Here we perform further parsing of the video packets, reading specific fields such as a frame sequence number, temporal layer ID, keyframe flag, etc. We read this information either from the payload (e.g., the VP8 Payload Descriptor [86]), or from the framemarking RTP header extension [19].

9. Stream repair. Here we monitor the stream for lost packets and send requests for retransmission using RTCP NACK packets [83] if necessary.

10. RTCP termination. Here we handle all incoming RTCP packets. This includes retransmission requests, bandwidth estimation, and keyframe requests.

After packets go through the receive pipeline, they pass to a `Conference` object, which is responsible for their further handling in the context of a specific conference.

7.1.2 Selective Forwarding

The `Conference` object implements selective forwarding. That is, for each received packet and each remote endpoint, it decides whether to forward the packet to the endpoint. The decision considers the following factors: LastN and the list of speakers in the conference, congestion control and simulcast (which determine the encodings currently enabled for each endpoint), and whether the endpoint supports the given media type.

A video packet is only forwarded to a destination endpoint if:

- The destination endpoint supports audio or video (depending on the packet media type).
- The source endpoint is in the LastN set.
- The packet's encoding is currently enabled for the destination endpoint. Simulcast uses three different encodings for each stream, and at any time at most one of them

Algorithm 1 Algorithm for selective forwarding.

```
1: procedure FORWARD(packet, sourceEndpoint)          ▷ Handles an incoming packet.
2:     handler ← null
3:     for endpoint in endpoints do
4:         if endpoint == sourceEndpoint then
5:             continue
6:         if ACCEPT(packet, sourceEndpoint, endpoint) then      ▷ This step must not
   modify the packet.
7:             if handler! = null then
8:                 handler.send(clone(packet))      ▷ The packet is cloned only if necessary.
9:             handler ← endpoint
10:    if handler! = null then
11:        handler.send(packet)                      ▷ Use the original packet, not a clone.
12:    else
13:        free(packet)                              ▷ No endpoint accepted the packet.
14: procedure ACCEPT(packet, sourceEndpoint, destinationEndpoint)
15:    mediaType ← getMediaType(packet)                                  ▷ Audio or video.
16:    encoding ← getEncoding(packet)                ▷ The packet's simulcast encoding.
17:    if mediaType == AUDIO then return true
18:    else if !supportsVideo(destinationEndpoint) then return false
19:    else if !isInLastN(sourceEndpoint) then return false
20:    else if !isEncodingEnabled(encoding, destinationEndpoint) then return false
21:    else return true
```

is enabled. Encodings are disabled for congestion control when the bandwidth estimation indicates that there is insufficient bandwidth for all encodings.

Because packets forwarded to an endpoint are always modified in the send pipeline (see the next section for details on this pipeline), they must be cloned[4]. A naïve implementation passes a cloned packet to each destination endpoint, discarding unwanted packets in the send pipeline. This results in unnecessary cloning for endpoints that do not accept the packet, and for the last endpoint that accepts it since it could just use the original. We use algorithm 1 to avoid these unnecessary operations.

7.1.3 Send Pipeline

We have a separate send pipeline for each endpoint. It handles packets coming from all other endpoints in the conference, and consists of the following steps.

[4]To clone a packet, we create a new buffer and copy the packet's contents.

1. Simulcast. Performs simulcast related logic, such as rewriting the SSRC, Timestamp, and Sequence Number, producing a continuous stream even when the accepted encoding changes. This is also where the internal simulcast state is updated and the enabled encodings are changed if necessary.

2. Packet caching. We save outgoing packets in case they need to be retransmitted later. Since packets are modified later in the pipeline, we create and save a copy of each packet.

3. Statistics tracking. Here we track statistics for individual streams, such as number of packets and bytes sent, and packet loss reported by the receiver.

4. Transport-wide congestion control [29] tagging. Here we tag outgoing packets with a sequence number common for all streams (used for congestion control). This step modifies the RTP headers of packets.

5. SRTP or SRTCP [11] encryption and tagging. Here we perform the SRTP "protect" procedure. The packet's payload is encrypted and an authentication tag is computed and attached. This step modifies the RTP payload of packets.

6. Basic statistics tracking. Here we count the combined number of packets and bytes sent.

7.2 Optimizing Computational Performance

7.2.1 Evaluation and Optimizations for Simulcast

In Chapter 4, we observed that without simulcast CPU usage is proportional to total bitrate. However, with the addition of simulcast there is an increased potential for CPU usage not to scale linearly. Some encodings are received and processed (contributing to CPU usage) but are not forwarded (and do not contribute to the network bitrate).

While it is not possible to entirely eliminate the processing of unused encodings, we found significant improvements in performance when applying two optimizations. First, if an encoding is not selected by any receiving endpoint, we do not perform decryption of the payload (though we still do the authentication step). We can do this because the information in the headers is not encrypted. The second is the implementation of the selective forwarding algorithm (see section 7.1.2) instead of the naïve approach of processing packets.

We compared the performance before and after these optimizations with the same testbed and procedure used in Chapter 5. We can see a consistent decrease in CPU usage, which is more significant for large conferences (since more of the high-bitrate streams are non selected). For a 16 participant conference, the CPU usage decreased by 40%. Figure 7.2 shows the full results.

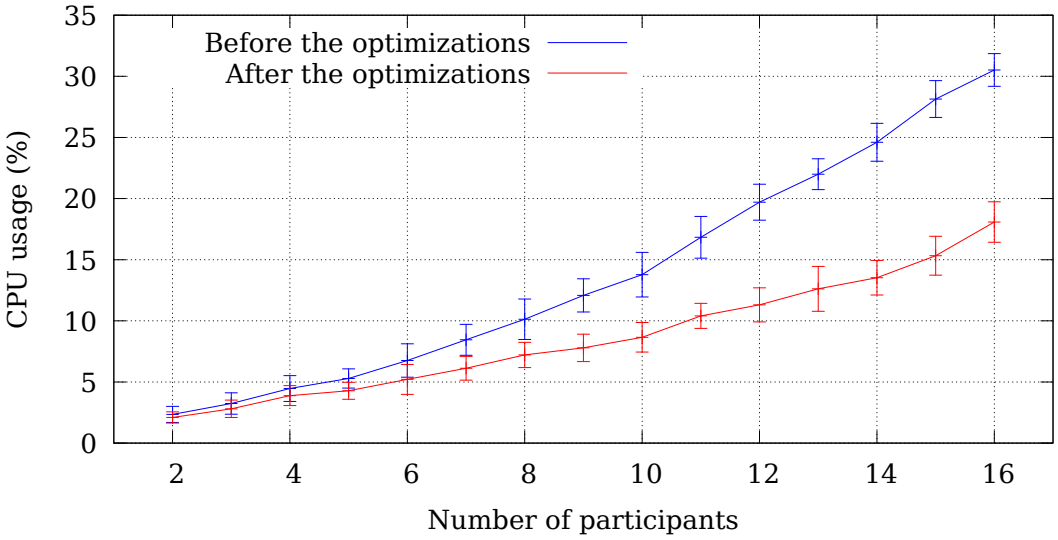


Figure 7.2: *CPU usage before and after the simulcast optimizations. The error bars represent one standard deviation.*

7.2.2 Profiling

To better understand which codepaths contribute most to the CPU usage, we performed sampling profiling on a test machine while generating representative load. That is, we created test conferences with sizes matching the distribution observed in the production service (see Chapter 6).

We observed the following results. SRTP transformations (encryption an decryption) account for 33% of the processing time, network operations such as sending to or receiving from a socket 31%, internal JVM operations 13%, and various other RTP handling such as parsing, modifying headers, etc. 5%. Bandwidth estimation and RTCP related functions account for 5% of time, caching 2%, simulcast-related logic 9%, and all other functions 3%.

We see that operations directly proportional to bitrate (network ops and SRTP) account for over 64% of the processing time. We also see that, as expected, the vast majority of CPU time is spent in the packet pipelines. This is certainly over 84%, and probably above 95% since most of the internal JVM functions are most likely related to the pipeline.

The SRTP calculations have already been optimized using a separate library based on *openssl*. There are different implementations, and the best choice depends on hardware, availability of the *openssl* lib, and JVM version. Because of this we perform a benchmark and select the best algorithm at runtime. Further optimizations in this layer are unlikely to have any significant effect.

The results also confirm that the dominant speaker selection algorithm is efficient, as expected. It is included in the 3% of "other".

We expect that further optimizations of the socket layer in *ice4j*, simulcast, and the bandwidth estimation code may yield good results.

7.3 Optimizing Memory Performance

SFU applications do not require large amounts of memory. Their primary task is forwarding packets, and apart from the memory used to represent the contents of the packet itself, there is little context that is preserved. Once a packet is forwarded the memory is no longer needed. There is one important exception – the packet cache.

The packet cache saves packets so that retransmission requests can be satisfied. Since packets may be modified in the send pipelines, they need to be cached after the modifications[5]. This requires more memory as the outgoing bitrate is usually higher than the incoming. A simple calculation shows that the total amount of memory used by the cache is not much by modern standards. If packets are cached for two round-trip-times, the maximum RTT is 1 second, and the SFU handles $1\,\mathrm{Gbps}$ of traffic (all generous assumptions), it needs just $2\,\mathrm{s} \times 1\,\mathrm{Gbps} = 256\,\mathrm{MB}$ for the cache.

On the other hand, memory allocation throughput can be challenging, particularly on systems using automatic garbage collection (GC). The application continually allocates small blocks of memory[6], uses them for a short period of time and then releases them. This pattern of memory allocation puts a significant strain on the garbage collector.

[5]Technically, they are cached before the SRTP transformation, but after all other transformations.

[6]RTP packets are sized to fit the network PMTU, and *webrtc.org* limits their size to under 1300 bytes.

The penalty and processing overhead of garbage is a concern that is often raised in engineering circles, especially in the context of Java based applications. There is a trade-off between processing overhead and the duration of the pauses necessary to perform GC, and the default algorithms favor low processing overhead at the price of long pauses.

We discovered this during development of Jitsi Videobridge, when seemingly unrelated changes resulted in periodic freezing of the video playback. It turned out that Java application threads were being paused for garbage collection for a relatively long period of time, on the order of hundreds of milliseconds. We started an investigation to understand why the pauses are so long, and how to reduce them.

7.3.1 Garbage Collection in Java

The purpose of Garbage Collection (GC) is to automatically reclaim memory no longer needed by the application. This requires the system to know which objects are currently in use and which can be reclaimed. In Java, this is based on reachability from certain types of objects known as Garbage Collection Roots (GCR). These include active threads, local variables, and static fields. Objects not reachable from any GCRs may be reclaimed.

To avoid incorrectly marking objects as unreachable, the Java Virtual Machine (JVM) has to temporarily pause all application threads. This is achieved with a Stop-The-World (STW) pause, during which no application threads are allowed to run.

The Java Development Kit (JDK) offers several algorithms for garbage collection, with different characteristics. We examine three of them here: **Parallel GC** (PGC, the default), **Concurrent Mark and Sweep** (CMS), and **Garbage First** (G1). More accurately, these are combinations JVM options, which configure different GC behavior. The details of the JDK implementation is out of scope for this thesis, therefore we simplify some concepts. For a more comprehensive discussion, we refer readers to *Java Garbage Collection* [69] handbook by *Plumbr* and the *Java Garbage Collection Basics* tutorial by Oracle[7].

Parallel GC (PGC)

This algorithm is based on the *generational hypothesis*, which is the observation that in most applications objects are in use either briefly or for extended periods, with very

[7]https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

few objects having "medium life expectancy". With PGC, the memory space is divided space into a **young generation** and an **old generation**. New objects are allocated in the *Eden* space, a region of the young generation space with a fixed size. When *Eden* is filled up, this triggers a *Minor GC*.

Minor GC is used to clean the **young generation**. With PGC this is accomplished using a **mark and copy** operation. The **mark** phase traverses the objects in the young generation marking those that which are reachable. It assumes that any object in the **old generation** is a GCR. The **copy** phase copies the reachable objects into a separate **survivor** space, incrementing each object's *generation*. When an object reaches a certain number of generations (15 by default), it is *promoted* to the **old generation**.

After copying the reachable objects out of *Eden*, it is marked as free and may be used again. This does not require explicit handling of unreachable objects, making the operation quick, especially when most of the objects are unreachable. However, it requires separate memory pools in the young generation.

Major GC is used to clean the **old generation**. Usually, the old generation is significantly larger and occupied by objects less likely to be garbage. This operation occurs less frequently than minor GC, and usually takes more time.

PGC uses a *mark-sweep-compact* strategy to clean the old generation. The **mark** phase is similar to that of the young generation. The **sweep and compact** phase defragments the memory in-place. That is, it moves the reachable objects to occupy a contiguous region in the beginning of the old generation space. This operation can be quite slow.

The "parallel" in PGC refers to the fact that the **mark** phases execute on multiple threads in parallel. The **mark** operations trigger an STW pause. There is a **Serial** version of the algorithm, which performs **mark** in a single thread, but we do not consider it here because it offers no advantages.

Concurrent Mark and Sweep (CMS)

The CMS algorithm is designed to avoid long STP pauses while cleaning the old generation. It uses the same separation in young/old spaces, and the same **mark and copy** procedure for the young generation as PGC.

It handles the old generation in a different way. First, it uses free lists instead of a compacting the reachable objects into the begging of the space. Second, it performs most of the **mark** phase concurrently with the application threads. The **mark** phase is divided into three stages: initial mark, concurrent stage, and a final re-mark[8]. The initial mark and final re-mark both require STW pauses, though they are relatively short. Most of the computation work happens in the concurrent stage, while application threads are also running.

Garbage First (G1)

The G1 algorithm is newer than PGC and CMS, and is designed to achieve predictable and configurable durations of the STW pauses. It utilizes a different model for dividing the memory space, using smaller heap regions, each containing both young and old objects. This allows it to perform garbage collection incrementally.

We refrain from a more detailed discussion of G1 here as it is not required to understand our work, and refer to the Java Garbage Collection handboot [69].

Choosing the Right GC configuration

Different garbage collection algorithms offer a trade-off between three variables: the cost of computation, the total duration of STW pauses, and the distribution or maximum length of STW pauses.

In general, the expectation is that the PGC algorithm is the simplest and has the lowest computation cost. However, since it requires a STW pause for the whole duration of the major GC operation, it is expected to sometimes trigger long pauses.

There is no "best" algorithm, and the most appropriate one depends on the application for two reasons. Firstly, applications have different objectives. Some favour minimizing pauses, while others minimizing computation overhead. Secondly, the results vary based on the nature of the application and its workload.

Because of this, the general recommendation is to experiment to find the best configuration for the given application [69]. This is what we did for Jitsi Videobridge, and we describe the process in the sections below.

[8]This is a simplification; for more detailed description refer to the Java Garbage Collection handbook [69].

7.3.2 Introducing a Memory Pool

One possible way to improve garbage collection performance is to perform some of the memory management in the application, so it can take advantage of application-specific knowledge. In the case of our SFU, we know that most memory allocation comes from the need for `byte[]`s used for RTP packets (see section 7.1 above).

We propose the use of a pool of `byte[]` instances, which are used instead of allocating new ones from the Java heap. This should reduce the allocation rate of heap memory, reducing the strain on the garbage collector.

We face two main problems, the first of which is concurrent access by multiple threads. We have a potentially large set of threads running the packet pipelines, and acquiring a lock for each allocation may degrade performance. We solve this using striping. We have N independent partitions and choose one randomly for each allocation. We choose N based on the number of cores at runtime[9].

The second problem is selecting the size for the `byte[]` buffers. The simplest solution is to use a sufficiently large value to accommodate any RTP packet, for example 1500 bytes. However, this would use more memory than necessary, placing more strain on the garbage collector.

To solve this, we analyzed the request sizes the application makes under representative load. We show the distribution in figure 7.3. Based on this distribution we calculate that using 1500 byte buffers leads to overhead of 71.5%. We propose segmenting the pool into groups with different buffer size. For example, we can use one pool with buffer of size 705, and one of size 1230 (the maximum observed request size). This lowers the overhead to 43.0%. Choosing a threshold of 705 optimizes the overhead.

Similarly, we calculate the optimal choice of thresholds when using more than two groups. With three groups we can reach an overhead of 24.4% ($t1 = 220, t2 = 775$). With four groups we start to see diminishing returns, with an optimal overhead of 15.2% ($t1 = 220, t2 = 753, t3 = 1166$). Therefore, we segment the pool into three groups, with the above thresholds. That is, we use one pool of `byte[]` of size 220, which is used to satisfy requests for sizes of up to 220. A second pool of buffers with size 755 satisfies requests from 221 to 775, and a third pool with buffers of size 1240 satisfies requests from 776 to 1240. For larger requests (if any), we fall back to allocating new memory form the JVM. The procedures for allocating and returning a buffer are described in Algorithm 2.

[9]Currently we just set it equal to the number of cores, but it is not clear if this is optimal.

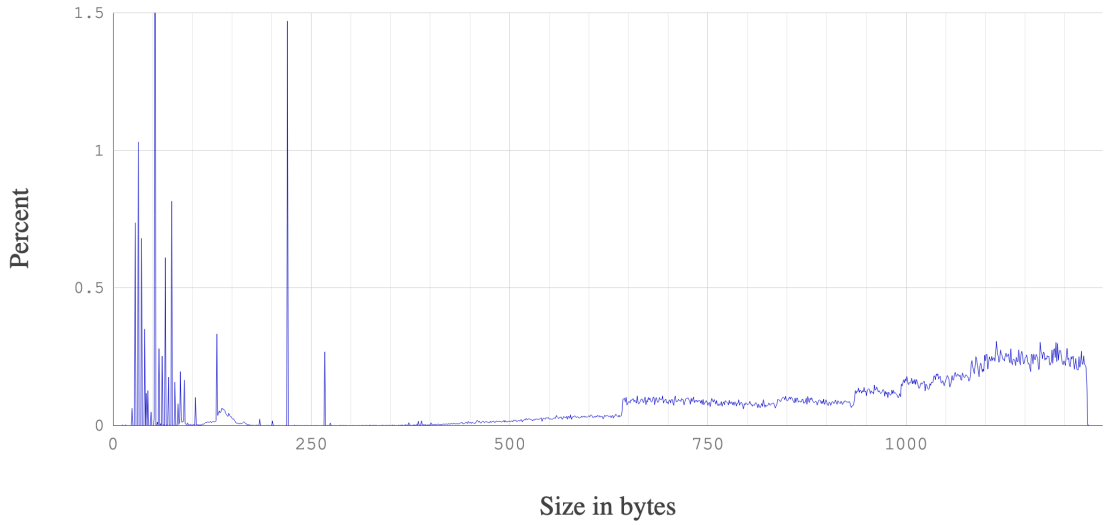


Figure 7.3: *The distribution of memory allocation requests by number of bytes. The group of small requests (*<100 *bytes) come from audio packets. There is a spike at* 220 *coming from padding-only, fixed-size packets that some clients use for bandwidth probing. The majority of requests come from video packets with a wide distribution of approximately* 650 *to* 1240 *bytes.*

We hypothesise that introducing this memory pool improves garbage collection performance, but the results are hard to predict theoretically. We expect less requests to the Java heap (as the pool satisfies most of them), causing the Eden space to fill up more slowly, leading to less frequent Minor GC. However, the objects from the pool eventually end up in the old generation. With PGC, we expect this to lead to longer pauses, due to the de-fragmentation phase. The only way to determine the effects is with experimentation, which we present in the next section.

7.3.3 Evaluation

We set out to evaluate the effects of different GC algorithms and configurations, as well as the memory pool introduced in the previous section, on the performance of Jitsi Videobridge. Specifically, we seek to quantify the trade-off between CPU overhead and STW pauses due to garbage collection.

This proved challenging for several reasons.

Algorithm 2 Algorithm for allocating and returning a buffer from the memory pool. If any of the queues are empty, we allocate a new buffer on the heap.

```
 1: procedure INITIALIZE
 2:     T_1 ← 220
 3:     T_2 ← 775
 4:     MAX_SIZE ← 1240
 5:     N ← numberOfCores()
 6:     poolA ← Queue[N];
 7:     poolB ← Queue[N];
 8:     poolC ← Queue[N];
 9: procedure GETBUFFER(size)              ▷ Allocate a buffer of size size
10:     r ← randomInt(N)
11:     if size <= T_1 then return poolA[r].head()
12:     else if size <= T_2 then return poolB[r].head()
13:     else if size <= MAX_SIZE then return poolC[r].head()
14:     else return newbyte[size];                ▷ Fallback to the Java heap.
15: procedure RETURNBUFFER(buffer)        ▷ Returns a specific buffer to the pool
16:     if buffer.size <= T_1 then
17:         poolA[r].add(buffer)
18:     else if buffer.size <= T_2 then
19:         poolB[r].add(buffer)
20:     else if buffer.size <= MAX_SIZE then
21:         poolC[r].add(buffer)
```

- *Defining outcomes.* We know that long pauses are unacceptable as they lead to packet delays, which in turn causes audio and video playback freezes. However we do not know what the exact limit of acceptability is. Jitsi Videobridge version 1.0 has been used in production systems for long enough that we can consider whatever it achieves acceptable. We reason that delaying packets results in receivers' jitter buffer being extended, increasing end-to-end latency. Therefore, we favour shorter and fewer pauses at the expense of CPU overhead.

- *Repeatable and representable load.* We found that the results varied greatly with the type of load we introduced to the system. We see different results based on the number of participants, conference size distribution, whether or not the test includes creating new conferences (as opposed to taking measurements of existing conferences only), and the conference duration. We had to define a representative test and find a way to make it repeatable.

- *The need for a "warmup".* We found that performance characteristics change when the JVM runs for a longer time. Due in part to the nature of the garbage

collection system, and perhaps also to the just-in-time compiler. We had to find when the results began to converge, and use a "warmup" procedure before each test.

Test Scenario

We set out to define a test scenario with the following characteristics:

1. Representative of the type and size of conferences we see in production.
2. Includes creating new conferences and stopping conferences.
3. Can be scripted to make it repeatable.
4. Produces a sufficient load on the machine.
5. Short enough to perform all experiments in a reasonable timeframe.
6. Uses a limited number of machines, so it can run on the grid we have available.

We defined the following sequence, which uses 16 machines in the cloud with one endpoint on each machine. First, we form two conference of size 8, and stay in this configuration for 5 minutes. Then we form four conferences of size 4, and hold for 3 minutes. Then, we form one conference of size 4 and six conferences of size 2, and hold for 3 minutes. Then we form eight conferences of size 2 and stay for 3 minutes. Finally we form two conferences of size 8, and hold for 3 minutes.

The sequence lasts 17 minutes, plus approximately one minute for initialization, and is implemented in a shell script.

Warmup Procedure

Before each test we perform the following warmup procedure. First, we restart the JVM process. Then we run the test scenario described above twice. The process takes approximately 35 minutes.

We verified that this is sufficient by running the test scenario four times, observing the differences in CPU usage and pauses. There is a significant difference between the first and second runs, a minor but measurable difference between the second and third runs, and no discernible difference between the third and fourth runs.

Measurements

We use the Amazon Web Services cloud for our experiment. The SFU runs on a dedicated `c5.xlarge` instance. The endpoints run on machines in the same Virtual Private Cloud (VPC) to ensure optimal network conditions between them. They use Google Chrome 72.0.3626.119-1 and stream a 720p 30 fps version of the well known *FourPeople* sequence[10] using simulcast.

The most important measurement is the durations of the STW pauses due to garbage collection. We enable verbose logging for the JVM and extract the pauses directly from its log file.

We also measured CPU usage and network bitrate, allowing us to calculate an efficiency score in terms of megabits per percent of CPU usage. We calculate this as $e = bitrate/(cpu\_usage - 1.5)$, where cpu_usage is the average CPU usage in percent, 1.5 is the baseline CPU usage we measured while the system is idle, and $bitrate$ is the average network bitrate in megabits per second. The averages are taken over the course of the test scenario, and sampled every 5 seconds.

Comparing the bitrate allows us to validate that the test was run successfully. We expect the same bitrate in all tests since the machines are in the same network and use the same configuration. If one of the grid machines fails, which happened on a few occasions, we observe a dip in the bitrate for this test run, and we repeat it.

Results

We performed tests with seven different configurations. First, we tested Jitsi Videobridge version 1.0 with the default Parallel GC algorithm in JDK 8, which is our definition of *acceptable* performance. All other tests were performed with version 2.0 of Jitsi Videobridge because it is the one we are interested in optimizing. We tested the PGC and CMS algorithms for garbage collectors, with and without the additional memory pool. Finally, we tested the default configuration of the G1 algorithm with the memory pool.

Figure 7.4 details the distribution of GC pauses for two of the tests. We report the summarized results from all tests in table 7.1. For each configuration we present:

- `pps` The average number of STW pauses due to garbage collection per second ("pauses per second").

[10] https://media.xiph.org/video/derf/

Table 7.1: *The GC pauses, CPU usage, and bitrate for the seven different configurations.*

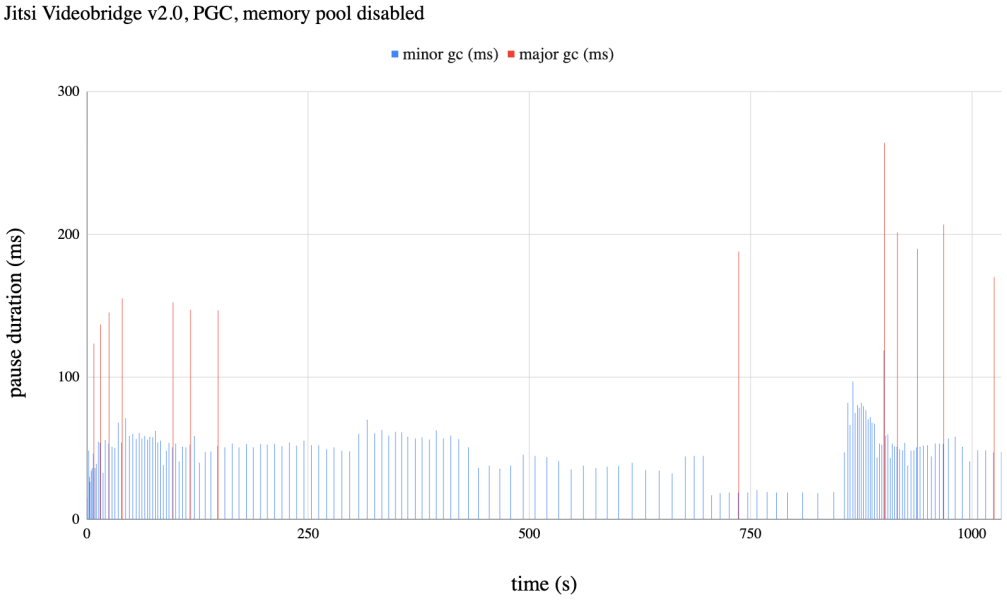
	v1.0	PGC	PGC + pool	CMS	CMS + pool	G1 + pool
pps	0.29	0.16	0.08	0.21	0.14	0.27
mp	384	264	248	106	82	153
pf	0.92	0.91	0.46	0.98	0.51	1.10
cpu	24.7	17.9	18.0	20.4	20.0	19.5
br	108.5	109.4	110.3	109.2	110.1	109.9
eff	4.68	6.67	6.68	5.77	5.96	6.10

- `mp` The length of the longest STW pause due to garbage collection, in milliseconds ("max pause").
- `pf` The percent of the total time spent in STW pauses due to garbage collection ("percent frozen").
- `cpu` The average CPU usage in percent.
- `br` The average bitrate in megabits per second.
- `eff` The efficiency in *Mbps* per percent CPU, calculated as described above.

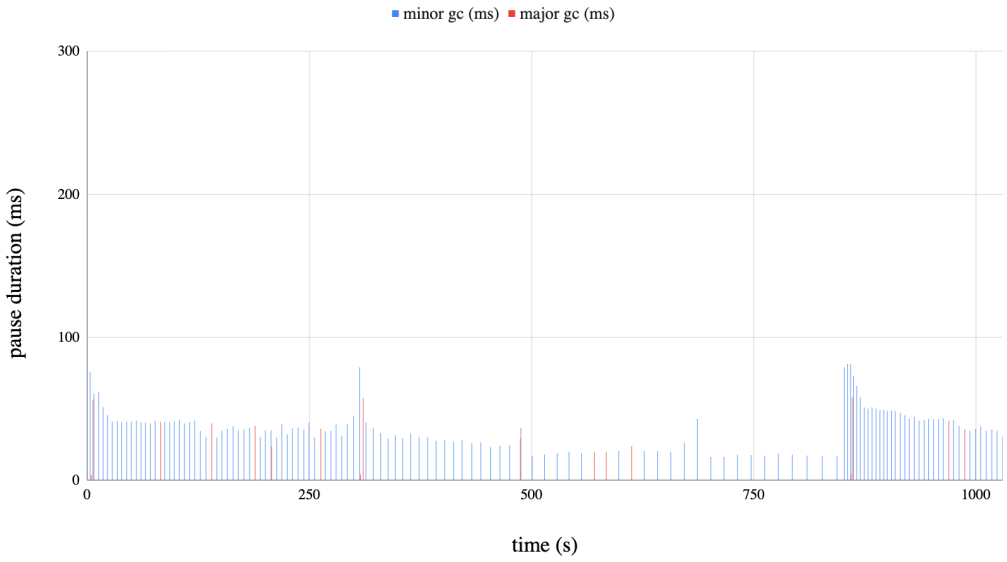
Interpretation

The tests with version 1.0 of Jitsi Videobridge give us a baseline for performance. We see a pause of 384 milliseconds, much longer than expected. It is surprising that such causes does not lead to noticeable freezing in the video playback.

When using the default PGC algorithm we failed to reproduce the extremely long pauses that prompted us to start this investigation. This could be because these tests use a more powerful machine, or because it was better warmed up than in previous experiments. While the longest pauses are shorter than in version 1.0, they are still significantly long. As expected, they come from Major GC events (see figure 7.5), i.e., cleaning the old generation. Contrary to our initial expectation, the introduction of the memory pool does not increase their length, though the decrease is very slight (from 264 to 248). We also notice that the frequency of pauses and the total duration has dropped by half with the addition of the memory pool. Finally, we see improved efficiency compared to version 1.0, which is expected because version 2.0 contains many CPU optimizations. The addition of the memory pool improves all metrics.



(a) *The default Parallel GC algorithm, without the application memory pool.*



(b) *The Concurrent Mark and Sweep algorithm, with the application memory pool.*

Figure 7.4: *The distribution of stop-the-world pauses due to garbage collection using the default GC algorithm (a) and CMS with the memory pool (b). Minor GC events are shown in blue and Major GC events in red. CMS pauses less frequently and for much shorter periods. Cleaning the old generation causes a long pause for PGC, but not for CMS. We can also see a similarity in how the frequency changes with time due the conference pattern in the test scenario. The beginning (*$5\,\mathrm{min}$*) and ending (*$3\,\mathrm{min}$*) use two* 8*-endpoint conferences, causing higher load. The middle period uses more conferences with fewer endpoints, causing lower load.*

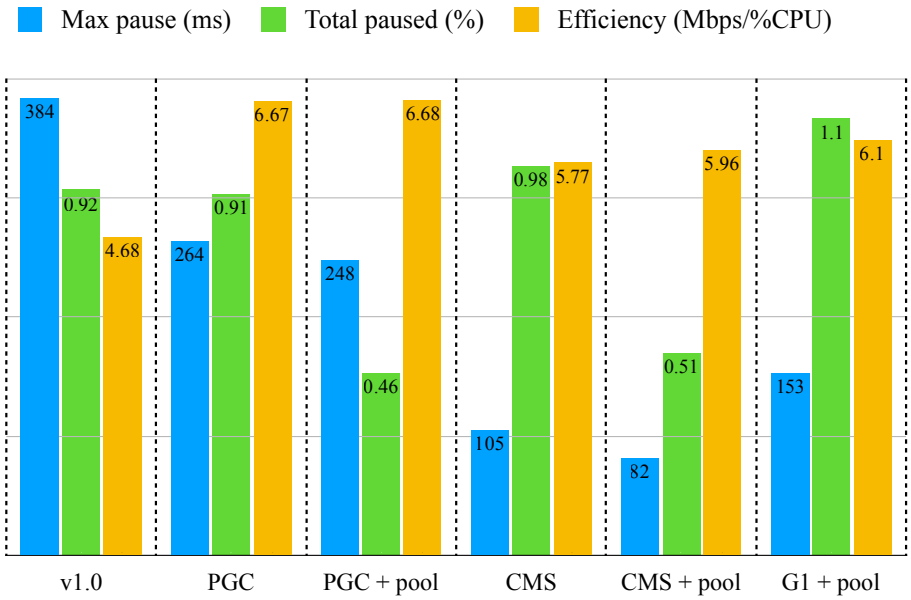


Figure 7.5: *Longest pause duration, total pause duration, and efficiency for the six different configurations tested.*

Using the CMS algorithm, the addition memory pool again improves all metrics: the longest pause drops from 106 to 82 milliseconds, the pause frequency drops from 0.21 to 0.14 per second, the pause duration drops from 0.98% to 0.51%, and efficiency increases from 5.77 to 5.96 Mbps/%cpu.

Comparing CMS vs PGC with the memory pool enabled we see a drop of efficiency as expected. It drops from 6.68 to 5.96, approximately 11%. However, the longest pause is significantly reduced from 248 to 82 milliseconds. There is a slight increase in the total pause duration (from 0.46% to 0.51%. Overall, since SFU are sensitive to long pauses, CMS performs much better than PGC.

The performance of G1 is underwhelming. Compared to CMS it pauses more often, for as long as 153 ms. The total freeze duration is 1.1%, the highest of all tests. It offers a slight gain in efficiency. It is worth experimenting further, and testing different G1 configurations, but based on the current results we prefer CMS.

7.4 Conclusion

Selective Forwarding Units do not require a large amount of memory, but their pattern of memory allocation can be problematic when used with automatic garbage collection. The high allocation throughput leads to frequent GC activation, which can freeze the application for a significant time with GC algorithms optimized for low CPU overhead, such as the default algorithms that come with the Java Development Kit.

We proposed using a separate memory pool implemented inside the Java application to reduce the strain on the garbage collector. We then developed a methodology to evaluate the performance of different configurations of the garbage collector and memory pool. We performed experiments using the default GC algorithm in JDK as well as CMS and G1, and quantified their performance for the Jitsi Videobridge application.

We found that the best performance is achieved using the CMS algorithm together with our proposed memory pool. It leads to acceptable performance, defined as better than our baseline of Jitsi Videobridge version 1.0 using the default GC configuration. The longest stop-the-world pause (the most important metric) was reduced from 384 to 82 milliseconds. This trade-off was possible with an 11% drop in efficiency. The memory pool improved all metrics in all tests.

7.4.1 Further Research

Our experiment did not test different configurations of G1, some of which may offer better performance. Additionally, a new garbage collector for Java, Shenandoah[11], is currently in development. It is optimized for low pause durations for very large heaps. Further experiments should be performed to measure their performance.

Another way to reduce the strain on the garbage collector is to move the memory used for RTP packets out of the garbage collected heap. In Java, this is possible using *direct memory buffers*. Since the total amount of memory is not a concern and the chunks are relatively large[12], we can implement this efficiently using, for example, a pre-allocated portion of memory along with bitmaps to indicate the state of any chunk. However, performing this type of memory management comes with the risk of memory leaks. With the heap-backed memory pool we automatically fall back to the Java

[11] See https://wiki.openjdk.java.net/display/shenandoah/Main

[12] The main reason for segmenting the memory pool is strain on the GC, which no longer applies, we can simply use 1500 byte chunks.

garbage collector: if the application fails to return buffer, the GC will eventually reclaim it. With direct memory this leads to a leak.

Further research is also needed to better understand the overall effects of GC pauses. It is easy to see that extremely long pauses result in video freezing, but we do not know at what length this starts to occur. We do not know the effects of shorter pauses either. If a pause does not cause a freeze it is likely due to the receiver's jitter buffer being long enough to accommodate the delay. How much do garbage collection pauses affect the jitter buffer? Does the pause frequency have an effect, or just the maximum length? More research is needed to answer these questions and to find what pause duration distributions are acceptable for video conferencing.

Chapter 8

Conferences with Cascaded Selective Forwarding Units

So far we discussed conferences which use one Selective Forwarding Unit in a centralized topology (a star). In this chapter we explore the possibility of using a set of interconnected SFUs in a conference. There are two motivations for this. It could allow conferences to scale to a larger size, and it could improve users' connection quality if they connect to a nearby server. We focus on the second goal, but we also consider the first one for the design of the system.

8.1 Considerations for Geo-located Video Conferences

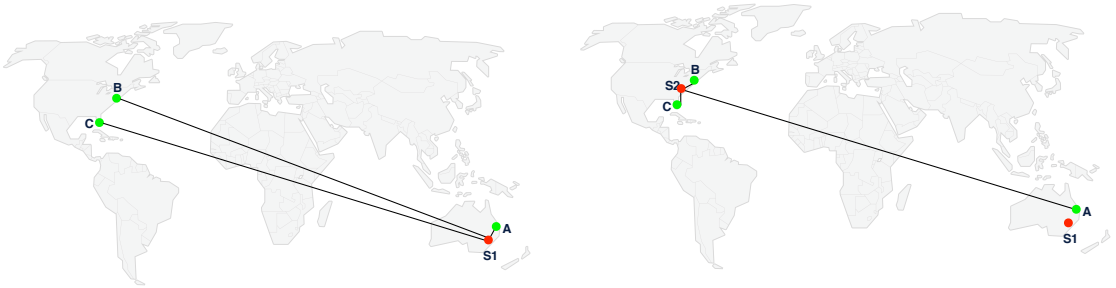
Modern cloud services such as Amazon AWS[1] make it easy to deploy a system comprising of servers in different geographic locations, and provide well developed tools, like Route53[2], for routing users to a server close to them. However, they are mostly designed for traditional web applications and the scenario in which one consumer accesses a certain resource in the cloud.

Video conferencing has the potential to greatly benefit from geo-location features, because it is interactive in nature, and it is very sensitive to network conditions such as banwidth, packet loss, round-trip-time (RTT), and variations in RTT (jitter). However, this case is not as simple as connecting one consumer to the closest server as it requires

[1] `https://aws.amazon.com`
[2] `https://aws.amazon.com/route53/`

the interconnection of two or more endpoints, and the problem of optimal selection of the server locations is not well explored.

Most systems, including Jitsi Meet and the *meet.jit.si* service and Microsoft Teams[3] use a simple, but limited approach. They deploy a set of independent servers in different geographic regions, and make the server selection when the conference is initially allocated. At this stage they have information only about the first (or first two) endpoint, and they select a server near the first endpoint. Once selected, the server is used for the whole duration of the conference, regardless of how the remaining endpoints are distributed geographically.



(a) *The server is selected in endpoint C's region, which is suboptimal.*

(b) *The server is selected in the region of endpoints A and B, reducing the end-to-end delay between them.*

Figure 8.1: *Centralized conferences with three endpoints in different regions, with different servers selected. Endpoints are represented in green, and servers in red.*

This is relatively straightforward to implement, and works well in most cases. Particularly, it is optimal any time all endpoint are in the same region, which is common. However, with endpoints from different regions we begin to see some problems. Consider the three-endpoint conference depicted in figure 8.1.a. Even though endpoints B and C are close to each other, their communication goes through a distant server, leading to an increased end-to-end delay between them. In this trivial example the problem can be solved by selecting a different server, as shown in figure 8.1.b. But in general there could be more than one pair of close endpoints, and the problem cannot be avoided using a single server.

There is a more subtle issue, which is present any time there are endpoints in different regions, regardless of where the server is. This is the latency between an endpoint and the server it is connected to. It is important, in addition to the end-to-end network latency between endpoints, because of stream repair based on packet retransmission.

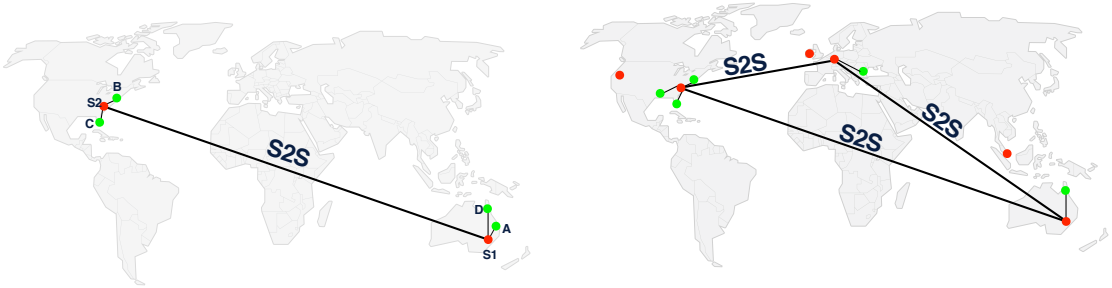
[3]https://tomtalks.blog/2019/06/where-in-the-world-will-my-microsoft-teams-meeting-by-hosted/

The longer this latency is, the larger the jitter buffer has to be in order to accommodate retransmitted packets, leading to increased end-to-end media delay.

Both issues can be addressed by introducing a set of servers, and connecting each endpoint to a local server. This is shown is figure 8.2. End-to-end latency is reduced in the problematic cases because communication goes through a server in the local region, instead of a remote region. The latency to each endpoint's server is reduced, because it is always in a local region.

However, there are also some disadvantages. While the RTT between endpoints in the same region decreases, it may increase for some endpoints. We can illustrate this using figure 8.1.b. The connection between endpoint A to B goes through a server S2. If we use both servers, the connection goes through S1 first and then S2. The direct path from A to S2 is likely shorter than from A to S1 and then S2.

Another disadvantage is the additional internal bandwidth required between the servers, and the complexity that it adds to the system.



(a) *The case of just two servers.*

(b) *The more general case with more than two servers, connected in a mesh.*

Figure 8.2: *Conferences with cascaded servers where each endpoint is connected to a local server. Endpoints are represented in green, and servers in red, and server-to-server connections with a thick line and marked "S2S".*

The overall effect of using geo-located cascaded SFUs depends on usage patterns of the service. For example, if it tends to be used for inter-continental conferences, the effects are more likely to be positive. It also depends on the available server locations, conference size, and other factors. In short, these are hard to predict.

8.2 Preliminary Analysis

The implementation of a distributed SFU system requires a substantial effort. Before undertaking this venture, we performed an analysis of the *meet.jit.si* service (for which the feature was first intended), attempting to predict the effects we should expect.

At the time the service was distributed in four AWS regions: us-east-1 (North Virginia), us-west-2 (Oregon), eu-west-1 (Ireland) and ap-southeast-2 (Sydney).

Each conference is allocated a single server, with its region being based on the location of the first participant in the conference. This means that as new participants join, they do not always connect to a server in the region closest to them. This gives us an opportunity to measure the round trip time from users detected as being located in one region, to a server in another region.

8.2.1 Round Trip Time Between Servers

We measured the RTT between servers in the four available regions. We used ICMP ping, sampling once a minute for a week. We found that the packet loss and the variation in RTT were both negligible. Figure 8.3 shows the results.

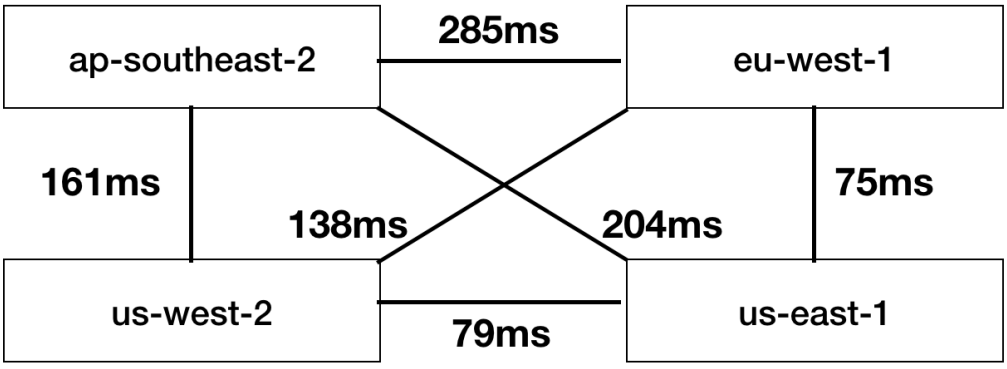


Figure 8.3: *The Round Trip Time in milliseconds between the four AWS regions that we use.*

8.2.2 Round Trip Time Between Endpoints and Servers

We measured the RTT between endpoints and the server they are connected to. For each user session, we obtained the RTT periodically during the conference, using the WebRTC statistics API [6]. We took the mean across time as the RTT for the session. We then grouped the sessions by the user region and the server region, and we show the means in table 8.1. The dataset comprises 40214 sessions, and the smallest group (users from us-west-2 connecting to ap-southeast-2) has 833 sessions.

Table 8.1: *The average RTT in milliseconds from users in a given region to a server in a given region. Vertically: user region; horizontally: server region.*

	server region	ap	eu	us-e	us-w
user region					
ap		329	291	242	292
eu		378	110	171	216
us-e		307	189	107	116
us-w		241	213	137	81

8.2.3 Distribution of Users

We analyzed the sizes of the conferences on *meet.jit.si* in a period of 6 weeks, during which we recorded 56383 conferences. Since the size of a given conference changes with participants joining and leaving, we looked at the total amount of time that conferences of a given size were active. The results are shown in Figure 8.4.

Additionally, we observe that 72% of all conferences never grow to a size of 3 or more for more than two minutes (we introduce this threshold, because if a user reloads the web-page this is incorrectly registered as a temporary third participant, until the old session times out). We refer to the remaining 28%, which have grown to a size of 3 or more for at least two minutes, as multi-party conferences.

Next, we search for situations in which a conference hosted on a server in one region has at least two participants from the same region, but different from the server region. This is similar to the situation depicted in figure 8.1.a, but generalized to any number of total endpoints.

We find that 33% of all multi-party conferences reach such a state at some point in their life cycle. Split by time, 13% of the time spent in multi-party conferences is spent in this situation.

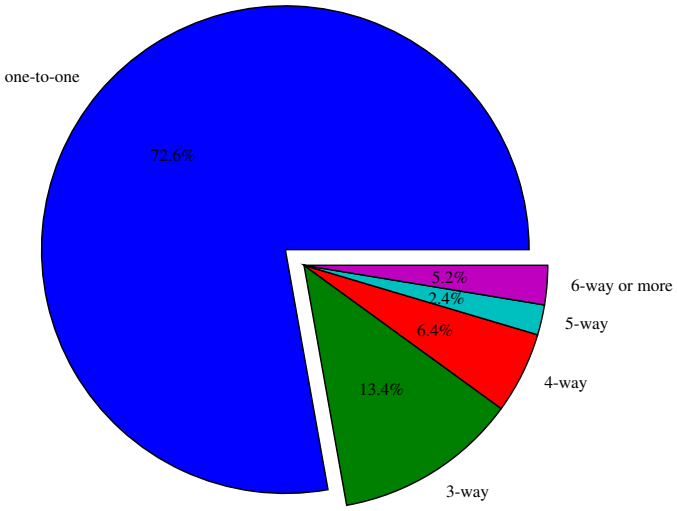


Figure 8.4: *The distribution of conference-time on our service by conference size.*

8.2.4 Analysis

The first thing we notice is that in almost all cases, the average RTT from a user to a server increases if we introduce an intermediary server in the user's region. We can calculate this based on the server-to-server and endpoint-to-server RTTs we measured above. As an example, the average RTT from users in Europe to our server in eu-west-1 is $110\,\mathrm{ms}$, and to us-east-1 it is $171\,\mathrm{ms}$. Taking into account the $75\,\mathrm{ms}$ RTT between our servers in eu-west-1 and us-east-1, if we route European users through a server in eu-west-1 we should expect to see an average RTT (to the server in us-east-1) of $185\,\mathrm{ms}$, which is an increase of $14\,\mathrm{ms}$. Figure 8.5 illustrates this. We see this pattern for all pairs of regions, with the sole exception of users in us-east-1 connecting to a server in eu-west-1, in which case we see a decrease in the average on $3\,\mathrm{ms}$.

This suggests that using distributed SFUs in a case like the one in figure 8.1.b slightly increases the average RTT between pairs of participants in the conference; i.e., it is counterproductive in addition to being costly.

This effect is strongest in the ap-southeast-2 region, where it turns out that users from within the region have a better RTT to servers in any other region, then they do to the server in their own region. We suspect that this is because ap-southeast-2 covers a much larger geographic area, including both countries in Asia as well as Australia. In this case introducing an intermediary server in ap-southeast-2 (Sydney) would sig-

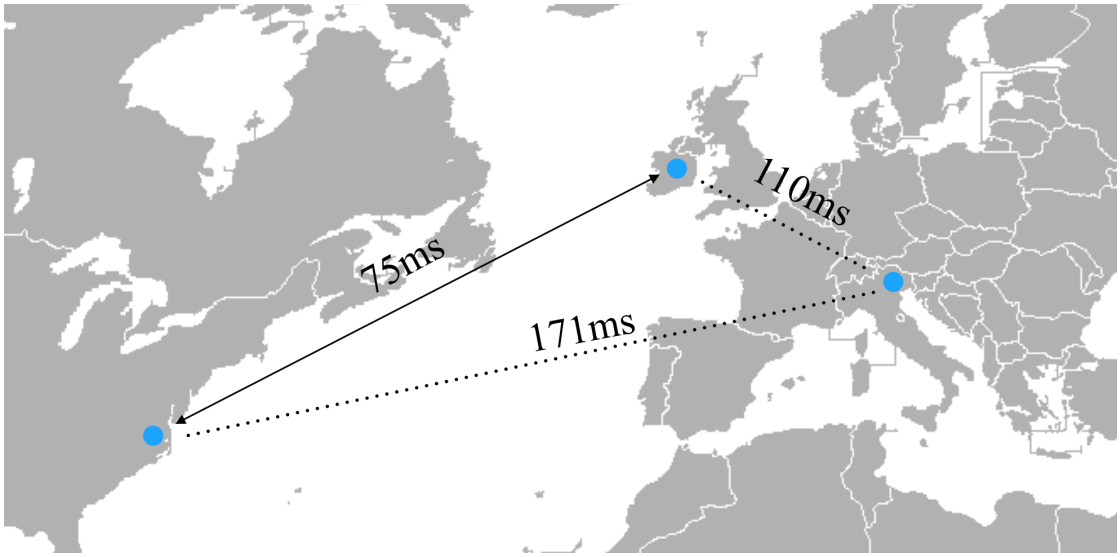


Figure 8.5: *The average RTT between users in Europe and our servers in Europe and the US, and the RTT between our two servers. The direct connection's latency is* $14\,\mathrm{ms}$ *lower.*

nificantly increase the RTT for many users, to the point where it might exceed the ITU recommendations and start to inhibit the ability of users to communicate.

We expect that placing servers in more regions (we currently use four regions, out of the 16 that AWS offers) will have a significant impact on the users average RTT, decreasing it for local users, and it will also decrease the negative effect of using an intermediary server. However, this also brings higher infrastructure costs.

We also notice that the majority of conferences on our service only have two participants, and would be unaffected by the introduction of cascaded SFUs. Multi-party conferences are one of the core features of the system, and as such we consider them an important use-case. The situation in which two endpoints close to each other connect to a distant server (figure 8.1.a) happens relatively often – in 33% of multi-party conference – which means that distributed SFUs have the potential to provide benefits for user experience.

In conclusion, while we have gained some insight, we are unable to accurately predict the overall effect. Next, we describe the system we implemented and present its results.

8.3 Implementation

Our goal is to extend the Jitsi Meet system with support for cascaded SFUs. The system already had a sharded architecture which allowed geo-location based on the first participant in the conference, and both clients servers were aware of their geographical location ("region").

8.3.1 Signaling

The Jitsi Meet systems uses two separate servers for signaling and multimedia.

The Jicofo[4] component is used for signaling. It maintains a Jingle [49] session with each endpoint, allocating the necessary media processing resources on a separate Jitsi Videobridge SFU instance. Since it can handle a very large number of conferences, we only have one Jicofo instance in each region.

For the SFUs we use an AWS autoscaling group in each region, all connected to the local Jicofo server. This allows us to easily scale up by bringing new machines.

This architecture makes it relatively easy for us to enable cascaded SFUs, because we can keep the centralized signaling server, which already has a mechanism to communicate to SFU instances through the COLIBRI [36]. We chose to keep the signaling only between Jicofo and SFUs, but not between any two SFUs. This significantly simplifies the system, because most decisions are made centrally in the signaling server, and each SFU has a single signaling link.

We had to do three things to make the signaling work:

- Cross-connect SFUs and signaling servers from all regions.
- Make the signaling server region-aware in its SFU selection strategy.
- Extend COLIBRI with descriptions of the additional SFUs in a conference.

Figure 8.6 shows the resulting infrastructure.

[4]https://github.com/jitsi/jicofo

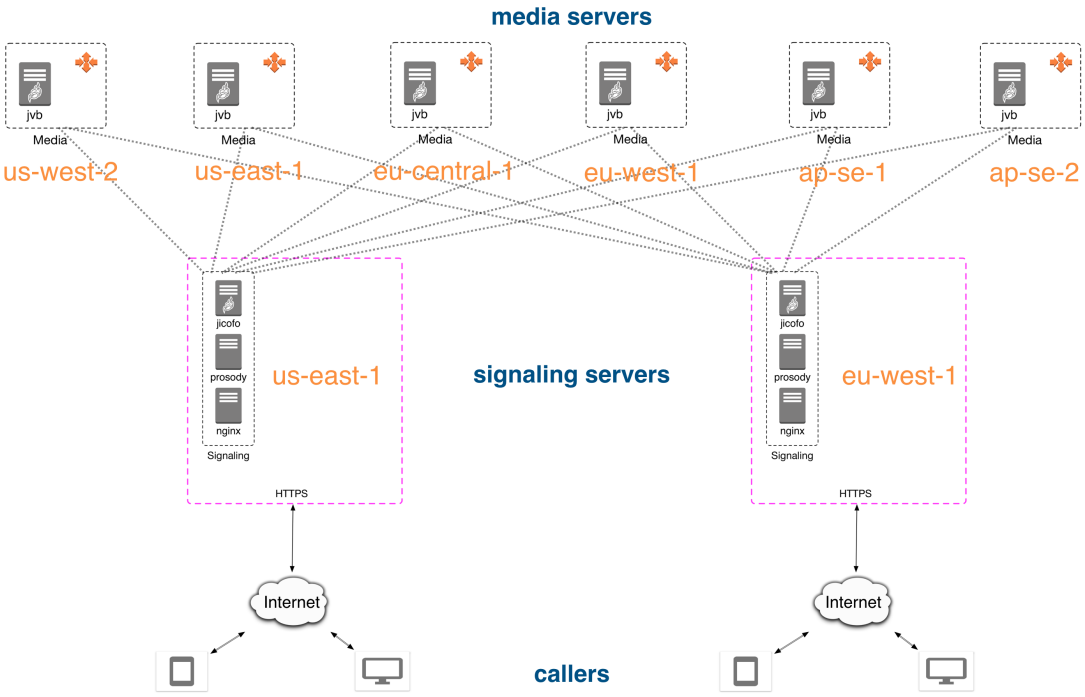


Figure 8.6: *The infrastructure used to enable cascaded SFUs. Each SFU connects to signaling servers in each region. The signaling servers can use any of the SFUs.*

8.3.2 Selection Strategy

By "selection strategy" we refer to the process that a signaling server uses to chose which SFUs to use for a given conference. With single-SFU conferences, this is relatively simple. A Jicofo server has access to a set of SFUs in the local region, and it only has to consider the load, i.e., to perform load balancing. It does this by selecting the least loaded SFU each time.

With cascaded SFUs the problem is more complex, as it has to attempt to use a localized SFU for each client, in addition to balancing the load.

We propose algorithm 3, which is designed to always select an SFU in the client region if available, then minimize the number of servers used, and finally balance the load. The algorithm is invoked any time a new endpoint joins a conference.

We see that load-balancing is only a third priority, which may lead to overloading. To avoid this we use relatively low thresholds for autoscaling. That is, once all SFUs in a region exceed a certain load level, we add new machines, which are immediately used for new conferences because of their initial low load.

Algorithm 3 Algorithm for SFU selection for a new client. We assume that `all_sfus` is a list of all available SFUs ordered by load.

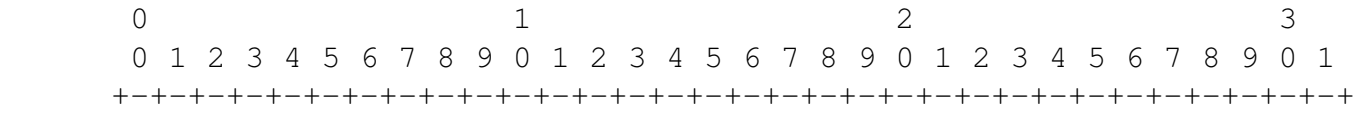
1: **procedure** SELECT($conference, endpoint$) ▷ Selects an SFU for a new endpoint in a conference.
2: **if** $conference.getSFUs().isEmpty()$ **then return** SELECTINITIAL($conference, endpoint$)
3: $conference\_sfus\_in\_region \leftarrow conferece.getSFUs().filterByRegion(endpoint.getRegion())$
4: **if** $conference\_sfus\_in\_region.isNotEmpty()$ **then return** $conference\_sfus\_in\_region.first()$
5: **else**
6: $sfus\_in\_region \leftarrow all\_sfus.filterByRegion(endpoint.getRegion())$
7: **if** $sfus\_in\_region.isNotEmpty()$ **then return** $sfus\_in\_region.first()$
return $all\_sfus.first()$
8: **procedure** SELECTINITIAL($conference, endpoint$) ▷ Makes the initial selection of an SFU for a conference. Attempts to use the least loaded SFU in the endpoint's region, otherwise the least loaded SFU in any region.
9: $sfus\_in\_region \leftarrow all\_sfus.filterByRegion(endpoint.getRegion())$
10: **if** $sfus\_in\_region.isNotEmpty()$ **then return** $sfus\_in\_region.first()$
11: **elsereturn** $all\_sfus.first()$

8.3.3 Server-to-server Transport: Octo

We chose to create a new encapsulation protocol for the server-to-server media transport, named Octo. It works on top of UDP and consists of a simple fixed-size header and payload. The payload is either RTP/RTCP or a "message" type used to carry additional messages between the endpoints of a conference[5].

Each SFU binds to a socket and receives remote Octo traffic on it. When a conference uses more than one SFU, Jicofo signals to each SFU the addresses of the other SFUs in the conference. In effect, the SFUs in a conference form a full-mesh topology, however this is a limitation of the current Jicofo implementation and not the Octo protocol itself.

The Octo header has the following format:



[5]For the endpoint-to-SFU connections, these messages go over either WebRTC `DataChannel`s or a custom `WebSocket` based protocol.

```
|R| M | S | res |              Conference ID                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Endpoint ID                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The `M` field specifies the media type (audio, video, or message). The `S` field specifies the simulcast encoding ID. The `Conference ID` field is used to multiplex different conferences on the same UDP port. The `Endpoint ID` identifies the source endpoint within the conference. The `R` and `res` fields are currently not used. The former is intended to allow topologies other than full-mesh, by instructing an SFU to either forward the packet to other SFUs or not.

8.3.4 Packet Retransmissions

Stream repair is performed hop-by-hop, that is between an endpoint and the SFU.

The current implementation does not perform stream repair for the SFU-to-SFU connections, because the network conditions we observed in the AWS network are very stable. Packets lost between the SFUs are eventually be requested by the endpoints. The intention is to add stream repair between the SFUs at a later stage.

We needed to make one small modification to the re-transmission algorithm in the SFU to avoid a scenario which would result in unnecessary retransmissions. This is illustrated in figure 8.7. We solve this by ignoring retransmissions which come less than one round-trip-time after the packet was sent.

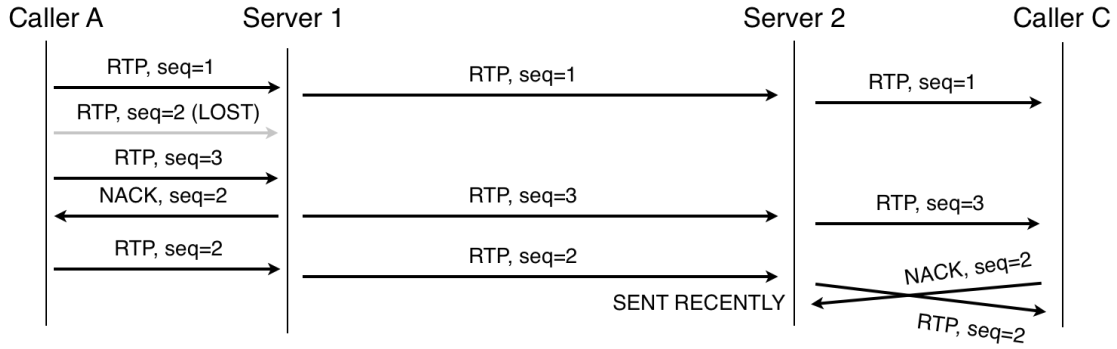


Figure 8.7: *The retransmission algorithm on the SFU needs to be modified to avoid sending packets unnecessarily.*

8.3.5 Simulcast

The current system always forwards all simulcast encodings between the SFUs. This simplifies the implementation, because an SFU does not need to consider other SFUs or the endpoints connected to them when making the decision. In fact, limiting the forwarded encodings requires some form of extra signaling. This solution is inefficient in terms of the server-to-server traffic, but does not have any effect on quality of service.

The intention is for a future version of the system to only send the simulcast encodings which are necessary.

8.3.6 Dominant Speaker Identification

Each SFU performs dominant speaker identification independently. Even though the inputs are technically different because of the different relative time that packets are processed, this does not have a significant impact on the output of the algorithm.

8.4 Results

After deploying the Octo system described above to the *meet.jit.si* service, we performed an experiment to measure the overall effects on connection quality. The conferences were hosted on the same set of servers, but each was randomly selected with a 50% probability to either enable the cascaded SFU mode or disable it, in which case the previous single-SFU mode (selected based on the first participant in the conference) was used. The experiment was run over three weeks and included over 28000 conferences.

We also note that at this point (based on the results of the preliminary analysis) we had extended the *meet.jit.si* with servers in two additional regions: eu-central-1 (Frankfurt, Germany) and ap-se-1 (Singapore). This makes a total of six regions.

We took two measurements: round trip time from an endpoint to its SFU, and end-to-end round trip time between endpoints.

The endpoint-to-SFU round trip time was taken from the WebRTC stats API [6], and is measured on the RTP level.

The end-to-end round trip time was measured for any pair of endpoints. It was based on a request originating from the JavaScript application of one endpoint, passing

through the WebRTC `DataChannel` to the SFU, passing to the remote endpoint's SFU (when cascaded SFUs are enabled) via the Octo protocol, and finally to the remote endpoint over the `DataChannel`. The remote endpoint's JavaScript application receives the message and generates a response that travels through the same path in reverse. Using the JavaScript layer often results in delays, which means that this measurement does not accurately measure the network round trip time between the endpoints (i.e., it add some positive noise). However, since the conferences are randomized, the noise in the two groups is equal; we can use this measurement to compare the network round trip times.

Figures 8.8 and 8.9 show the results. The combined values are average over all conferences, while the per-region values are grouped according to the reporting endpoint's local region. As an example, an endpoint in Sydney reporting end-to-end RTT to an endpoint in Europe is counted towards the Sydney region.

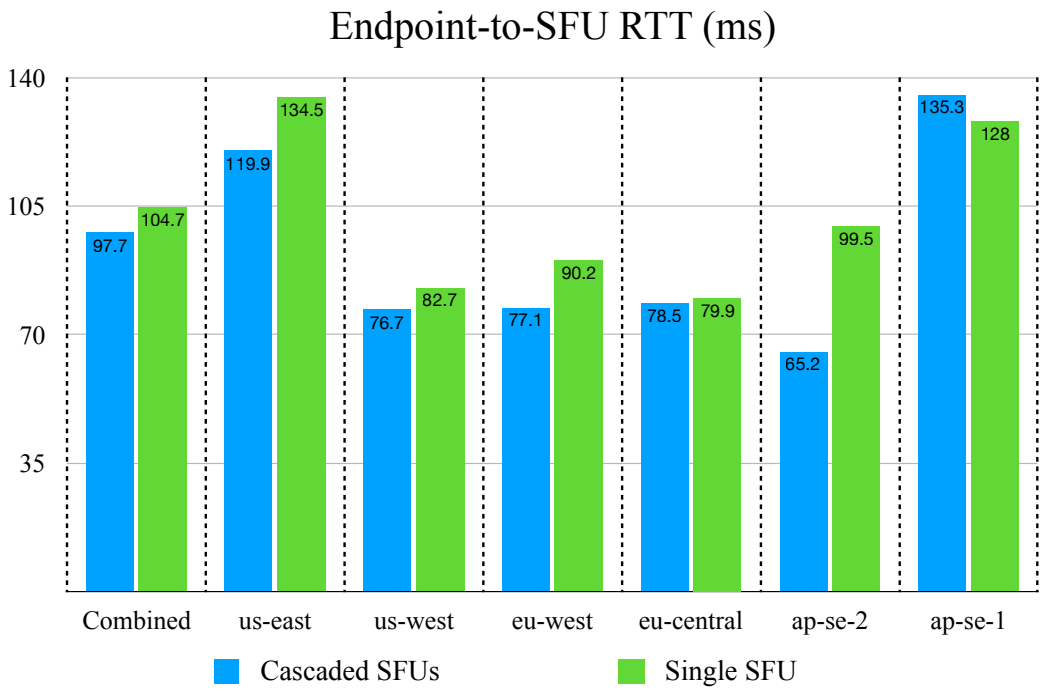


Figure 8.8: *The round-trip-time from endpoints to their SFU in milliseconds, split by the endpoint's region, and combined.*

We can see that, as expected, the latency to an endpoint's SFU is decreased when we use cascaded SFUs because each endpoint connects to a local SFU. However, the effect is small, only 7 ms overall. In the Singapore region the RTT actually increased by 7 ms. We do not know what the reason is, but we speculate that it may indicate incorrect

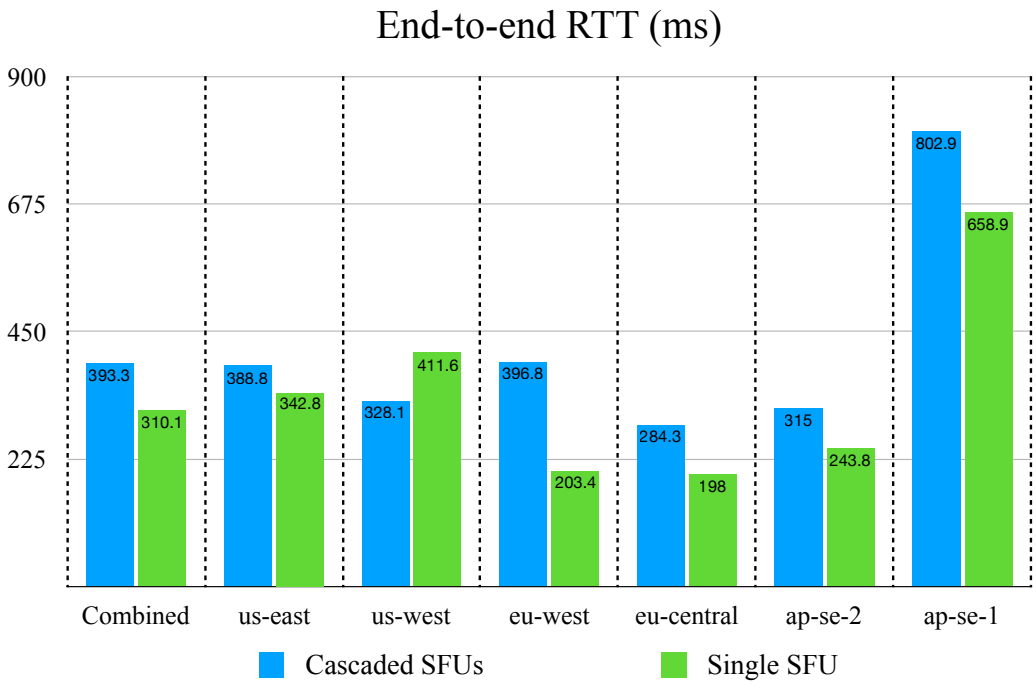


Figure 8.9: *The end-to-end round-trip-time between endpoints in milliseconds; split by the reporting endpoint's region, and combined.*

detection of the region by clients, i.e., some clients may be sent to the Singapore servers even though they have a lower RTT to another server.

However, introducing cascaded SFUs significantly increases the end-to-end round trip time between endpoints. Overall the increase is $83\,\mathrm{ms}$, and it holds in all regions except for us-west, where we see a decrease of $83.5\,\mathrm{ms}$. For the Singapore region, where RTTs are already very high, we see a further increase of $144\,\mathrm{ms}$.

8.5 Conclusion and Future Work

Geo-location for video conferencing is more complicated than it is for simple resource-consumer systems. Following the common wisdom and connecting all users to the server nearest to them is not always optimal, and in some cases has a negative impact on users. It is also technically challenging to implement and has other disadvantages, such as increased traffic for the infrastructure.

The simplest approach for geo-location – using a single server and choosing a region based on the first participant in the conference – is easy to implement and works reasonably well. The effect of adding geo-located cascaded SFUs to a video conferencing system depends on the hosting/cloud provider and the usage patterns of the service, including conference size and distribution of users. An in-depth analysis should be performed for the specific service under consideration, in order to get the most out of cascaded SFUs and not introduce harm.

Using cascaded SFUs has the potential to improve the user experience, as is evidenced by the reduced RTT to the local SFU, and end-to-end RTT in the us-west region. However, the selection strategy of connecting every endpoint to an SFU in its local region has the unintended side effect of significantly increasing overall endpoint-to-endpoint delay.

One hypothesis to explain this effect is that a significant fraction of the endpoints are "in the middle" of two regions. That is, there is more than one region with comparable round-trip-time. In this case using cascaded SFUs leads to only slight improvements of the endpoint-to-SFU delay, while significantly increasing the endpoint-to-endpoint delay (since the server-to-server delay is incurred as well). Figure 8.10 illustrates the idea. Further research is needed to find out whether this is the correct explanation in our case.

Further research is also needed to better understand in which situations cascaded SFUs are beneficial, neutral, or harmful. This may depend on conference size, e.g., there may be benefits for conferences over a certain size. It may also depend on the region. The us-west region saw an overall benefit of cascaded SFUs. Which of its characteristics contribute to this effect? With this knowledge we could design a better selection strategy to use cascaded SFUs optimally.

An additional question which arose from the results in the Singapore region is, how well does the region selection perform? Ideally, an endpoint should be identified as belonging to the region which minimizes its round trip time. However, in practice we cannot perform RTT measurements for all available regions prior to each endpoint joining a conference, so we rely on geo-location systems with pre-configured (although dynamically updated) network mappings. In our case this is Amazon's Route53 DNS based routing. We can measure how well it performs. Once an endpoint enters a conference, we can measure RTT to all available regions and find out whether the initial choice was optimal. We have now started an experiment to answer this question.

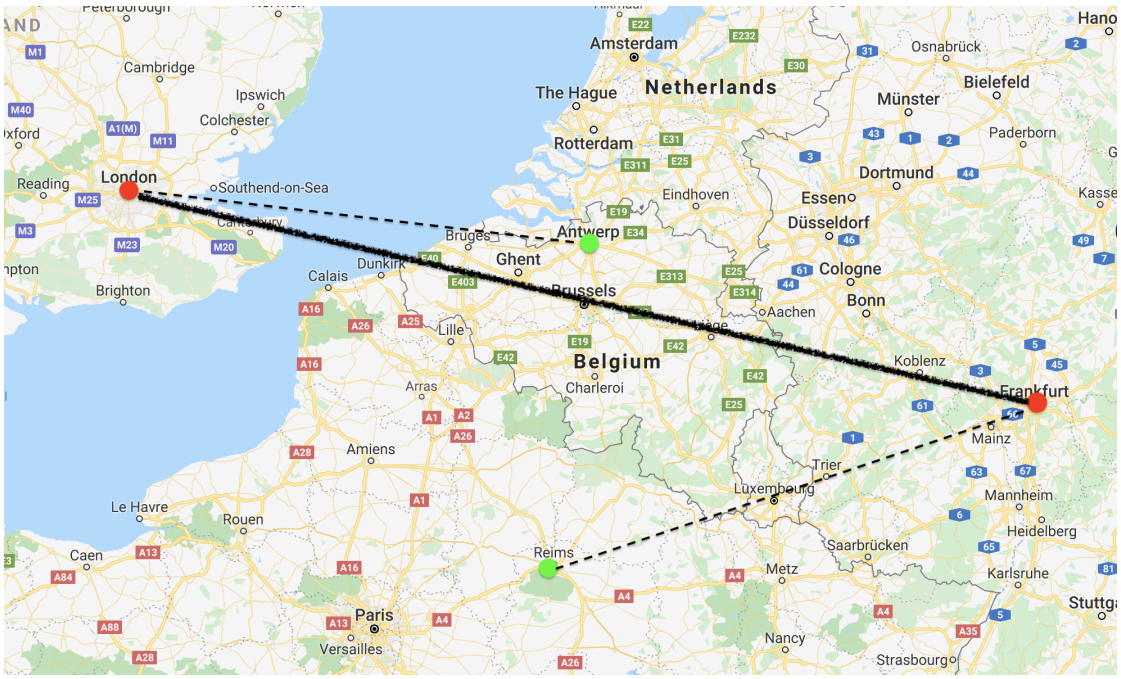


Figure 8.10: *A situation in which connecting endpoints to the nearest server is harmful. Each endpoint (in green) is equally close to the two servers (in red). Using both servers increases the end-to-end delay.*

Chapter 9

End-to-End Encrypted Conferences with Selective Forwarding Units

Financial institutions require enhanced security to protect their assets. Historically, this has been achieved that through internal communication channels or physical dedicated lines with trusted partners. Internally, high level encryption, coupled with smart management of security keys adds the final touch to as-secure-as-you-can-be systems.

This approach has two drawbacks. Establishing communication channels between institutions takes significant time and expense. Furthermore, it is difficult, if not impossible, to leverage the power of cloud infrastructures and Platform As A Service (PaaS) providers, because it is unacceptable for the data to leave the domain controlled by the institution.

In 2015, the Internet Engineering Task Force (IETF) formed a new working group called Privacy Enhanced RTP Conferencing (PERC[1]). Part of its charter addresses these problems, specifically it aims to provide a way to use RTP conferencing with a centralized media distributor (such as a Selective Forwarding Unit), while keeping the multimedia and metadata private and only available to a set of invited participants. This allows usage of media servers in the public cloud without compromising the security.

In this chapter, we investigate the feasibility of implementing the PERC double encryption specifications [38] in conjunction with WebRTC. We find that the system does not support simulcast as currently used in WebRTC and propose modifications to enable support for simulcast.

[1] https://datatracker.ietf.org/wg/perc/about/

9.1 PERC Double Encryption Implementation

WebRTC was designed to be encrypted end-to-end, however it was also designed to be peer-to-peer. Most video conferencing applications use a server component acting as the remote WebRTC peer and terminating the encryption. In the most recent version of the WebRTC 1.0 specifications [12], specific APIs and features support the sender simulcast use case. In that use case, a sending endpoint streams multiple copies of a media from the same source, but at different resolutions, to a media server. The media server then selectively forwards one of the incoming media streams to a remote receiving endpoint, depending on the receiver's capacity (bandwidth, CPU, display size, etc.) or user interface configuration. In a multiparty conference, this allows a media server to forward different streams to different peers (i.e., to function as a Selective Forward Unit or SFU).

Current implementations of simulcast for WebRTC, including Jitsi Videobridge, manage incoming simulcast, allowing receiving clients to be completely unaware that the technique is used on the sender side. In these implementations, the SFU switches the stream forwarded to a given receiver, modifying certain RTP header fields in order to mask the operation for the receiver and output one consistent RTP stream. It might, for example, switch from sending a high-resolution stream to a lower resolution stream (and back) as the available bandwidth fluctuates. Specifically, SFUs modify the Sequence Number (from now on `SEQ`), Synchronization Source (`SSRC`), and Timestamp (`TS`) fields in the RTP header.

In this configuration, the media stream is only encrypted between the sender and the SFU, and the SFU and the receiver, not within the server itself. Thus, anyone tampering with the media server can access unencrypted media content. At the system level, the PERC working group calls this *Hop-by-Hop* (HBH) encryption, as opposed to the *End-to-End* (E2E) encryption protecting media all the way between the sender and receiver(s).

The PERC working group produced a draft [38] describing a mechanism for systemic end-to-end encryption in conferences that use a Media Distributor (MD). In this context, the role of an MD can be served by an SFU and we henceforth use the two terms interchangeably. The PERC system consists of endpoints using two separate cryptographic contexts: end-to-end (E2E, or "inner") and hop-by-hop (HBH, or "outer"). The MD has access to the HBH context, but not the E2E context. Media packets are encrypted twice, once with the E2E context, and once with the HBH context. The MD then re-applies the HBH encryption with the correct context for each receiver. Control packets (RTCP) are not E2E encrypted. Figure 9.1 illustrates how this "double" system works. Note that RTCP packets are only handled HBH.

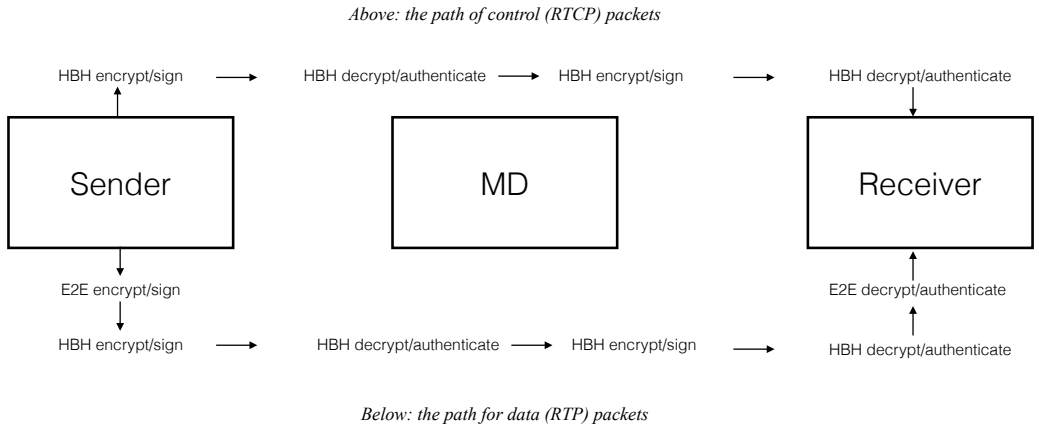


Figure 9.1: *The PERC procedure for End-to-End and Hop-By-Hop encryption for RTP packets (below) and RTCP packets (above).*

The PERC draft uses two RTP header extensions [71] to allow an MD to perform selective forwarding. The Frame Marking extension [19] is added by media senders, and includes additional information about an RTP packet, which an MD needs to make a forwarding decision. The Original Header Block (OHB) extension [38] is added by the MD whenever it modifies any of the RTP header fields. This allows the receiver to reconstruct an exact copy of the original E2E-encrypted RTP packet, and thus decrypt and verify it.

In this chapter, we discuss an implementation of the scheme proposed in the "double" draft [38], with some modifications. We do not examine the problem of distributing the keys needed for E2E encryption or that of identity verification.

The following two sections (9.1.1 and 9.1.2) describe the RTP Header Extensions and the modifications we propose. The rest of the sections (9.1.3 through 9.1.7) discuss the main problems we identified during the implementation, and the solutions we propose.

9.1.1 Original Header Block Extension

The Original Header Block (OHB) RTP header extension is defined in the "double" draft [38]. Its purpose is to allow an MD to modify a subset of the RTP header fields while still allowing receivers to perform packet authentication end-to-end. This is accomplished by having the MD add the original values of the modified fields in an OHB

Table 9.1: *The fields encoded in the Original Header Block header extension for different values of the length field and the R-bit.*

len	R-bit set	R-bit not set
0	PT	
1	SEQ	
2	PT, SEQ	
3	SSRC	
4	PT, SSRC	PT, TS
5	SEQ, SSRC	
6	PT, SEQ, SSRC	PT, SEQ, TS
7	SSRC, TS	
8	PT, SSRC, TS	
9	SEQ, SSRC, TS	
10	PT, SEQ, SSRC, TS	

extension. The receiver removes the extension and performs authentication using the original values. It then processes the packet further using the MD-modified values. The current version of the draft defines a structure which can contain at most the original RTP Payload Type (from now on PT) and SEQ fields, restricting the MD to only modifying these two fields. However, as previously explained, in the simulcast use case the MD may also need to modify the SSRC and TS.

We propose to keep the general scheme, and preserve the semantics of the OHB extension, while extending its structure to contain the additional fields needed for the WebRTC 1.0 simulcast use case. To optimize the overhead we keep using the 4-bit length field of the header extension header [71] to encode the structure of the remaining bytes. Specifically, we use it to encode whichever of the four fields are present in the following bytes: Payload Type (PT, 1 byte, including an extra R-bit), Sequence Number (SEQ, 2 bytes), SSRC (4 bytes), Timestamp (TS, 4 bytes).

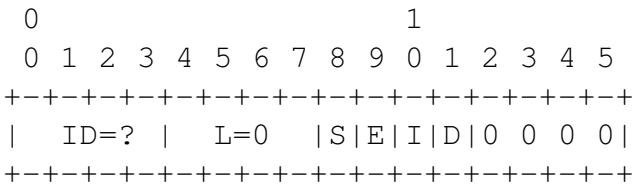
If the length field has a value of 0, 1, or 2 (indicating that there are, respectively, 1, 2 or 3 more bytes), then the format is identical to the current draft. This provides backward compatibility. That is, an endpoint implementing the extended format can also work with the current format without modification.

Table 9.1 lists the encoded fields (and the order in which they are included) for the different values of the length field and the R-bit. The R-bit is the most significant bit (MSB) of the `PT` field, and is present if and only if the `PT` field is present. If the extension includes the `PT` field, it consists of an R-bit (MSB) and 7 bits for the original RTP Payload Type. The R-bit is used to encode whichever fields are included when there is ambiguity.

9.1.2 Frame Marking

To achieve selective forwarding, the MD needs to know whether the packet is part of a keyframe or not. This information is not included in the RTP header of a packet and is usually part of the payload. Existing implementations extract this from the payload in a codec-specific way. With PERC this is no longer possible, because the payload is E2E encrypted and therefore unavailable to the MD. The framemarking draft[19] provides a solution to this problem. It defines an RTP header extension containing additional information about a packet (such as whether the packet is part of an independent frame). RTP header extensions are not E2E encrypted and thus, always remain available to the MD.

The details of the Header Extensions are as follows, and were implemented in our solution.



Note that we use the first of the remaining 4 bits to signal padding packets, see section 9.1.5.

9.1.3 Injecting Video Frames

Obviously one of the goals of the PERC architecture is to prevent an MD from modifying or injecting media content in a conference that it hosts. However, there are certain scenarios when injecting media may be useful.

For example, due to a long-standing bug in the Chrome / Chromium browser[2], the audio from a `MediaStream` which contains both audio and video tracks is never played back until some video packets are received. To work around this in the case where a participant sends no video, our SFU injects empty video frames in the beginning of a stream. We had to disable this behaviour and fall back to using two separate `MediaStream` objects for the audio track and the video track on the receiver side. This removes the audio playback problem, but disables synchronization between the playback of audio and video on the receiver side.

9.1.4 Retransmissions with RTX

One of the mechanisms WebRTC uses to handle packet loss is packet retransmissions [30], and its effectiveness depends highly on the delay [53]. Thus, handling retransmissions in the SFU is desirable, because it can reduce the delay of retransmitted packets by a half on average (since the SFU is in between any two peers on the transport path). Usually, the SFU implements two mechanisms. It requests missing packets from the senders if it itself failed to receive them, and it responds to retransmission requests from receivers.

The RTX format [62] is used for RTP packet retransmission. It uses its own RTP payload type with its own SSRC and thus has its own sequence number space. When a packet is retransmitted, it inherits the original packet headers with the exception of these three fields (`PT`, `SSRC` and `SEQ`). The payload type and SSRC are paired via signaling with those of the corresponding media stream, so the original values can be reconstructed on the receiving end. However, the sequence number, is encoded in the Original Sequence Number (OSN) field, which is the first two bytes of the RTP payload. As part of the payload it is encrypted by SRTP.

In the case of the double encryption scheme, if RTX packets are handled like regular RTP packets, the MD cannot read the OSN in incoming RTX packets or create its own RTX packets (as this involves modifying the payload by inserting the OSN).

We propose a solution to this problem by using RTX hop-by-hop only. That is, the RTX transformation is applied after the E2E SRTP transformation, but before the HBH. This is the same way that PERC "double" handles RTCP packets, except of course that the input is an RTX packet which encapsulates and already E2E encrypted media packet.

Specifically, the procedure for a sender to retransmit a packet is the following: first, it finds the unencrypted packet to be retransmitted and applies the E2E SRTP transformation. Then, it applies the RTX transformation, consisting of: replacing the Payload

[2]https://bugs.chromium.org/p/webrtc/issues/detail?id=5254

Type and SSRC with those of the RTX stream, inserting the OSN as the first two bytes of the payload, and generating a new sequence number. Lastly, it applies the HBH SRTP transformation.

When an MD receives an RTX packet, it first applies the reverse HBH SRTP transformation. It then restores the original E2E-encrypted packet by replacing the Payload Type and SSRC fields with those of the associated media stream, and restores the sequence number from the OSN (first two bytes of the payload). At this point it can treat the restored packet as usual: send it to other endpoints after optionally encapsulating it in RTX and encrypting it with the corresponding HBH context. The same procedure applies to receiving endpoints, though after restoring the RTP packet it has to be E2E verified and decrypted.

The above scheme allows an MD to insert packets into an RTX stream without impacting an MD's inability to insert packets in a media stream, as packets transported in RTX are still protected with the E2E context.

9.1.5 Terminating Bandwidth Estimation

Part of the mechanism used in *webrtc.org* for bandwidth estimation is actively probing for additional available bandwidth [16] by sending extra padding packets. An MD should terminate the bandwidth estimation, as the available bandwidth normally varies per endpoint. Thus, packets sent for bandwidth estimation only and contain no actual media should not be forwarded to other endpoints.

The RTP padding mechanism uses the last byte of the payload to encode the padding length [70], and so the field is encrypted by SRTP as part of the payload [11]. This means that an MD is not able to read the padding length field for E2E encrypted packets, and it cannot differentiate between padding-only packets and those that contain (some) media. Thus, in the context of PERC, if additional data needs to be sent for the purposes of bandwidth estimation, it should not be sent as RTP padding.

There are multiple ways of solving this problem. We propose using the RTX stream along with a padding-only indication in a header extension in order to allow probing-only packets to be sent HBH (and HBH only). Using the RTX format on its own with no other changes does not work for two reasons: the last byte of the RTX payload is still E2E encrypted, and the padding bit in the RTP header must match between the inner and outer SRTP transformations, because the "double" draft does not include a way for the original value to be recovered. Using a separate padding-only bit in the header also

has the advantage of allowing padding-only packets with more than 256 bytes of data to be sent.

We propose using the frame marking header extension to carry the padding-only bit. As this bit is not intended for use in packets containing actual media, we only define it in the one-byte format used for non-scalable streams (and not in the 3-byte format for scalable streams; see section 9.1.2 and also Sections 3.1 and 3.2 from the frame marking draft [19]). We use the first of the remaining bits (originally defined as 0) as the "padding-only" bit. We do not reuse the "discardable frame" bit because it has different semantics (the packet contains media, though it may be discarded if necessary).

9.1.6 RTP Timestamps

The current implementation of simulcast in *webrtc.org* (used in Chrome, Firefox, and Safari) uses different initial offsets for the RTP timestamp for each simulcast stream. This forces the SFU to rewrite the timestamps of the single RTP stream it outputs.

However, in the context of PERC allowing an MD to modify the RTP time stamp has has undesirable security implications. Namely, the possibility of the MD performing replay attacks. These attacks are prevented by the verification of the RTP sequence number.

We choose to allow rewriting of the timestamp field by the MD as described in sections 9.1 and 9.1.1 above. Our reasoning is the ease of implementation in existing software and the limited impact on security during local tests. This should not be allowed in real systems.

One potentially viable solution is for simulcast senders to use the same offset for all simulcast streams coming from a unique source. This removes the need for timestamp modification by the MD, while also allowing the receiver to verify the timestamps of all simulcast encodings end-to-end, preventing a form of attack where the MD does not initially forward one of the input streams, and then replays it at a later time.

Since RTP timestamps use a 32-bit integer field that wraps around, some form of replay attacks may still be possible. To prevent this, we suggest the conference be re-keyed often enough. In the case of the usual $90kHz$ RTP clock rate, it takes approximately 13 hours for the RTP timestamps to cycle, so re-keying should happen at least that often.

9.1.7 Signaling Double Encryption

An SFU needs two pieces of information to function in the double encryption scheme described above. These are the ID numbers for the two header extensions: framemarking and Origianl Header Block.

If an SFU needs to differentiate between the cases of double encryption being enabled or not, it can do so solely based on the presence of the OHB header extension. That is, if OHB is signaled, it always adds the OHB extension to RTP packets and follows the semantics defined in the "double" draft for handling other RTP header extensions. Otherwise, it should not add an OHB extension and is free to add, remove or modify other header extensions.

9.2 Evaluation

We implemented a proof-of-concept system by modifying the Jitsi Videobridge SFU and the Chromium browser. We used the unmodified Jitsi Meet web application. The system implements the proposals described in the section above and uses pre-generated and shared between the endpoints end-to-end encryption keys. At this stage, we are only assessing the performance characteristics and feasibility of the "double" system, and not the key exchange and verification aspects of any full-fledged system.

We performed measurements of CPU usage and analyzed the overhead incurred due to the additional header extensions we propose. We present the results in the sections below.

9.2.1 CPU Performance

As we saw in Chapter 7, the process of encryption/decryption (including signing and verification) can constitute a significant portion of an SFU's workload. However, the "double" system does not change the performance characteristics of an SFU, because it still only performs a single decrypt/encrypt operation, it just happens that the payload remains end-to-end encrypted.

Endpoints, however, must perform two encryption and decryption operations for each packet. We measured the CPU usage during participation in a one-to-one call with and without double encryption on a modern laptop (a 2015 15" MacBook Pro). We were unable to detect any difference between the two cases, on either the sending or receiving

endpoint. This is explained by the encoding/decoding process being much more CPU intensive than encryption/decryption.

9.2.2 Overhead Analysis

The original "double" SRTP encryption scheme used in PERC has adds some overhead. At minimum 4 extra bytes per RTP packet come from the additional SRTP [11] transformation and fields added by the EKT format [39]. The Original Header Block RTP Header Extension adds between 0 (when no headers were modified) and 8 bytes (when both the Payload Type and Sequence Number were modified, and the original packet contains no header extensions) to each RTP packet sent from the MD to receivers.

Our extended OHB format adds between 0 and 16 bytes (when all of `PT`, `SEQ`, `TS` and `SSRC` were modified and the original packet has no header extensions). However, it does not change the overhead compared to the current OHB format unless the extra features (modifying the `SSRC` and/or `TS` fields) are used.

Together, the overhead is between 4 and 20 bytes. In our implementation it is always 4 for audio packets and 16 for most video packets. This is because header extensions are already in use, the MD does not need to modify any headers for the audio streams, and modifies the `SEQ`, `SSRC` and `TS` fields for most video packets.

In a typical WebRTC call scenario using the Opus codec with $20\,\mathrm{ms}$ frames and the default bitrate of $40\,\mathrm{Kbps}$, we observe an average packet size of around 100 bytes. Thus the overhead between sending endpoints and the MD is 4%, and can be as high as 20% between the MD and receiving endpoints, though in most cases it is 4%.

For video, the average packet size is usually between 600 and 1200 bytes, depending on the bitrate. Thus, the overhead between sending endpoints and the MD is less than 0.7%, while the overhead between the MD and receiving endpoints is up to 4% (and 2.7% on average in our system).

For a combined audio and video call, the overhead is closer that of video, as video comprises most of the packets.

9.3 Related Work

The PERC working group at the IETF continues to be active, and there have been other proposals addressing some of the same problems that we discuss (some of them submitted after our work was originally written).

One idea is to replace the complex OHB mechanism and instead encapsulate the entire end-to-end encrypted RTP packet (together with its headers and authentication block) when applying the hop-by-hop encryption. This results in some additional overhead, but it is likely insignificant.

The question of handling RTX and other stream repair mechanisms such as Forward Error Correction (FEC) [88] has been the topic of much discussion. While it is agreed that there is a need for such mechanisms to be used hop-by-hop, there is no consensus on the best way to achieve it.

One proposal is to simply send packets for stream repair without additional protection. This is not viable, because lacking the E2E context an MD cannot authenticate incoming packets, which may enable certain DoS attacks when used with FEC.

Another proposal referred to as "triple", consists of applying the hop-by-hop encryption to stream repair packets twice. This adds extra complexity, because the mechanism is different from the way the encryption of RTP and RTCP packets is handled by PERC. The second encryption round, since it uses the same context, does not have any useful cryptographic properties. It also requires modifications to the way SRTP is performed, because with normal operation the same context cannot be used twice for a packet. This is because SRTP replay protection is based on the RTP sequence number of the packet, which does not change. This proposal is now included in the latest version of the double draft [38].

9.4 Conclusion and Future Work

In this chapter, we illustrated a proof-of-concept implementation of the proposed PERC standard. As far as we are aware, this is the first such implementation. Out of this work it is clear that much of what is currently implemented in the browser is still quite proprietary, and supporting Chrome in our case required extra work. However, solutions to almost all the problems we faced already existed in some specifications, e.g., framemarking [19] or "diet"EKT [39], and this paper provides the details necessary to achieve an implementation. We also showed that while the current OHB format is in-

sufficient, it can be easily extended to achieve what is needed. Finally, we demonstrated that the performance overhead is negligible in term of CPU usage and very low ($< 4\%$) in terms of bandwidth. For the intended use case where E2E encryption is necessary for compliance reasons, the additional bandwidth is completely acceptable.

Chapter 10

Conclusions and Perspectives

This chapter concludes the thesis, summarizing the addressed problems and pointing out the contributions. It also offers perspectives for further research.

10.1 Conclusion

The primary goal of this thesis was to propose and evaluate ways of increasing the efficiency of video conferencing services using Selective Forwarding Units and the WebRTC framework, as well as to improve the overall user experience.

We began by considering the bandwidth requirements for a conference, which are generally higher when using an SFU, as compared to other approaches such as using mixing servers (MCUs).

In Chapter 4, we proposed a selective forwarding scheme using speech activity to automatically follow the active speaker in a conference. We proposed an algorithm for "dominant speaker identification" that works using audio level indications as opposed to raw audio streams, making it suitable for use in an SFU. We compared the selective forwarding system to a full-star topology in which all streams are forwarded, and observed significant reduction in the required bandwidth – from quadratic to linear growth with the number of endpoints.

Next, in Chapter 5, we presented a thorough evaluation of the use of WebRTC simulcast for video conferencing. The feature further reduces the bandwidth required for a conference, at the price of a slight decrease in image quality. The computational overhead was negligible.

In Chapter 6, we proposed a hybrid architecture using a peer-to-peer topology for conferences of size two, and an SFU for larger conferences. We evaluated the effects in a production system and observed significantly reduced server traffic as well as improved connection quality.

Chapter 7, focused on the performance of SFU software implementations. We detailed an efficient algorithm for selective forwarding and observed lowered CPU usage. We proposed a technique for memory management aimed to reduce the strain on the Java garbage collector (GC). We developed a methodology to measure the side effects of GC (temporary pausing of all application threads), and performed an experiment to determine the GC configuration most suitable for use in an SFU. We found that the *Concurrent Mark and Sweep* algorithm performs best, and that our proposed technique improves the results regardless which GC algorithm is used.

In Chapter 8, we explored a distributed server architecture aimed at reducing the latency in a conference by connecting endpoints to servers in close geographical proximity. We performed a randomized experiment and found that overall, the latency to the immediate server was slightly lower, but it had the unintended effect of significantly increasing end-to-end latency. Based on the distribution of users in a conference and other factors, the system can have either a positive or a negative effect, and the challenge is in understanding these factors to perform server selection in an optimal way. To this end, we propose further research.

Chapter 9 diverged from the main topic of efficiency and pursued another goal of this thesis – supporting novel features using SFU architectures. It detailed the implementation of an end-to-end encrypted conference system with WebRTC and simulcast. We proposed extensions to the existing PERC specification necessary to support simulcast and we showed the overhead is negligible.

10.2 Perspectives

Some of the results in this thesis presented opportunities for further research. This section lists some of them.

10.2.1 Memory management

One of the challenges we faced in Chapter 7 was the lack of knowledge about what constitutes acceptable performance for the distribution of garbage collection pauses.

The effects of these pauses on video conferencing have not been explored. We observe very long pauses triggering freezes in the video playback, but we don't know at what pause length the effect appears. We hypothesise that longer and more frequent pauses will have an effect on receivers' jitter buffers, forcing them to grow larger and increasing end-to-end delay. Both of these questions can be answered experimentally by artificially delaying packets in the SFU and observing the jitter buffer length.

In our experiments we only tested the default configuration of the G1 algorithm. Further testing with different G1 constraints, and other algorithms (like Shenandoah), might reveal better configurations.

Finally, moving the buffers used for RTP packet contents out of the Java heap has the potential to significantly reduce the strain on the garbage collector and thus, the GC pause duration and frequency. The risk with this approach is that memory leaks are not caught by the Java garbage collector, and instead accumulate until the application runs out of memory. Quantifying the effects can only be done experimentally.

10.2.2 Simulcast, Scalable Video Coding, and Different Video Codecs

Scalable Video Coding (SVC) is a technique that provides features similar to simulcast. An SFU receiving a scalable stream can produce different encodings by selecting a subset of the stream's layers. As far as we are aware no comparison between the two approaches for use in video conferencing has been published.

A comparison would have to consider the video codec(s) to use. Simulcast is codec independent, but some codecs offer native implementations with enhanced performance. As we showed in Chapter 5 the VP8 implementation of simulcast in *webrtc.org* is very efficient in terms of encoding complexity, but this does not automatically translate to other codecs. In contrast, SVC needs explicit support on the codec level. The VP9 codec supports SVC with temporal, spacial and signal-to-noise ratio scalability. However, it is currently not as widely supported as VP8, which is in fact mandatory to implement in WebRTC.

The VP8 simulcast implementation also supports temporal scalability, which can be used to further lower the bitrate when there is insufficient bandwidth.

There are multiple studies evaluating video codec efficiency, but most of them focus on the storage or live streaming scenarios. The new generation of codecs like HEVC and VP9 offer much better compression ratio at the price of encoding complexity. Video conferencing requires near real-time encoding, which makes the case significantly different.

10.3 Cascaded Selective Forwarding Units

The server selection strategy we proposed in Chapter 8 was disappointing and ultimately unsuccessful in lowering end-to-end latency. However the distributed server architecture as a whole still has the potential to improve connection quality, if we can find a better selection strategy.

The first step in this direction would be to understand the situations in which the simple strategy fails. One hypothesis is that the region detection itself is not working correctly. We have started a project to test this, by directly measuring endpoints' round trip time to all of the available regions.

Another hypothesis is that a lot of conferences have endpoints with similar proximity to more than one server (as illustrated on figure 8.10). This can also be confirmed or refuted by the experiment mentioned above.

The positive results we observe in the us-west region (see Section 8.4) suggest that some of the region-wide characteristics affect the outcome. Perhaps this region is more isolated, with few of its endpoints being also close to other regions. Studying the characteristics of this region will offer further insight. If this is the case, limiting the distributed server conferences to one server per continent may be a good solution.

As we mention in Section 8.1, lower RTT to the next-hop server is beneficial for stream repair. Quantifying this effect would be useful when evaluating different strategies. With distributed servers we observe a decrease of $8\,\mathrm{ms}$ on average, but we don't know how significant this effect is.

Our aim is to improve connection quality, but so far our tests have concentrated on measuring the delay. This is because delay is an easy to measure proxy for connection quality. Measuring other metrics, such as packet loss, jitter, and achieved stable bitrate may offer additional insight.

One area that we haven't explored so far is optimizing the server-to-server traffic. The good first step may be to apply the LastN and simulcast techniques from Chapters 4 and 5, adapted for the server-to-server use case.

Finally, the distributed server architecture has a completely different use-case that may prove more impactful – supporting very large conferences. A single server can only handle a limited number of endpoints, and by using an appropriate server-to-server topology we can significantly increase it. This is an area that we plan to explore in the future.

Bibliography

[1] Apache license, 2004.

[2] UK Home Broadband Performance
The performance of fixed-line broadband delivered to UK residential consumers, May 2018.

[3] Wire security whitepaper, August 2018.

[4] Iso/iec 23009-1:2019: Dynamic adaptive streaming over http (dash) - part 1: Media presentation description and segment formats, August 2019.

[5] H Alvestrand. RTCP message for Receiver Estimated Maximum Bitrate. `https://tools.ietf.org/html/draft-alvestrand-rmcat-remb-03`, October 2013.

[6] H. Alvestrand and V. Singh. Identifiers for WebRTC's Statistics API. `https://www.w3.org/TR/2018/CR-webrtc-stats-20180703/`, 2018.

[7] Peter Amon, Madhurani Sapre, and Andreas Hutter. Compressed domain stitching of hevc streams for video conferencing applications. In *Packet Video Workshop (PV), 2012 19th International*, pages 36–40. IEEE, 2012.

[8] Zlatka Avramova, Danny De Vleeschauwer, Kathleen Spaey, Sabine Wittevrongel, Herwig Bruneel, and Chris Blondia. Comparison of simulcast and scalable video coding in terms of the required capacity in an IPTV network. In *Packet Video 2007*, pages 113–122. IEEE, 2007.

[9] J. Bankoski, J. Koleszar, L. Quillio, J. Salonen, P. Wilkins, and Y. Xu. VP8 Data Format and Decoding Guide. RFC 6386, IETF, November 2011.

[10] Salman Baset and Henning Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. *Proceedings - IEEE INFOCOM*, 01 2005.

[11] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman. The Secure Real-time Transport Protocol (SRTP). RFC 3711, IETF, March 2004.

[12] Adam Bergkvist, Daniel C. Burnett, Cullen Jennings, Anant Narayanan, Bernard Aboba, Taylor Brandstetter, and Jan-Ivar Bruaroey. WebRTC 1.0: Real-time Communication Between Browsers. `https://www.w3.org/TR/2018/CR-webrtc-20180927/`, 2018.

[13] Frank Bossen, Benjamin Bross, Karsten Suhring, and David Flynn. Hevc complexity and implementation analysis. *IEEE Trans. Cir. and Sys. for Video Technol.*, 22(12):1685–1696, December 2012.

[14] E. Brosh, S.A. Baset, D. Rubenstein, and H. Schulzrinne. The delay-friendliness of tcp. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '08, pages 49–60, New York, NY, USA, 2008. ACM.

[15] Daniel C. Burnett, Adam Bergkvist, Cullen Jennings, Anant Narayanan, Bernard Aboba, Jan-Ivar Bruaroey, and Henrik Boström. Media Capture and Streams. `https://www.w3.org/TR/2019/CR-mediacapture-streams-20190702/`, 2019.

[16] 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*, MMSys '16, pages 13:1–13:12, New York, NY, USA, 2016. ACM.

[17] Marcel Dischinger, Andreas Haeberlen, Krishna P. Gummadi, and Stefan Saroiu. Characterizing residential broadband networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, pages 43–56, New York, NY, USA, 2007. ACM.

[18] Richard Dosselmann and Xue Dong Yang. A comprehensive assessment of the structural similarity index. *Signal, Image and Video Processing*, 5(1):81–91, 2011.

[19] E. Berger, S. Nandakumar and M. Zanaty. Frame Marking RTP Header Extension, March 2017. IETF Internet Draft.

[20] Alexandros Eleftheriadis, Reha Civanlar, and Ofer Shapiro. Multipoint videoconferencing with scalable video coding. *Journal of Zhejiang University SCIENCE A*, 7(5):696–705, May 2006.

[21] Jeroen Famaey, Steven Latré, Niels Bouten, Wim Van de Meerssche, Bart Vleeschauwer, Werner Van Leekwijck, and Filip Turck. On the merits of svc-based http adaptive streaming. 05 2013.

[22] Yao-Chun Fang, Chun-Yi Lee, Yin-Ming Wang, Wang Chung-Neng, and Tihao Chiang. Low complexity lossless video compression. In *International Conference on Image Processing*, Singapore, Singapore, October 2004. IEEE Press.

[23] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455, IETF, December 2011.

[24] Scott Firestone, Thiya Ramalingam, and Steve Fry. *Voice and Video Conferencing Fundamentals*. Cisco Press, first edition, 2007.

[25] M. Ghanbari. *Standard Codecs: Image Compression to Advanced Video Coding*. Institution Electrical Engineers, 2003.

[26] M. Handley, V. Jacobson, and C. Perkins. SDP: Session Description Protocol. RFC 4566, IETF, July 2006.

[27] R. Harrison and K. Zeilenga. The Lightweight Directory Access Protocol (LDAP) Intermediate Response Message. RFC 3771, IETF, April 2004.

[28] P. Helle, H. Lakshman, M. Siekmann, J. Stegemann, T. Hinz, H. Schwarz, D. Marpe, and T. Wiegand. A Scalable Video Coding Extension of HEVC. pages 201–210. IEEE, March 2013.

[29] S. Holmer, M. Flodman, and E. Sprang. RTP Extensions for Transport-wide Congestion Control. `https://tools.ietf.org/html/draft-holmer-rmcat-transport-wide-cc-extensions-01`, October 2015.

[30] Stefan Holmer, Mikhal Shemer, and Marco Paniconi. Handling packet loss in webrtc. In *International Conference on Image Processing (ICIP 2013)*, pages 1860–1864, 2013.

[31] David Hyatt Ian Hickson. HTML5: A vocabulary and associated APIs for HTML and XHTML. http://www.w3.org/TR/html5/, September 2010.

[32] ITU-T. ITU-T recommendation G.114. Technical report, International Telecommunication Union, 2003.

[33] ITU-T Rec. G.711. G.711 : Pulse code modulation (pcm) of voice frequencies.

[34] ITU-T Rec. H.264. Advanced video coding for generic audiovisual services.

[35] Emil Ivov. Optimizing real-time communications over the internet protocol, 2008.

[36] Emil Ivov, Lyubomir Marinov, and Philipp Hancke. Xep-0340: Conferences with lightweight bridging (colibri), January 2014.

[37] Julian Jang-Jaccard, Surya Nepal, Branko Celler, and Bo Yan. Webrtc-based video conferencing service for telehealth. *Computing*, 98(1-2):169–193, January 2016.

[38] C. Jennings, P. Jones, R. Barnes, and A. Roach. SRTP Double Encryption Procedures, 2017. IETF Internet Draft.

[39] C. Jennings, J. Mattsson, D. McGrew, D. Wing, and F. Andreasen. Encrypted Key Transport for Secure RTP, 2016. IETF Internet Draft.

[40] R. Jesup, S. Loreto, and M. Tuexen. WebRTC Data Channels. `https://tools.ietf.org/html/draft-ietf-rtcweb-data-channel-13`, January 2015.

[41] Hari Kalva, Velibor Adzic, and Borko Furht. Comparing mpeg avc and svc for adaptive http streaming. pages 158 –159, 01 2012.

[42] Gregorij Kurillo, Allen Y. Yang, Victor Shia, Aaron Bair, and Ruzena Bajcsy. New emergency medicine paradigm via augmented telemedicine. In Stephanie Lackey and Randall Shumaker, editors, *Virtual, Augmented and Mixed Reality*, pages 502–511, Cham, 2016. Springer International Publishing.

[43] J. Lazzaro. Framing Real-time Transport Protocol (RTP) and RTP Control Protocol (RTCP) Packets over Connection-Oriented Transport. RFC 4571, IETF, July 2006.

[44] J. Lennox, E. Ivov, and E. Marocco. A Real-time Transport Protocol (RTP) Header Extension for Client-to-Mixer Audio Level Indication. RFC 6464, IETF, December 2011.

[45] J. Lennox, J. Ott, and T. Schierl. Source-Specific Media Attributes in the Session Description Protocol (SDP). RFC 5576, IETF, June 2009.

[46] Jonathan Lennox and Henning Schulzrinne. A protocol for reliable decentralized conferencing. In *Proceedings of the 13th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '03, pages 72–81, New York, NY, USA, 2003. ACM.

[47] A. Li. RTP Payload Format for Generic Forward Error Correction. RFC 5109, IETF, December 2007.

[48] Yuan Jiang Liao, Zhonghua Wang, and Yanfen Luo. The design and implementation of a webrtc based online video teaching system. *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, pages 137–140, 2016.

[49] Scott Ludwig, Joe Beda, Peter Saint-Andre, Robert McQueen, Sean Egan, and Joe Hildebrand. XEP-0166: Jingle, 2005.

[50] R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766, IETF, April 2010.

[51] Lyubomir Marinov. Equalization of silence audio levels in packet media conferencing systems, 2015. US Patent 9325853.

[52] J. Mattsson, D. McGrew, D. Wing, and F. Andreasen. Encrypted Key Transport for Secure RTP, October 2014. IETF Internet Draft.

[53] Marcin Nagy, Varun Singh, Jörg Ott, and Lars Eggert. Congestion control using fec for conversational multimedia communication. In *Proceedings of the 5th ACM Multimedia Systems Conference*, MMSys '14, pages 191–202. ACM, 2014.

[54] S. Nandakumar and C. Jennings. Annotated Example SDP for WebRTC. `https://datatracker.ietf.org/doc/draft-ietf-rtcweb-sdp/`, October 2018.

[55] William B Norton. Internet service providers and peering. In *Proceedings of NANOG*, volume 19, pages 1–17, 2001.

[56] Orrin E. Dunlap. *The Outlook for Television*. Harper & Brothers Publishers, New York, USA, 1932.

[57] J. Ott, S. Wenger, N. Sato, C. Burmeister, and J. Rey. Extended RTP Profile for Real-time Transport Control Protocol (RTCP)-Based Feedback (RTP/AVPF). RFC 4585, IETF, July 2006.

[58] J. Postel. User Datagram Protocol. RFC 768, IETF, August 1980.

[59] J. Postel. Transmission Control Protocol. RFC 793, IETF, September 1981.

[60] E. Rescorla. WebRTC Security Architecture. `https://tools.ietf.org/html/draft-ietf-rtcweb-security-arch-20`, July 2019.

[61] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, IETF, January 2012.

[62] J. Rey, D. Leon, A. Miyazaki, V. Varsa, and R. Hakenberg. RTP Retransmission Payload Format. RFC 4588, IETF, July 2006.

[63] Roach, A., Nandakumar, S., and P. Thatcher. RTP Stream Identifier (RID) Source Description (SDES), October 2016. IETF Internet Draft.

[64] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. RFC 5245, IETF, April 2010.

[65] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389, IETF, October 2008.

[66] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261, IETF, June 2002.

[67] J. Rosenberg, H. Schulzrinne, and O. Levin. A Session Initiation Protocol (SIP) Event Package for Conference State. RFC 4575, IETF, August 2006.

[68] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120, IETF, March 2011.

[69] Salnikov-Tarnovski, Nikita and Smirnov, Gleb. *Plumbr Handbook Java Garbage Collection*. Plumbr.

[70] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550, IETF, July 2003.

[71] D. Singer and H. Desineni. A General Mechanism for RTP Header Extensions. RFC 5285, IETF, July 2008.

[72] D. Singer, H. Desineni, and R. Even. A General Mechanism for RTP Header Extensions. RFC 8285, IETF, October 2017.

[73] Varun Singh. *Protocols and Algorithms for Adaptive Multimedia Systems*. School of Electrical Engineering, Aalto University, Helsinki, Finland, 2015.

[74] Varun Singh, Albert Abello Lozano, and Jörg Ott. Performance analysis of receive-side real-time congestion control for webrtc. In *Proc. of IEEE Workshop on Packet Video*, PV '13, 2013.

[75] J. Spittka, K. Vos, and JM. Valin. RTP Payload Format for the Opus Speech and Audio Codec. RFC 7587, IETF, June 2015.

[76] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663, IETF, August 1999.

[77] Thatcher, P., Zanaty, M., Nandakumar, S., Burman, B., Roach, A., and B. Campen. RTP Payload Format Restrictions, March 2017. IETF Internet Draft.

[78] J. Uberti, C. Jennings, and E. Rescorla. JavaScript Session Establishment Protocol. https://tools.ietf.org/html/draft-holmer-rmcat-transport-wide-cc-extensions-01, February 2019. IETF Internet Draft.

[79] JM. Valin and C. Bran. WebRTC Audio Codec and Processing Requirements. RFC 7874, IETF, May 2016.

[80] JM. Valin, K. Vos, and T. Terriberry. Definition of the Opus Audio Codec. RFC 6716, IETF, September 2012.

[81] Ilana Volfin and Israel Cohen. Dominant speaker identification for multipoint videoconferencing. *Computer Speech Language*, 27(4):895 – 910, 2013.

[82] Yubing Wang. Survey of objective video quality measurements. 2006.

[83] S. Wenger, U. Chandra, M. Westerlund, and B. Burman. Codec Control Messages in the RTP Audio-Visual Profile with Feedback (AVPF). RFC 5104, IETF, February 2008.

[84] M. Wenzel and C. Meinel. Full-body webrtc video conferencing in a web-based real-time collaboration system. In *2016 IEEE 20th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 334–339, May 2016.

[85] M. Westerlund and S. Wenger. RTP Topologies. RFC 7667, IETF, November 2015.

[86] P. Westin, H. Lundin, M. Glover, J. Uberti, and F. Galligan. RTP Payload Format for VP8 Video. RFC 7741, IETF, March 2016.

[87] H. R. Wu and K. R. Rao. *Digital Video Image Quality and Perceptual Coding (Signal Processing and Communications)*. CRC Press, Inc., Boca Raton, FL, USA, 2005.

[88] M. Zanaty, V. Singh, A. Begen, and G. Mandyam. RTP Payload Format for Flexible Forward Error Correction (FEC), July 2019.

[89] Fan Zhai, Y. Eisenberg, T. N. Pappas, R. Berry, and A. K. Katsaggelos. Rate-distortion optimized hybrid error control for real-time packetized video transmission. *Trans. Img. Proc.*, 15(1):40–53, January 2006.

[90] Virginie Zoumenou, Madeleine Sigman-Grant, Gayle Coleman, Fatemeh Malekian, Julia M. K. Zee, Brent J. Fountain, and Akela Marsh. Identifying best practices for an interactive webinar. 2015.

Appendices

Appendix A

List of publications

This thesis is in part based on previous research by the author, published in several peer-reviewed international journals and conferences, and described in several US patents and IETF internet drafts. These publications are listed below.

International journals

[Pub1] **Experimental Evaluation of Simulcast for WebRTC**, Boris Grozev, George Politis, Emil Ivov, Thomas Noël, IEEE Communications Standards Magazine (Volume 1, Issue 2, 2017)

This paper details the implementation of simulcast in a video conferencing system based on WebRTC and presents a thourough evaluation of its effects on image quality, server load, and client load.

The author of this dissertation is the main author of this paper. He contributed to the ideas and concepts discussed in the paper. He also implemented the algorithm for preemptive keyframe requests, designed and conducted the experiments, analyzed the results, and edited the manuscript.

International Conferences

[Pub2] **Experimental Evaluation of Dynamic Switching between One-on-One and Group Video Calling**, George Politis, Boris Grozev, Pawel Domas, Emil Ivov, Thomas Noël, 2018 Principles, Systems and Applications of IP Telecommunications (IPTComm), Chicago, IL, USA, 16-18 Oct. 2018

This paper proposes a system which switches dynamically between a server-based and peer-to-peer connection during a conference, for the purpose of improving

user experience and reducing service cost. It presents an experimental evaluation of both goals.

The author of this dissertation was one of the co-authors of this paper. His contributions consisted of designing and performing the experiments for the infrastructure resource use, and co-editing the manuscript.

[Pub3] **Considerations for deploying a geographically distributed video conferencing system**, Boris Grozev, George Politis, Emil Ivov, Thomas Noël, 2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, 8-10 Jan. 2018

This paper presents a preliminary study of a multi-server video conferencing architecture, connecting each user to a server in their local region, with the goal of reducing the end-to-end next-hop latency.

The author of this dissertation was the main author of this paper. He contributed the ideas and concepts discussed in the paper, designed and conducted the experiments, analyzed the results, and edited the manuscript.

[Pub4] **PERC double media encryption for WebRTC 1.0 sender simulcast**, Boris Grozev, Emil Ivov, Arnaud Budkiewicz, Ludovic Roux, Alexandre Gouaillard, 2017 Principles, Systems and Applications of IP Telecommunications (IPTComm), Chicago, IL, USA, 25-28 Sept. 2017

This paper proposes an extension to the PERC specification for end-to-end encrypted video conferences, which enables the use of simulcast. It also proposes a novel way of handling stream repair. It details the proposed extension header format, and analyzes the overhead.

The author of this dissertation was the main author of this paper. He contributed the ideas discussed in the paper, implemented a proof-of-concept for the proposed new format, performed the overhead analysis, and edited the manuscript.

[Pub5] **Last N: Relevance-Based Selectivity for Forwarding Video in Multimedia Conferences**, Boris Grozev, Lyubomir Marinov, Varun Singh, Emil Ivov, Proceedings of the 25th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video Portland, Oregon — March 18 - 20, 2015

This paper proposes a scheme for endpoint selection based on speech activity. It presents an efficient algorithm for dominant speaker identification, which works using audio level indications instead of decoding audio streams, making it usable in a Selective Forwarding Unit architecture. The propose scheme is experimentally compared to a full-star architecture in terms of bandwidth usage.

The author of this dissertation was the main author of this paper. He contributed to the ideas and concepts discussed in the paper. He implemented an early proof-of-concept version of the dominant speaker detection algorithm, performed the experiments and analyzed the results.

Internet Drafts

[Pub6] **Using RTX with Privacy Enhanced RTP Conferencing (PERC)**, Boris Grozev, Emil Ivov, Alexandre Gouaillard, Work in Progress, Internet Engineering Task Force, draft-grozev-perc-double-rtx-00[1], March 27, 2017

This draft proposes an alternative way of handling stream repair, and retransmission using the RTX format in particular, in the PERC specification for end-to-end encrypted conferencing.

The author of this dissertation contributed the idea, and co-edited the text.

[Pub7] **Allowing Synchronisation Source (SSRC) and Timestamp Rewriting in Privacy Enhanced RTP Conferencing (PERC)**, Boris Grozev and Emil Ivov, Work in Progress, Internet Engineering Task Force, draft-grozev-perc-ssrc-00 [2], March 28, 2017

This draft extends the PERC specification to allow a Media Distributor to modify RTP packets' Synchronization Source (SSRC) and Timestamp fields, in order to support simulcast.

The author of this dissertation contributed the idea, and co-edited the text.

Patents

[Pub8] **Rapid optimization of media stream bitrate**, Emil Ivov, Boris Grozev, Georgios Politis, United States patent US20190245945A1, January 2019

This publication describes a system for optimal selection of initial bitrate for a video conferencing session. A database is used to save a history of achieved stable bitrates between any two given endpoints.

The author of this dissertation contributed to the initial concept, and editing of the description.

[Pub9] **On demand in-band signaling for conferences**, Emil Ivov, Boris Grozev, United States patent US10171526B1, January 2019

[1] https://tools.ietf.org/html/draft-grozev-perc-double-rtx-00
[2] https://tools.ietf.org/html/draft-grozev-perc-ssrc-00

This publication describes a system that uses an additional signaling channel, minimizing the number of signaling exchanges required in large conferences.

The author of this dissertation conceived of the idea and contributed to its description.

[Pub10] **Selective internal forwarding in conferences with distributed media servers**, Emil Ivov, Boris Grozev, United States patent application US20180375906A1, December 2018

This publication describes a multi-server video conferencing system, which applies the Last N scheme proposed in **[Pub5]** for the server-to-server connections in order to reduce internal bandwidth use.

The author of this dissertation contributed to the initial concept, and editing of the description.

[Pub11] **Dynamic adaptation to increased SFU load by disabling video streams**, Emil Ivov, Boris Grozev, Georgios Politis, United States patent US9712570B1, July 2017

This publication describes a system which uses the Last N scheme proposes in **[Pub5]** and simulcast in order to adapt to system load and avoid overloading a server.

The author of this dissertation contributed to the initial concept, and editing of the description.

[Pub12] **Multiplexing sessions in telecommunications equipment using interactive connectivity establishment**, Emil Ivov, Boris Grozev, Georgios Politis, United States patent US 9674140B1, June 2017

This publication describes a system which allows Interactive Connectivity Establishment (ICE) sessions to be multiplexed on a single IP address and port pair, by encoding a server identifier in the "username fragment" field of ICE packets.

The author of this dissertation conceived of the idea, created a proof-of-concept implementation, and contributed to its description.

Appendix B

Patents

In this appendix we present several US patents that came about from the author's work on this thesis, and which were pursued during his employment at Atlassian Plc.

B.1 Dynamic Adaptation to Increased SFU Load by Disabling Video Streams

This invention[Pub11] allows a Selective Forwarding Unit to dynamically adapt to overuse of the system resources (CPU, network bandwidth, or other), with the goal of improving the user experience, by adjusting the Last-N value, or the set of enabled simulcast or SVC encodings for a specific conference, or a set of conferences.

The SFU detects when it is overusing its available resources (e.g. it is using too much bandwidth). This indicates potential degraded experience for all users of the SFU (possibly hundreds) because packets are being lost on the network, resulting in video freezes or other adverse effects. It reacts by selecting a set of conference participants S and either lowering the LastN, or selecting a set of high-bitrate simulcast or SVC encodings currently being sent and temporary disabling them and forwarding instead a lower-quality and lower-bitrate version of the stream. This has two effects:

1. Lowering the resource usage. The goal is to lower it enough so that users do not have degraded experience.
2. For the participants in the set S, it forwards less data, and (in the case of LastN adjustments) fewer video streams. For example, if LastN was lowered from 5 to 4

for a participant, he or she will see 4 streams instead of 5. This is a negative effect for this participant, but it does not affect other participants in the conference.

The set S is chosen in such a way as to lower the resource usage enough (1), and minimize the negative effect from (2).

B.1.1 Algorithm

We use a monitoring function L which estimates the load $L(t)$ of an SFU at a given time t. This can be for example, the network utilization, CPU usage, or other metric or combination of metrics. The values of L are monitored, and events of overuse are detected when the value increases above a given threshold $T\_OVERUSE$. The threshold can be pre-configured (e.g. set to $100\,\mathrm{Mbps}$), or calculated dynamically. The value $O = L(t) - T\_OVERUSE$ is interpreted as the degree of overuse.

When an overuse event is detected, the SFU adapts by selecting a subset of the conference participants, and either

1. Decreasing the LastN value for each of them by a specific number
2. Switching some of the high-bitrate encodings sent to them to lower-bitrate encodings

The behavior then depends on whether it is using the LastN or simulcast/SVC variant.

LastN

In the case of the SFU selects a set S of pairs (P, i) where P is a participant and i is an integer (the semantics being that the LastN value for participant P is lowered by i). The set S is selected with an optimization procedure, which aims to minimize the negative effect on users while sufficiently lowering the system load.

Given a participant P and a number i, we use the following two concepts: $E_L(P, i)$ and $E_U(P, i)$. The first one, $E_L(P, i)$, describes the expected effect on the system load if the LastN value for P is reduced by i. And the second one, $E_U(P, i)$, describes the expected effect on the participant P if their LastN values is reduced by i. The exact definitions of E_L and E_U are not specified. An example is given below. We extend E_U

Algorithm 4 Algorithm for lowering LastN and recovery.

```
1: procedure MONITOR
2:     while true do
3:         x ← L(now)
4:         if x > T_OVERUSE then
5:             S ← choose_set_to_decrease(T_OVERUSE − x)
6:             decrease(S)
7:         else if X < T_UNDERUSE then
8:             S ← choose_set_to_restore(x − T_UNDERUSE)
9:             restore(S)
10:        wait(TIME_WAIT)
```

(and E_L) to sets of participants by taking the sum: $E_U(S) = sum_{P,i \in S} E_U(P, i)$ The set S is chosen as a subset of the set of all pairs (P, i) with i lower than the LastN value for P, so that the following hold:

1. The desired decrease in system load will be achieved, that is: $E_L(S) >= O$
2. S contains pairs with distinct participants. This is just a sanity check.
3. The negative effect on the users is minimized: the choice minimizes $E_U(S)$

The participants for which LastN was decreased are stored as $S_{decreased}$ (which again contains pairs (P, i)).

Recovery happens when the SFU detects underuse, indicated by $L(t) < T\_UNDERUSE$. The SFU reacts by restoring the LastN value for some of the participants for which it was reduced. It chooses the set for restoring in a similar way to above. Specifically, it chooses S as s subset of $S_{decreased}$, so that:

1. The expected increase in load is tolerable: $E_L(S) < T\_UNDERUSE - L(t)$
2. The choice maximizes the effect on users: $E_U(S)$

Simulcast or SVC

In the simulcast or Scalable Video Coding case the SFU selects a set of streams which it is currently forwarding in high-bitrate, and switches to a lower-bitrate version of the same stream. If the SFU is using simulcast, this involves stopping forwarding of the

high-bitrate simulcast stream and starting to forward the low-bitrate stream. If the SFU is using Scalable Video Coding, this involves dropping the additional layers. The set of streams is selected with an optimization procedure, aiming to achieve the desired effect on load while minimizing the negative effect on users.

Given a stream S we use the concept of expected load decrease, $E(S)$, which is the expected effect on the load of the SFU if it switches to streaming a lower bitrate version of S. The set of streams to disable is chosen as follows: first all streams are ordered according to E in descending order. Then the SFU selects the first k streams whose total effect (sum of $E(S)$) is at least O. This minimizes the number of affected streams, and thus the effect on users.

B.1.2 Example

Let the monitoring function $L(t)$ be network utilization measured as the average egress bitrate on the server for the last 5 seconds. Let us use pre-configured values for the thresholds:

$$T_OVERUSE = 200MbpsT_UNDERUSE = 180Mbps$$

Let us assume a constant bitrate $B = 2Mbps$ for each video stream (this is a naive assumption, in a real-world implementation we would calculate the current bitrate of streams).

Let us define $E_L(P, i) = B * i$. This is reasonable, because we will send i less streams than before.

Let us define E_U as follows: $E_U(P, i) = i/LastN(P)$. This definition takes into account the fraction of streams that the user will stop receiving, and so captures the idea that a decrease from 10 to 9 has less of a negative effect than a decrease from 2 to 1.

Let the SFU currently host the following three conferences C_1 with 20 participants and $LastN = 3$, C_2 with 10 participants and $LastN = 5$, and C_3 with 3 participants and $LastN = 2$.

Suppose the SFU measures $L(t) = 210Mbps$. We then have $O = 210Mbps - 200Mbps = 10Mbps$. This constitutes overuse. According to the procedure described above, the SFU will search for a set S which will result in an expected decrease in load of $10Mbps$ or more, while minimizing $E_U(S)$.

For the participants P with $LastN = 2$ we have: $E_U(P, i) = i/2$.

For the participants P with $LastN = 3$ we have: $E_U(P, i) = i/3$.

For the participants P with $LastN = 5$ we have: $E_U(P, i) = i/5$.

The general problem of choosing S can be modelled as a knapsack problem. With the example definitions it can be simplified. In this case the solution will have 5 participants from conference C_2, with $i = 1$ for each. That is, the SFU will choose 5 participants from C_2, and decrease their LastN from 3 to 2. This has an expected effect on load of $E_L(S) = 5 * B = 10Mbps$, which is larger than the overuse threshold O.

B.2 Multiplexing Sessions in Telecommunications Equipment Using Interactive Connectivity Establishment

This invention[Pub12] allows routers performing Network Address Translation (NAT) to create and maintain NAT mappings for Interactive Connectivity Establishment (ICE) sessions without taking active part in the ICE exchange and without the need for communication between the NAT routers and the ICE endpoints. Allows a single public IP address and transport port number to be used for ICE sessions terminating in multiple endpoints.

In the case where each ICE endpoint has a publicly accessible IP address, the client can just connect to it using this address. But when one IP address and port number pair is shared between multiple ICE endpoints, the entity doing the multiplexing (the NAT router in this case) needs to somehow select the ICE endpoint to be used. This is a problem when a specific ICE endpoint must be used for a given connection (e.g. so that the client is connected to the correct conference). Other solutions include terminating ICE at the entry point that holds the IP address (i.e. at the element we call a "NAT router"

The system includes a set of ICE endpoints (e.g. media servers such as Selective Forwarding Units) without public IP addresses, and a NAT router (or a set of NAT routers, possibly sharing the same public address, and sharing their NAT table), which provide NAT services for the set of ICE endpoints. One or more signaling servers are used to setup an ICE session between clients on the internet, and a specific ICE endpoint. One example use case is connecting a client to a multimedia conference hosted on one of the ICE endpoints.

The ICE endpoints (which may be running either full ICE or ICE Lite) generate their part of the username (the local ice-ufrag, or ICE local username fragment) in a way that encodes an identifier for the ICE endpoint in it. Any scheme that encodes

an identifier for the ICE endpoint in a way that the NAT router can decode can be used. One example of an encoding method is to encode the IP address of the ICE endpoint as a string using dotted-decimal notation, and append to it a randomly generated 8 character string. The NAT router uses the "remote username fragment" from the USERNAME attribute of STUN packets coming from clients, in order to decide which ICE endpoint is the destination. It extracts the encoded identifier of the ICE endpoint and uses this endpoint for subsequent packets from the given (source IP address, source port number) pair (i.e. it creates an entry in its NAT table for the given client address). The NAT routers use the following procedures to create and maintain entries in their NAT table:

1. When a datagram is received on the public or private address, which matches one of the existing entries in the NAT table, it is translated and forwarded accordingly (i.e., it acts like a regular NAT router).

2. When a datagram is received on the public address, which does not match any existing entry in the NAT table:

   (a) The NAT router attempts to interpret the datagram as a STUN packet and looks for a USERNAME attribute.

      i. If such an attribute is found, the "remote u-frag" is extracted[ICE] and the identifier for the ICE endpoint is extracted from the "remote u-frag" A destination address is constructed based on ICE endpoint identifier. If the example encoding scheme given above is used, the NAT endpoint will remove the last 8 characters from the remote username fragment, and parse the resulting string as an IP address. This will be the destination IP address.

   (b) If a destination address is not found in "a" above, the packet is dropped without further processing.

   (c) If a destination address is found, the NAT router chooses (for example depending on configuration) a protocol to use for the connection to the ICE endpoint (UDP or TCP), and adds an entry to its NAT table, mapping the source IP address and port number of the packet to the IP address of the NAT router and the destination port number of the packet (or a pre-configured, or otherwise obtained (e.g. also encoded in the remote username fragment) port number).

      i. If TCP was chosen as the protocol, it initiates a TCP connection towards the destination address.

   (d) The datagram is forwarded (after possibly being held until the TCP connection is open) to the destination address.

3. NAT table entries are timed out if no packets are received for a certain configured amount of time. This is also normal NAT router operation.

4. NAT table entries are removed if a STUN packet with response code 403 (Forbidden) is detected (indicating denied consent).

The scheme can be used with UDP, with TCP using framing [43], or other protocols. When used with TCP:

1. The NAT router terminates the TCP connection, and interprets the payload as framed in RFC4571 format, with the exception of a possible initial SSL handshake, which the NAT router also terminates.

2. RFC4571-framed datagrams are handled the same way as datagrams received over UDP (see above).

STUN packets have a well-known format and are recognized (differentiated from RTP/RTCP/DTLS packets) by the magic cookie [65]

More than one NAT router can be used, with a shared NAT table. The determination of the ICE endpoint is deterministic, based on the STUN `USERNAME` attribute, so in the absence of a mapping all NAT routers will select the same ICE endpoint.

B.2.1 Example

As an example we use a video conferencing service hosted on a set of SFUs, as illustrated on figure B.1. The SFUs have no public IP address and they are fronted by a router.

When a client joins a conference, it firstcontacts a signaling server which provides it with remote ICE candidates to be used to connect to a specific SFU. In this case this is the address of the router, shared between the three SFUs. The singling info also includes a username fragment, in which the SFU used for the session encodes its private IP address (10.0.0.3 in this case).

The client then attempts to establish an ICE session with 1.2.3.4. The router examines the first packet of the session, a STUN Binding Request, reads the IP address encoded in the `USERNAME FRAGMENT` field and maps this session to the specific SFU (SFU 3 in this case). Subsequent packets in this session are forwarded to SFU 3.

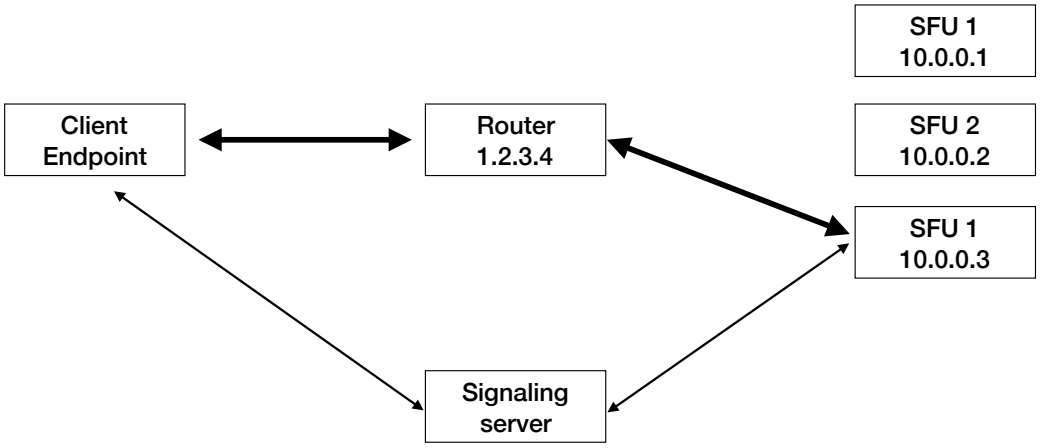


Figure B.1: *Using ICE multiplexing with SFUs in a video conferencing service. The set of SFUs have only private IP addresses, which they encode in their ICE username fragments.*

B.3 Rapid Optimization of Media Stream Bitrate

This invention[Pub8] allows endpoints connecting to a real-time media distribution system, such as a Selective Forwarding Unit, to optimally initialize their stream bitrate based on performance statistics from previous connections originating from the same network.

In this section we refer to the SFU or other server component used to host the conference as the "Media Server" (MS).

B.3.1 Problem Description

In the context of a video conference, middleboxes often analyze incoming traffic sent by endpoints and determine that they are either overusing or underusing available bandwidth. That process is often referred to as "bandwidth estimation" and it produces a rough estimate for the capacity of the link between the endpoint and the middlebox. Once it has calculated the estimate, the middlebox often indicates it to the endpoint. At this stage, after taking into account the bitrate it currently uses, the amount of loss it is sustaining and the indication from the middlebox, the endpoint adjusts its sending bitrate, in some cases ramping it up to higher levels indicated by the middle box.

An important constraint for the above process comes from the nature of the underuse estimations produced by middleboxes. In short, underuse is detected by the lack of anomalies in incoming traffic. To avoid causing a network congestion, algorithms for congestion control start with a relatively low value, and increase it after underuse is detected (this is often referred to as "ramp-up" The amount of time necessary for the ramp-up to the stable bitrate that can be used for the session depends on different factors such as the round-trip time (RTT) between the endpoints and the receiver's (middlebox's) estimation. The ramp-up speed is usually limited to a fraction of the current bitrate per second (Google Congestion Control uses 8% per second as the maximum rate of increase[16]). As a result, ramp-up speed is sensitive to the initial value, and using a lower-than-necessary initial value manifests as prolonged periods of poor user experience in the early stages of a video conference.

B.3.2 Proposed Solution

In order to shorten the duration of the initial capacity discovery process, a service will maintain a database which maps an IP address or network to a history of previously achieved stable bitrates. The definition of a "network" is configurable by the middlebox administrators and can vary from a single IP address to a $/X$ group of IP addresses. Every time an endpoint connects to a middlebox using this invention, the database is consulted and an initial value for the bitrate estimation is chosen based on the endpoint's current network. After the bandwidth estimation has stabilized, the database will be updated with the new stable bitrate, which can be used for future sessions. Figure B.2 illustrates this.

B.3.3 Database

The database stores the following information: timestamp, endpoint/client IP address, MS IP address, the stable bitrate. Example record:

$$(timestamp, client\_ip, server\_ip, stable\_bitrate)$$

$$(147274358684062, 192.168.0.1, 10.10.0.1, 1Mbps)$$

The specific format of the data, the software used, and the type of the database are not substantial.

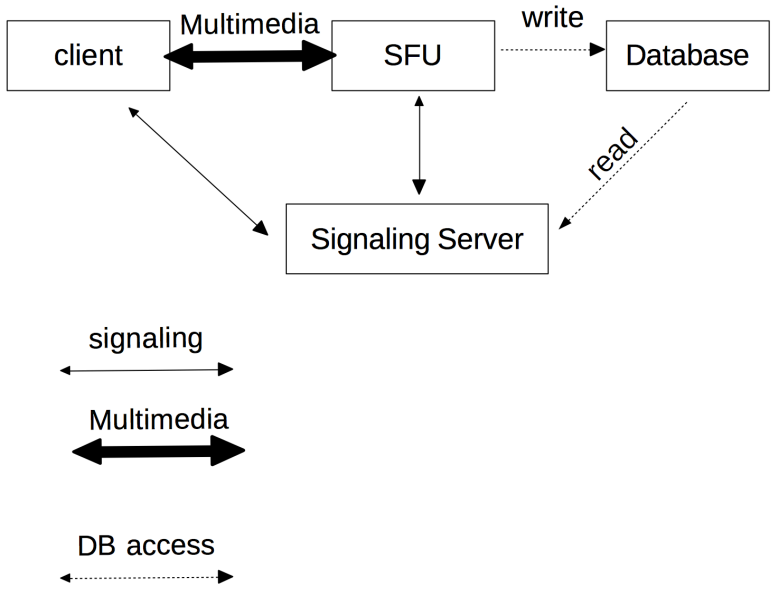


Figure B.2: *The components and types of connections used for optimization of the initial bitrate*

B.3.4 Signaling Server Procedures

When a new client joins a call, it first contacts the signaling server. The signaling server selects the MS to be used (if the client joins an existing conference, the MS is already determined. If it is a new conference, an MS is allocated). At this point it knows the IP addresses of both the client and the MS. It queries the database for records matching the IP addresses (see the paragraph on matching below). If there are results, it selects the most recent one (based on the timestamp field). If it is "recent enough" (the timestamp is at most T seconds in the past, where T is a parameter to be chosen), the server extracts the value from the "stable bitrate" field, and passes that value to the client.

If there are no appropriate results, the signaling server proceeds with the conference normally (without providing a "start bitrate" value to the client). It is worth noting that, in practice, the "Signaling Server" and the "Media Server" may be co-located, part of the same process or even indistinguishable from an implementation perspective.

B.3.5 Client Procedures

During session initiation or in the very early stages of the session itself, the client learns from the signaling server the target value that it feeds to it bitrate estimation algorithm. If the signaling server does not indicate a specific value, the client uses a default (which is standard client behavior).

B.3.6 Media Server Procedures

During a session with a client, the MS keeps estimating the available bandwidth (obtaining a bwe value (bandwidth estimation)) from the client to the MS (this is standard MS behavior). Periodically (for example every X minutes), it checks for a stable rate. If it finds a rate considered stable, it updates the database with the current timestamp, the client IP address, its own IP address (used for the connection to the client), and the stable rate. A rate R is considered stable if:

1. $bwe >= R$ for the last C_1 seconds.
2. The actual receiving rate is $>= R$ for the last C_2 seconds.
3. bwe has been non-decreasing for the last C_3 seconds.

B.3.7 Matching Entries in the Database

This describes how to decide whether a pair of IP addresses IP_c (client) and IP_s (server) matches an entry in the database with values E_c and E_s for the client and server addresses.

The simplest way is to match exactly, that is to match the pair if and only if $IP_c = E_c$ and $IP_s = E_s$. More complex options may also be used, such as matching only a network prefix with a given length for either the client or server address (or both).

Example: exact matching for the client IP address, and matching with a 24-bit prefix for the server address. The pairs match if and only if: $IP_c = E_c$ and the first 24 bits of IP_s and E_s are the same (i.e. IP_s and E_s are in the same $/24$ network). Any of these options can be used.

B.4 Selective Internal Forwarding in Conferences with Distributed Media Servers

This invention[Pub10] reduces the amount of data transferred between SFUs in a conference with cascades SFUs.

We are interested in a conferencing system based on a Selective Forwarding Unit. In order to minimize the traffic towards a client the media server does not forward every packet of every video stream, but selects only a subset of the packets to forward to clients. Traditionally a conference uses a single SFU, with all clients connected to it. This limits the number of participants in a single conference (to somewhere between a few dozen and a few hundred at most). One way to scale this further is to distribute the SFU among multiple servers. We will refer to a conference which uses multiple SFUs in this way as a "distributed conference".

B.4.1 Description

Each client in the conference is connected to only one SFU. The SFUs (there may be one, two, or more) are connected together, and they forward the clients' media streams between themselves. They can be directly connected to each other in a full-mesh topology, or in another way (e.g. in a multi-level hierarchy). This invention applies the ideas of selective forwarding to the streams being sent between the SFUs in a distributed conference. Specifically, each SFU in a distributed conference only forwards a subset of the streams coming from its clients to other SFUs. There can be different ways in which to decide which streams/packets should be forwarded. The examples below illustrate this.

Example 1: Explicit Configuration Based on Signaling

The set of streams to be forwarded is explicitly signaled. One way to do this is to use a centralized conference controller (a "conference focus" which makes the decision of which streams should be forwarded, and then signals this information to each SFU. This configuration changes dynamically based on activity in the conference, so the controller must constantly send updates to the SFUs.

Example 2: Forwarding Based on Subscription

SFUs signal to each other the streams that they want to receive.

Example 3: Forwarding Based on LastN

Each SFU performs dominant speaker detection independently, based on all the audio streams which it receives, in order to decide which video streams to forward to the clients connected to it.

While performing dominant speaker detection, a given SFU S_1 constructs an ordered list of all endpoints in the conference (some of them on S_1 and some of them on on other SFUs):

$$L_{global} =< e_1, e_2, e_3, ..., e_k >$$

Here e_1 is the current dominant speaker, e_2 is the previous, etc. Let us define the list L_{local} as the list of all endpoints connected to S_1 ordered according to L_{global}.

We propose to extend the idea of LastN and apply it to streams being sent between the SFUs. Specifically, SFU S_1 will forward to all other SFUs in the conference:

1. The audio for the first N endpoints from the list L_{local}.
2. The video for the local endpoints which are also part of the list $L_{global} =< e_1, e_2, e_3, ..., e_N >$. That is, the first N endpoints in the list L_{global}.

The aim of this specific selection is to:

1. Allow each SFU to perform dominant speaker detection independently (in order to not have to use a centralized node, or implement a complex distributed solution).
2. Send all audio streams which may potentially be needed for LastN by another SFU.
3. Send the minimum amount of video streams which are expected to be needed by other SFUs.

Since audio streams take much less resources, and because in large conferences a significant percentage of the participants are likely to be muted, another viable solution is to always send all non-silence audio streams between SFUs. This has the advantage of simplicity.

Example 4: Forwarding Based on LastN with the Option of Explicit Subscriptions

SFUs use the scheme described in **Example 3**, but in addition they also allow explicit subscriptions. For example if a client has specifically selected to see the video of someone in the conference, who is not in the LastN set, then this clients SFU will explicitly subscribe to the desired stream.

Other ways to make the selections are possible, some might be based on one on more of the examples above.

B.5 On Demand In-band Signaling for Conferences

This invention[Pub9] allows participants in a video conference to receive meta-data about the audio and video streams that they are receiving on-demand (whenever a new stream is available) and in-band from the media server (as opposed to through the signaling path). This allows for video conferences to scale to a larger number of participants.

In a typical online video conference scenario a participant is connected to two servers. One is a signaling server, which manages authentication, authorization, session establishment, identification of the other endpoints in the conference, and exchange of metadata about the audio and video streams (such as e.g. SSRC identifiers, Media Stream identifiers, etc). The other one is a media server (such as an Multipoint Control Unit (MCU) or a Selective Forwarding Unit (SFU)), which handles the audio and video streams.

One reason for this separation is that the processing of multimedia is much more expensive than the processing of signaling, and therefore a system scaled optimally needs a larger number of media servers than signaling servers. That is, it is often the case that one signaling server works with a set of multiple media servers.

When clients join a conference and receive a media stream, they need some meta data in order to know how to display the stream (e.g. which participant in the conference the stream belongs to). This information is usually exchanged through signaling. That is, the signaling server provides to clients in the conference a mapping of all other clients to their stream identifiers (SSRCs).

For large conferences (e.g. with hundreds of participants) this becomes impractical, because the list is large and current WebRTC implementations often show performance issues processing them. A large set of participants also implies frequent changes and hence significantly increased signaling traffic, which may be a problem of its own. Fur-

ther, in a large conference, a given client will only receive media from a subset of the other clients (e.g. it will receive audio only for the people who unmute, and video for the last 5 speakers). And the set of streams which are sent to a given client changes dynamically, as decided by the media server.

In our solution, the signaling server leaves off the part of signaling which describes the mapping from client to media stream identifier. Instead, this information is provided to clients by the media server, in an in-band channel. Further, the server does not provide the full list of mappings, but adds new ones dynamically as it starts to send new streams to a particular client.

B.5.1 Example

In a multi-media conference based on WebRTC, the signaling contains two types of information. One is information about the media types (i.e. codecs) in use, and about session establishment (i.e. ICE credentials and candidates, and DTLS certificate fingerprints). The second is a set of mappings from a Synchronization Source (SSRC) to a structure containing information about the stream, such as: Media Stream ID (MSID, used to identify flows which originate from the same source and share the same clock, and which should therefore be played out synchronized (e.g. audio and video coming from an endpoint)), an identifier of the conference endpoint (e.g. “this is John”

With our solution the signaling server will exchange the first type of information, which will allow the client to establish a connection to the media server. Once the connection is established, the client and the media server open a WebRTC DataChannel, over which they can communicate structured data. When the media server begins to send a new audio or video stream to a client, it also sends the SSRC to MSID mapping over the DataChannel, which the client adds to its list of mappings.