Live Streaming with Gossip

THÈSE N^o 4777 (2010)

PRÉSENTÉE LE 30 JUILLET 2010

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE PROGRAMMATION DISTRIBUÉE

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Maxime Monod

acceptée sur proposition du jury:

Prof. R. Hersch, président du jury
Prof. R. Guerraoui, directeur de thèse
Dr A. Argyraki, rapporteur
Prof. B. Garbinato, rapporteur
Dr A.-M. Kermarrec, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2010

Abstract

Peer-to-peer (P2P) architectures have emerged as a popular paradigm to support the dynamic and scalable nature of distributed systems. This is particularly relevant today, given the tremendous increase in the intensity of information exchanged over the Internet. A P2P system is typically composed of participants that are willing to contribute resources, such as memory or bandwidth, in the execution of a collaborative task providing a benefit to all participants. File sharing is probably the most widely used collaborative task, where each participant wants to receive an individual copy of some file. Users collaborate by sending fragments of the file they have already downloaded to other participants.

Sharing files containing multimedia content, files that typically reach the hundreds of megabytes to gigabytes, introduces a number of challenges. Given typical bandwidths of participants of hundreds of kilobits per second to a couple of megabits per second, it is unacceptable to wait until completion of the download before actually being able to use the file as the download represents a non negligible time. From the point of view of the participant, getting the (entire) file as fast as possible is typically not good enough. As one example, Video on Demand (VoD) is a scenario where a participant would like to start previewing the multimedia content (the stream), offered by a source, even though only a fraction of it has been received, and then continue the viewing while the rest of the content is being received.

Following the same line of reasoning, new applications have emerged that rely on *live streaming*: the source does not own a file that it wants to share with others, but shares content as soon as it is produced. In other words, the content to distribute is *live*, not pre-recorded and stored. Typical examples include the broadcasting of live sports events, conferences or interviews.

The *gossip paradigm* is a type of data dissemination that relies on random communication between participants in a P2P system, sharing similarities with the epidemic dissemination of diseases. An epidemic starts to spread when the source randomly chooses a set of communication partners, of size *fanout*, and infects them, i.e., it shares a rumor with them. This set of participants, in turn, randomly picks *fanout* communication partners each and infects them, i.e., share with them the same rumor. This paradigm has many advantages including fast propagation of rumors, a probabilistic guarantee that each rumor reaches all participants, high resilience to churn (i.e., participants that join and leave) and high scalability. Gossip therefore constitutes a candidate of choice for live streaming in large-scale systems.

These advantages, however, come at a price. While disseminating data, gossip creates many duplicates of the same rumor and participants usually receive multiple copies of the same rumor. While this is obviously a feature when it comes to guaranteeing good dissemination of the rumor when churn is high, it is a clear disadvantage when spreading large amounts of multimedia data (i.e., ordered and time-critical) to participants with limited resources, namely upload bandwidth in the case of high-bandwidth content dissemination.

This thesis therefore investigates *if* and *how* the gossip paradigm can be used as a highly efficient communication system for live streaming under the following specific scenarios: *(i)* where participants can only contribute limited resources, *(ii)* when these limited resources are heterogeneously distributed among nodes, and *(iii)* where only a fraction of participants are contributing their fair share of work while others are freeriding.

To meet these challenges, this thesis proposes *(i)* gossip++: a gossip-based protocol especially tailored for live streaming that separates the dissemination of metadata, i.e., the location of the data, and the dissemination of the data itself. By first spreading the location of the content to interested participants, the protocol avoids wasted bandwidth in sending and receiving duplicates of the payload, *(ii)* HEAP: a *fanout* adaptation mechanism that enables gossip to adapt participants' contribution with respect to their resources while still preserving its reliability, and *(iii)* LiFT: a protocol to secure high-bandwidth gossip-based dissemination protocols against freeriders.

Keywords Live streaming, gossip, epidemic dissemination, large-scale distributed systems, peer-to-peer, P2P, overlay, freeriding.

Résumé

Les architectures pair-à-pair (P2P) représentent un paradigme particulièrement populaire pour soutenir la nature dynamique, évolutive ainsi que résister à la charge des systèmes distribués à très large échelle, charge qui a augmenté ces dernières années de manière considérable avec l'intensification des échanges par Internet ainsi que la taille des données échangées. Un système P2P est généralement composé de participants disposés à fournir leurs ressources, typiquement de la mémoire ou de la bande passante, dans l'exécution d'une tâche collaborative offrant un bénéfice à ses participants. Le partage de fichiers est probablement la tâche collaborative la plus utilisée actuellement, où chaque participant cherche à recevoir une copie du fichier partagé et collabore ainsi en échangeant avec les autres participants des fragments du fichier déjà reçus.

L'échange de fichiers contenant des données multimédias, de taille allant de centaines de mégaoctets à des gigaoctets, introduit de nouveaux défis à relever. Compte tenu de la bande passante des participants, plusieurs centaines de kilobits par seconde à quelques mégabits par seconde, il est inacceptable d'attendre la réception complète du fichier avant d'être en mesure de l'utiliser, étant donné le temps que le téléchargement représente. Du point de vue du participant, obtenir la totalité du fichier le plus rapidement possible n'est donc généralement pas suffisant. La vidéo à la demande (VoD de l'anglais *Video on Demand*) représente un scénario typique où un participant souhaite commencer la visualisation du contenu multimédia (le flux, *stream*), offert par une source, alors que seulement une fraction de celui-ci a été reçue, puis poursuivre le visionnement pendant que le reste du contenu est téléchargé.

Suivant cette idée, une nouvelle tâche collaborative a émergé en tant que diffusion de flux en direct (*live streaming*) : la source ne possède pas un fichier stocké qu'elle veut partager avec les autres participants, mais le contenu est envoyé pendant qu'il est produit. En d'autres termes, le contenu à distribuer est *en direct* (*live*) et non pas pré-enregistré ni stocké. Des applications typiques sont la diffusion en direct d'événements sportifs, de conférences ou d'interviews.

Le paradigme du *gossip*, qui peut être traduit en commérage en français, est un type de diffusion de données qui repose sur la communication au hasard entre les participants. Ce type de diffusion est similaire à la diffusion épidémique d'une maladie. Une rumeur ou une épidémie commence à se propager quand la source choisit au hasard un ensemble de participants, de taille *fanout*, et les infecte, en partageant la rumeur avec chacun d'eux. Chaque participant de cet ensemble, à son tour, choisit au hasard le même nombre *fanout* de participants et partage avec eux cette même rumeur. Ce paradigme a de nombreux avantages dont la propagation rapide des rumeurs, la garantie probabiliste que chaque rumeur atteint chaque participant, la résistance à la dynamique du système (les participants qui le joignent et le quittent) ainsi que l'extensibilité à de larges audiences. Ce paradigme est donc particulièrement indiqué pour la diffusion de flux en direct dans des systèmes à large échelle.

Ces avantages ont toutefois un prix. Pendant la diffusion épidémique, de nombreux doublons de la même rumeur sont créés et les participants reçoivent habituellement de multiples copies de la même rumeur. Bien que ceci représente de toute évidence une caractéristique demandée lorsqu'il s'agit de garantir une bonne diffusion de rumeurs en cas d'arrivées et de départs soudains des participants, c'est un net désavantage en cas de propagation de grandes quantités de données multimédias sachant que les participants ont des ressources limitées, à savoir leur bande passante dans le cas précis.

Cette thèse étudie donc la possibilité et la mise en oeuvre de l'utilisation du paradigme du gossip en vue d'une diffusion efficace d'un flux en direct dans les scénarios suivants : *(i)* lorsque les participants ont des ressources limitées, *(ii)* lorsque ces ressources sont distribuées aux participants de manière hétérogène et *(iii)* lorsque seule une fraction de ces participants contribuent de bonne foi avec une partie de leurs ressources, tandis que d'autres profitent du système, en tant que resquilleurs ou fraudeurs (*freeriders*).

Afin de relever ces défis, cette thèse propose *(i)* gossip++ : un protocole particulièrement adapté au besoin de la diffusion de flux en direct, basé sur le paradigme du gossip, dans lequel la rumeur propagée est composée de l'emplacement du contenu, contenu qui est par la suite téléchargé par les participants intéressés, évitant ainsi de recevoir des doublons du contenu lui-même et donc de gaspiller de la bande passante, *(ii)* HEAP : un mécanisme d'adaptation du *fanout*, permettant au gossip d'adapter la contribution des participants (l'utilisation de leurs ressources) en fonction de la quantité de leurs ressources à disposition tout en préservant la fiabilité du gossip et *(iii)* LiFT : un protocole pour garantir l'efficacité de gossip dans un contexte d'échanges de grandes quantités de données en présence de fraudeurs.

Mots-clés Diffusion de flux en direct, *live streaming*, diffusion épidémique, commérage, *gossip*, systèmes distribués à large échelle, pair-à-pair, P2P, topologie de recouvrement, resquilleurs, fraudeurs, *freeriding*.

This thesis is dedicated to Jean and Daniel, my late grandfathers,
Ulysse, Taotim and Kilian, my nephews and godsons,
and Olivia, my lovely angel.

Acknowledgments

First, I would like to thank my advisor, Prof. Rachid Guerraoui, for welcoming me in his lab and having guided my research while at the same time, giving me a maximum of freedom in my working method, in the directions followed or choices made. I also thank him for the countless jogging trips, allowing us to resolve research issues, discuss the progress of various projects and talk about life in general. These runnings have also allowed us to know ourselves better while preparing physically for various races, including the Marathon.

I thank Prof. Roger David Hersch for being president of the jury and Dr. Katerina Argyraki, Dr. Anne-Marie Kermarrec and Prof. Benoît Garbinato for accepting to be members of the jury.

Thanks to Benoît Garbinato and Jarle Hulaas who had the difficult task of supervising the work that Jesper and I had to make in the european project PALCOM, during Rachid's sabbatical leave to MIT. I very much appreciated their follow-up and their many advices that have been very beneficial throughout this thesis. I am grateful for Jamila Sam's professionnalism and all her help for the smooth running of the course entitled "Introduction to Object-oriented Programming".

I thank every coauthor with whom I had the chance to work and in particular Davide Frey who has shown me that apart from the *Spaghetti Westerns*, Italians were also very good at *Spaghetti Programming*; Kévin Huguenin for his many invitations to his place and for making me discover duck confit and "pommes de terre à la sarladaise"; Anne-Marie Kermarrec for having welcomed me many times in Rennes and for her unfailing scientific and moral support; Boris Koldehofe in memory of *Weißbiere* drunk together in Stuttgart and Vivien Quéma for indoor football games and the many programming nights, both at EPFL or in Grenoble.

I thank all former and current lab members: Aleksandar, Bastian, Boris, Dan, Fabien, Giuliano, Jesper, João, Kristine, Marko, Maysam, Michal, Mihai, Nikola, Oana, Partha, Petr, Ron, Sidath, Seb, Seth, Vincent, for the many discussions we had and also for activities we performed together, such as (day and night) barbecues, fitness, jogging, soccer games, sailing, skiing, tennis, adventure park or a flight over the Alps. I would like to especially thank Kristine for having always been present and receptive towards many requests I had, with a very open-mind, always very patient, and for making this lab lively, by organizing various events for birthdays or births, for instance. I thank her specially for proofreading this thesis, an example among many others of her kindness and dedication. A special thought for her grandson Timothée, who, with his smiles and laughs, constantly reminds us that we are finally only big children!

I thank the whole LAMP team led by Prof. Martin Odersky, and especially Danielle, Michel, Rachele, Stéphane and Vincent, for their encouragement and good time spent together during our trips to Sardinia or Aarhus, precisely with Martin and Nikolay.

Thanks to Prof. Alfred Strohmeier and Prof. Jörg Kienzle for giving me the desire to pursue research and do a PhD at EPFL.

Of course, I will never forget the unconditional support of my family: Daniel (1922–2008) and Catherine; Jean (1924–2006) and Jacqueline; Francine; Jean-Da and Elisabeth; Guy and Lise; Cédric, Céline and Ulysse (2008–); Leslie, Christophe and Taotim (2010–), and Françoise. *Bacis* to my family in Ticino: Janine, Vania and Alan, and best greetings to my Othenin-Girard cousins: Bernard, Irina, Alex, and Nico.

I also warmly thank the de Weck family for very good times together, by the fireside in their chalet in Lac Noir, on the shores of Lake Neuchâtel or during our trips to Bombay, Boston and New York.

I thank Olivier Paroz and Frédéric Bourqui for sharing with me their passion for computer science, Marco Bonetti for reinforcing me in the idea of joining EPFL, Pierre-Alain Dumont for giving me his confidence in the Bluepage/Actio adventures and my first job in computer science, Lt Col Olivier Henchoz for giving me the chance to lead a company and Cécilia Bigler & Sylviane Dal Mas for organizing the anniversary of the School of Computer and Communication Sciences together.

Thanks to all my friends who have contributed directly or indirectly to this thesis and who were still satisfied with my reduced presence in our joint activities during these years: Alain & Anne, Alban, Alex “JDK”, André “le Pfist’r”, Axel du conseil, Ban, Barben & Janice, Basile, Bastian & Laura, Bérard, Borlat, Bouvier, Chens & Jenny and their future little boy, Christian and his “Gagg”, Christophe “Dynafit”, Claudio, Darko, David, de Chambrier and family, Denti, Didier, Didier “gren” and family, Diego, Duboss’, FN & Mag’, Franck, François “The Dude”, Frey, Genton, Goss’ & Julie, Grégoire “Gel énergétique”, Guillaume & Maryll and their future child, Guillaume & Shahida & Juliette & Clémentine, Jean-Michel, Jesper & Tania, Jo & Katy & Emily, José, Juliette, Ky-Anh, Kaufmann, Landert and family, Lato “de combat 90”, Lucas “Les Ouais”, Luyet, Marc & Caro, Martial & Carolyne, Mathieu “The Mathiew” & Marie-Julie, Mélanie, Nicolas and Fred “aux dents blanches”, Paulo “le cabri”, Pernet, Perret, PH, Phil & Catherine & Noémie & Rachel & Kilian, Pierre “le dauphin" and family, R1 & Delphine, Renaud "frère d'arme", Ron & Laura, Rumley, Sacha, Sami & Carrie, Schouwey, Seb and family, Stocker & Lili and family, Stefan, Stefano and family, Svend, Théo and family, Vincent & Aline, Yann and family, and Yves "au taquet". Bachelor's parties, rallyes, via ferratas, climbing, rafting, barbecues, Patrouille des Glaciers, Raids, skydiving, paragliding (among other things): what memories shared with you!

Last but not least, I thank my dear and loving Olivia for sharing with me unforgettable moments, for all her love, and support in difficult times. With her, not only does everything become possible, but also easy! I look forward to many moments and experiences to come; I promise to bend my knees when skiing and try to serve better when we play tennis.

Remerciements

Tout d'abord, j'aimerais remercier mon directeur de thèse, le Prof. Rachid Guerraoui, pour m'avoir accueilli dans son laboratoire et avoir guidé mes recherches tout en me laissant un maximum de liberté dans ma méthode de travail, que ce soit dans les directions suivies ou choix effectués. J'aimerais aussi le remercier pour les innombrables sorties de jogging que nous avons faites, nous permettant de régler différents points de recherche, parler de l'avancement des différents projets ainsi que sur la vie en général. Ces courses nous ont aussi permis de mieux nous connaître et de passer des moments privilégiés tout en nous préparant physiquement pour diverses courses, notamment pour le Marathon.

Je remercie le Prof. Roger David Hersch de présider le jury, ainsi que la Dr Katerina Argyraki, la Dr Anne-Marie Kermarrec et le Prof. Benoît Garbinato pour avoir accepté d'être membres de ce jury.

Merci à Benoît Garbinato et Jarle Hulaas qui ont eu la lourde tâche de superviser le travail que Jesper et moi-même avons dû effectuer au sein du projet PALCOM pendant l'année sabbatique de Rachid au MIT. J'ai beaucoup apprécié leur suivi et leurs nombreux conseils m'ont été très bénéfiques tout au long de cette thèse. Je remercie Jamila Sam pour son professionnalisme et toute son aide pour le bon fonctionnement du cours "Introduction à la Programmation Objet".

Je remercie tous les coauteurs avec qui j'ai eu la chance de collaborer et en particulier Davide Frey qui m'a démontré que mis à part les *western spaghettis*, les italiens étaient aussi très forts pour le *code spaghetti* ; Kévin Huguenin pour ses nombreuses invitations à la maison et pour m'avoir fait découvrir ainsi le confit de canard et les pommes de terre à la sarladaise ; Anne-Marie Kermarrec pour m'avoir souvent accueilli à Rennes et avoir été un soutien scientifique et moral sans faille ; Boris Koldehofe en souvenir des *Weißbiere* bues à Stuttgart et Vivien Quéma pour les parties de foot *indoor* et les nombreuses nuits de programmation à l'EPFL ou à Grenoble.

Je remercie tous les anciens et actuels membres du laboratoire : Aleksandar, Bastian, Boris, Dan, Fabien, Giuliano, Jesper, João, Kristine, Marko, Maysam, Michal, Mihai, Nikola, Oana, Partha, Petr, Ron, Sidath, Seb, Seth, Vincent, pour les nombreuses discussions que l'on a eues et aussi pour les activités que l'on a pratiquées ensemble, comme par exemple les barbecues (de jour comme de nuit), le fitness, le jogging, les matchs de foot, de la voile, du ski, du tennis, le parc aventure ou encore un vol au-dessus des Alpes. Je tiens à remercier tout particulièrement Kristine pour avoir toujours été présente et réceptive à l'égard des maintes demandes que j'ai pu lui faire, avec un grand esprit d'ouverture, toujours positive et très patiente, et pour avoir fait vivre le laboratoire en organisant divers événements que ce soit pour les anniversaires ou les naissances, par exemple. Je la remercie spécialement pour la relecture de cette thèse, un exemple parmi tant d'autres de sa gentillesse et de son dévouement. Une pensée toute particulière pour son petit-fils Timothée qui, avec ses sourires et ses rires, nous rappelle constamment que nous ne sommes finalement que des grands enfants !

Je remercie toute l'équipe du LAMP dirigée par le Prof. Martin Odersky, et en particulier Danielle, Michel, Rachele, Stéphane et Vincent, pour leurs encouragements ainsi que les bons moments passés ensemble lors de nos voyages en Sardaigne ou à Aarhus avec Martin et Nikolay justement.

Merci au Prof. Alfred Strohmeier et au Prof. Jörg Kienzle pour m'avoir donné envie de poursuivre dans l'académique et de faire un doctorat à l'EPFL.

Bien évidemment, je n'oublierai jamais le support inconditionnel de ma famille : Daniel (1922–2008) et Catherine ; Jean (1924–2006) et Jacqueline ; Francine ; Jean-Da et Elisabeth ; Guy et Lise ; Cédric, Céline et Ulysse (2008–), Leslie, Christophe et Taotim (2010–) et Françoise. *Bacis* à ma famille du Tessin : Janine, Vania et Alan et meilleures salutations à mes cousins Othenin-Girard : Bernard, Irina, Alex et Nico.

J'aimerais aussi remercier très chaleureusement la famille de Weck pour les très bons moments passés ensemble, que ce soit au coin du feu dans les alpes fribourgeoises, au bord du lac de Neuchâtel ou encore lors de nos escapades à Bombay, Boston et New-York.

Je remercie Olivier Paroz et Frédéric Bourqui pour m'avoir transmis leur passion pour l'informatique ; Marco Bonetti pour m'avoir conforté dans l'idée d'aller à l'EPFL ; Pierre-Alain Dumont pour m'avoir donné toute sa confiance dans l'aventure Bluepage/Actio ainsi que mon premier emploi dans l'informatique ; le lt col Olivier Henchoz pour m'avoir donné la chance de faire face à de grandes responsabilités en tant que *cdt cp e r* et Cécilia Bigler & Sylviane Dal Mas pour l'organisation de la fête de la faculté I&C.

Merci à tous mes amis qui ont contribué de près ou de loin à cette thèse et qui se sont satisfait de ma présence amoindrie à nos activités en commun durant ces quelques années : Alain & Anne, Alban, Alex "JDK", André "le Pfist'r", Axel du conseil, Ban, Barben & Janice, Basile, Bastian & Laura, Bérard, Borlat, Bouvier, Chens & Jenny et le futur petit, Christian et ses "Gagg", Christophe "Dynafit", Claudio, Darko, David, de Chambrier et famille, Denti, Didier, Didier "gren" et famille, Diego, Duboss', FN & Mag', Franck, François "The Dude", Frey, Genton, Goss' & Julie, Grégoire "Gel énergétique", Guillaume & Maryll et leur futur premier enfant, Guillaume & Shahida & Juliette & Clémentine, Jean-Michel, Jesper & Tania, Jo & Katy & Emily, José, Juliette, Ky-Anh, Kaufmann, Landert et famille, Lato “de combat 90”, Lucas “Les Ouais”, Luyet, Marc & Caro, Martial & Carolyne, Mathieu “The Mathiew” & Marie-Julie, Mélanie, Nicolas et Fred “aux dents blanches”, Paulo “le cabri”, Pernet, Perret, PH, Phil & Catherine & Noémie & Rachel & Kilian, Pierre “le dauphin” et famille, R1 & Delphine, Renaud “frère d’arme”, Ron & Laura, Rumley, Sacha, Sami & Carrie, Schouwey, Seb et famille, Stocker & Lili et famille, Stefan, Stefano et famille, Svend, Théo et famille, Vincent & Aline, Yann et famille et Yves “au taquet”. Enterrements de vie de garçons, rallyes, via ferratas, grimpe, descente en rafting, barbecues, Patrouille des Glaciers, Raids, saut en parachute, vol en parapente (entre autres) : que de souvenirs inoubliables partagés avec vous !

Finalement, je remercie ma chère et tendre Olivia de partager avec moi des moments inoubliables, pour tout son amour et son soutien dans les moments plus difficiles. Avec elle tout devient non seulement possible, mais aussi facile ! Je me réjouis déjà des nombreux moments et expériences à venir et je lui promets de plier les genoux à ski et de me donner de la peine, au tennis, pour les engagements.

Preface

This thesis describes the PhD work done at the Distributed Programming Laboratory, School of Computer and Communication Sciences, EPFL, under the supervision of Prof. Rachid Guerraoui, from 2005 to 2010. During this period, besides the work presented in this thesis, I also published the work contained in my master's thesis [MKR06] worked on an event-driven programming model for mobile devices [GGH$^+$06, GGH$^+$07b], on an oracle for measuring and reacting to resource usage in the same context of mobile devices [GGH$^+$07a] and also on distributed computing in social networks [GHKM09a, GHKM09b, GHK$^+$].

This thesis focuses on the problem of live streaming with a gossip-based dissemination protocol and is a composition of published papers [BGKM07, FFM07, FGK$^+$09b, FGK$^+$09a, GHKM09c, FGKM10, GHK$^+$10].

Live Streaming with Gossip

[GHK$^+$10] *LiFTinG : Lightweight Freerider-Tracking Protocol in Gossip.* Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec, Maxime Monod, and Swagatika Prusty. In *International Middleware Conference (Middleware)*, 2010.

[FGKM10] *Boosting Gossip for Live Streaming.* Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, and Maxime Monod. In *Internation Conference on Peer-to-Peer Computing (P2P)*, 2010.

[GHKM09c] *On Tracking Freeriders in Gossip Protocols.* Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec, and Maxime Monod. In *International Conference on Peer-to-Peer Computing (P2P)*, 2009.

[FGK$^+$09a] *Heterogeneous Gossip.* Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Boris Koldehofe, Martin Mogensen, Maxime Monod, and Vivien Quéma. In *International Middleware Conference (Middleware)*, 2009.

[FGK$^+$09b] *Stretching Gossip with Live Streaming.* Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Maxime Monod, and Vivien Quéma. In *International Conference on Dependable Systems and Networks (DSN)*, 2009.

[FFM07] *A Generic Theoretical Framework for Modeling Gossip-Based Algorithms.* Yaacov Fernandess, Antonio Fernández, and Maxime Monod. *Operating Systems Review (OSR)* 41(5) :19–27, 2007.

[BGKM07] *Towards Fair Event Dissemination.* Sébastien Baehni, Rachid Guerraoui, Boris Koldehofe, and Maxime Monod. In *International Workshop on Distributed Event Processing, Systems and Applications (DEPSA), co-located with the International Conference on Distributed Computing Systems (ICDCS)*, 2007.

Distributed Computing in Social Networks

[GHK$^+$] *Decentralized Polling with Respectable Participants.* Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec, Maxime Monod, and Ymir Vigfusson. *Distributed Computing*, under submission.

[GHKM09b] *Decentralized Polling with Respectable Participants.* Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec, and Maxime Monod. In *International Conference On Principles Of DIstributed Systems (OPODIS)*, 2009.

[GHKM09a] *Brief Announcement : Towards Secured Distributed Polling in Social Networks.* Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec, and Maxime Monod. In *International Symposium on Distributed Computing (DISC)*, 2009.

Event-driven Programming Model

[GGH$^+$07a] *The Weight-Watcher Service and its Lightweight Implementation.* Benoît Garbinato, Rachid Guerraoui, Jarle Hulaas, Alexei Kounine, Maxime Monod, and Jesper H. Spring. In *International Conference on Embedded Computer Systems : Architectures, Modeling and Simulation (IC-SAMOS)*, 2007.

[GGH$^+$07b] *Pervasive Computing with Frugal Objects.* Benoît Garbinato, Rachid Guerraoui, Jarle Hulaas, Maxime Monod, and Jesper H. Spring. In *Symposium on Pervasive Computing and Ad Hoc Communications (PCAC), co-located with the International Conference on Advanced Information Networking and Applications (AINA)*, 2007.

[GGH$^+$06] *Frugal Mobile Objects.* Benoît Garbinato, Rachid Guerraoui, Jarle Hulaas, Maxime Monod, and Jesper H. Spring. In *Euro-American Workshop on Middleware for Sensor Networks (EAWMSN), co-located with the International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2006.

Master's Thesis Publication

[MKR06] *Looking Ahead in Open Multithreaded Transactions.* Maxime Monod, Jörg Kienzle, and Alexander Romanovsky. In *International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2006.

Préface

Cette thèse décrit le travail de doctorat effectué au Laboratoire de Programmation Distribuée, Faculté Informatique et Communications, EPFL, sous la supervision du Prof. Rachid Guerraoui, de 2005 à 2010. Pendant cette période, mis à part le travail présenté dans cette thèse, j'ai également publié le travail de mon projet de diplôme [MKR06], travaillé à l'élaboration d'un modèle de programmation événementielle pour systèmes embarqués [GGH+06, GGH+07b], au développement d'un composant servant à mesurer l'utilisation de ressources et à y réagir dans ce même contexte des systèmes embarqués [GGH+07a] et finalement sur du calcul distribué dans les réseaux sociaux [GHKM09a, GHKM09b, GHK+].

Cette thèse traite du problème de diffusion épidémique d'un flux en direct et elle est composée de différents travaux publiés [BGKM07, FFM07, FGK+09b, FGK+09a, GHKM09c, FGKM10, GHK+10].

Diffusion Epidémique d'un Flux en Direct

[GHK+10] *LiFTinG : Lightweight Freerider-Tracking Protocol in Gossip.* Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec, Maxime Monod et Swagatika Prusty. In *International Middleware Conference (Middleware)*, 2010.

[FGKM10] *Boosting Gossip for Live Streaming.* Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec et Maxime Monod. In *Internation Conference on Peer-to-Peer Computing (P2P)*, 2010.

[GHKM09c] *On Tracking Freeriders in Gossip Protocols.* Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec et Maxime Monod. In *International Conference on Peer-to-Peer Computing (P2P)*, 2009.

[FGK+09a] *Heterogeneous Gossip.* Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Boris Koldehofe, Martin Mogensen, Maxime Monod et Vivien Quéma. In *International Middleware Conference (Middleware)*, 2009.

[FGK$^+$09b] *Stretching Gossip with Live Streaming.* Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Maxime Monod et Vivien Quéma. In *International Conference on Dependable Systems and Networks (DSN)*, 2009.

[FFM07] *A Generic Theoretical Framework for Modeling Gossip-Based Algorithms.* Yaacov Fernandess, Antonio Fernández et Maxime Monod. *Operating Systems Review (OSR)* 41(5) :19–27, 2007.

[BGKM07] *Towards Fair Event Dissemination.* Sébastien Baehni, Rachid Guerraoui, Boris Koldehofe et Maxime Monod. In *International Workshop on Distributed Event Processing, Systems and Applications (DEPSA), coorganisé avec International Conference on Distributed Computing Systems (ICDCS)*, 2007.

Calcul Distribué dans les Réseaux Sociaux

[GHK$^+$] *Decentralized Polling with Respectable Participants.* Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec, Maxime Monod et Ymir Vigfusson. *Distributed Computing*, en soumission.

[GHKM09b] *Decentralized Polling with Respectable Participants.* Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec et Maxime Monod. In *International Conference On Principles Of DIstributed Systems (OPODIS)*, 2009.

[GHKM09a] *Brief Announcement : Towards Secured Distributed Polling in Social Networks.* Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec et Maxime Monod. In *International Symposium on Distributed Computing (DISC)*, 2009.

Modèle de Programmation Evénementiel

[GGH$^+$07a] *The Weight-Watcher Service and its Lightweight Implementation.* Benoît Garbinato, Rachid Guerraoui, Jarle Hulaas, Alexei Kounine, Maxime Monod et Jesper H. Spring. In *International Conference on Embedded Computer Systems : Architectures, Modeling and Simulation (IC-SAMOS)*, 2007.

[GGH$^+$07b] *Pervasive Computing with Frugal Objects.* Benoît Garbinato, Rachid Guerraoui, Jarle Hulaas, Maxime Monod et Jesper H. Spring. In *Symposium on Pervasive Computing and Ad Hoc Communications (PCAC), coorganisé avec International Conference on Advanced Information Networking and Applications (AINA)*, 2007.

[GGH$^+$06] *Frugal Mobile Objects.* Benoît Garbinato, Rachid Guerraoui, Jarle Hulaas, Maxime Monod et Jesper H. Spring. In *Euro-American Workshop on Middleware for Sensor Networks (EAWMSN), co-organisé avec International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2006.

Publication du Projet de Diplôme

[MKR06] *Looking Ahead in Open Multithreaded Transactions.* Maxime Monod, Jörg Kienzle et Alexander Romanovsky. In *International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2006.

Table of Contents

Labor disgraces no man; unfortunately you occasionally find men disgrace labor.

Ulysses S. Grant

1 Introduction

1.1 Context and Objectives

The dissemination of information is ubiquitous in our lives. From oral conversations, such as coffee break discussions, meetings or (mobile) phone calls, to visual communication, such as newspapers, television, emails, images, videos, we are all the time exposed to, absorbing and participating in the dissemination of information, be that in a narrow one-to-one fashion with friends and family or in a broader one/many-to-many fashion with people we hardly know, e.g., through usage of blogs and social networks. Throughout the last decade, the intensity of this information dissemination has increased tremendously as the traditional communication channels have been undergoing a transformation towards digitalization over the Internet, examples include paper-based regular mails to emails, phone calls to Voice-over-IP (VoIP), online libraries, podcasts and live streaming, be it radio or TV broadcasting.

This convergence towards Internet-based communication channels combined with new habits and scale in communication patterns have put an increasing amount of demands on the computer systems facilitating this. In such an environment, these systems, which by nature are distributed, must be *(i)* dynamic: adapt to and tolerate failures and constantly joining and leaving of communication participants, namely *churn*, and *(ii)* scalable: be able to scale as the number of communication participants grow from a couple of users to millions of users, and the amount of data they exchange grow from kilobytes to terabytes, representing longer or higher quality content. Under these challenging circumstances, these distributed systems must be able to provide the following guarantees in order to be considered suitable for communication: *(i)* time and order: the data sent must arrive to the communication partners in a timely and orderly manner not to disrupt the communication, and *(ii)* reliable and secure: the data sent must arrive to the correct communication partner and without having been altered.

Peer-to-peer (P2P) or *decentralized* architectures have emerged as a popular paradigm to support the dynamic and scalable nature of such distributed communication systems. In a P2P system, a set of participants gather on an ad-hoc basis to participate in some *collaborative tasks*. What is characteristic about a P2P system is that there is no central control. Participants can join and leave at will, and with no central control, there are no single points of failure. This way, at any point in time, a P2P system really represents an ever-changing environment of collaborating participants that are ideally all willing to equally contribute resources, such as memory or bandwidth, in the execution of the task towards a result, providing a benefit to all participants.

File sharing is probably the most widely used collaborative task, where each participant wants to receive an individual copy of the shared file and collaborates by sending fragments of the file they have already downloaded to other participants. The contribution of the participants, in terms of resources, is therefore a fraction of their bandwidth for uploading/relaying received fragments of the shared file, whereas their benefit is to receive all the fragments making up a single complete copy of the shared file. From the point of view of the participant, what is typically of importance when receiving a file is to get it as fast as possible and in its entirety. The order in which the fragments arrive is irrelevant as the file is typically not opened before being complete.

Sharing of multimedia content represents a special case of file sharing that is subject to a number of constraints. What characterizes files containing multimedia content is their size, typically easily reaching the hundreds of megabytes to gigabytes. Compared to the bandwidths of participants (typically hundreds of kilobits per second to a couple of megabits per second), waiting until completion before actually being able to use the file represents a non negligible time, meaning that from the point of view of the participant receiving the multimedia file, getting the (entire) file as fast as possible is typically not good enough. Video on Demand (VoD) represents such a scenario. In VoD, the participant receiving the multimedia content, offered by some source, would like to start previewing the received multimedia content (the stream), even though only a fraction of it has been received, and then continue the viewing while the remaining parts of the file is received.

This gives rise to timing and ordering constraints on the sharing of multimedia files, in that *(i)* it would not make sense to see, say, the end of a movie before the start, and *(ii)* when starting to preview the movie, one would like to continue like this without any disruptive pauses due to communication hiccups. The time the participant has to wait until the stream starts playing is called the *buffering delay*. The buffering delay can span from almost nothing, i.e., the rate at which content arrives is much larger than the rate at which it is played (usually the case for low quality podcasts or videos) to the time needed to transfer the whole file in the worst case, i.e., the download rate is so poor compared to the playing speed that the client needs to wait to receive the file almost entirely before starting to play it. Optimal buffering strategies are studied in [CE07]. At the end of the streaming process, the played stream – representing the result of the collaborative task – is then either trashed, such as when watching videos on Youtube [You], or is stored locally into a file that can be played back later.

Following the same line of reasoning, new applications emerged introducing *live streaming*: the source does not own a file that it wants to share with others, but shares content as soon as it is produced. In other words, the content to distribute is *live*, not pre-recorded and stored. Typical examples include the broadcasting of live sports events, conferences or interviews. As in VoD, from the point of view of the receiving participant there is a clear incentive for playing the received content as soon as possible. Contrary to VoD, however, data itself is not available in advance, and hence no data can be downloaded ahead of time.

The *gossip paradigm*, also named *epidemic dissemination* [EGKM04] or *rumor mongering* [KSSV00], introduced in [DGH$^+$87], is a type of data dissemination that relies on random communication between participants in a P2P environment. An epidemic starts to spread when a source randomly chooses a set of communication partners, of size *fanout*, and infects them, i.e., it shares a rumor with them. This set of participants, in turn, randomly picks fanout communication partners each and infects them, i.e., share with them the same rumor. This paradigm has many advantages including *(i)* fast propagation of rumors, *(ii)* probabilistic guarantee that each rumor reaches all participants, and *(iii)* high resilience to churn and high scalability [KMG03]. Gossip therefore constitutes a candidate of choice for live streaming in large-scale systems, as documented in the literature [DXL$^+$06, LCW$^+$06, LCM$^+$08].

These advantages, however, come at a price. By spreading around a rumor randomly, gossip creates many duplicates of the same rumor and nodes usually receive multiple copies of the same rumor. While this is obviously a feature when it comes to guarantee a good dissemination of the rumor in case of churn, it is a clear disadvantage in case of spreading large amounts of multimedia content (i.e., time and order-critical data) with participants having limited resources, namely upload bandwidth.

With this in mind, this thesis investigates *if* and *how* the gossip paradigm can be used as a highly efficient communication system for a live streaming scenario under the following specific scenarios: *(i)* where participants can only contribute limited resources, *(ii)* when these limited resources are heterogeneously distributed among nodes, and *(iii)* where an ideal state in the P2P environment is not reached, meaning that participants are not contributing their fair share of work. These scenarios represent a number of challenges:

Live Streaming with Gossip in Constrained Environment Gossip is creating many duplicates of rumors as the dissemination spread follows a random pattern. Nodes can therefore receive multiple copies of the same rumors resulting in a clear waste of bandwidth in case the rumors themselves represent relatively large content. Since gossip provides only probabilistic guarantees that each rumor will reach every node, it is not guaranteed that each node receives every produced rumor. In addition, with constrained bandwidth and unreliable communication channels, rumors can also be dropped or lost. Gossip thus needs to be tailored in such a way that the usage of bandwidth fits the constrained bandwidths of nodes and boosted so that each node does not miss any produced rumor.

Heterogeneous Bandwidth-constrained Environment Gossip inherently asks all nodes to contribute the same amount of upload bandwidth as they are all supposed to send roughly the same number of rumors to the same fanout number of partners. Gossip is namely *load-balancing* while large-scale systems exhibit very heterogeneous nodes' capabilities. Since the best efficiency in dissemination is achieved when all contributed resources are optimally used, it seems trivial that balancing the load equally on all nodes can only underutilize nodes with high capabilities and overload nodes with low capabilities. Gossip therefore needs to be able to adjust the load on each node proportionally to the amount of resources they can actually contribute.

Presence of Freeriders Some participants might not want to contribute their resources, for various reasons. They can have so limited resources that they always try to benefit more than what they can actually contribute. They might realize that a small decrease in benefit results in a greater decrease in contribution and thus can save some resources for other tasks. Worse, this situation is usually amplified when participants selfishly collaborate in order to benefit within the coalition without contributing as expected. Participants that do not contribute their fair share are called *freeriders* and *colluding freeriders* when they collaborate to decrease their contribution within a coalition. Gossip must therefore provide means to either tolerate a proportion of freeriders without impacting on the benefit of honest nodes or detect and expel freeriders from the dissemination.

1.2 Contributions

The main contributions of this thesis are the following:

Live Streaming with Gossip in Constrained Environment

- We show, empirically, that in a bandwidth-constrained environment, participants cannot arbitrarily increase the fanout parameter to increase the reliability of the dissemination.

  In bandwidth-constrained environments, participants contribute to the system only a limited upload bandwidth, comparable to the stream rate that the source produces. Dissemination of a clear stream, the exact copy of the original stream, to all nodes is theoretically feasible if the average upload bandwidth is larger than the stream rate. Theoretical works on gossip usually show that the fanout is an obvious knob for tuning reliability, in other words, the larger the fanout the higher the probability the initial rumor has chances of reaching all nodes in the system. In practice, our results show that there is a clear optimal range of fanout values and that increasing the fanout too much has negative impact on the dissemination.

- We show, empirically, that the frequency at which participants change communication partners has to be the highest possible.

  Theoretical work on gossip usually assume that participants pick different fanout partners at each gossiping period, i.e., the set of fanout partners change at each communication step. In opposite, many recent work try to disseminate information epidemically in a *mesh*, where each node has a fixed set of communication partners and are bound to them for their entire lifetime in the system. Keeping the dissemination simplicity of gossip, we show that changing neighbors dynamically and continuously provides the best results and that epidemics in mesh-based systems have to tackle problems that are inexistent in gossip, e.g., explicitly splitting the source stream to only name one.

- We present an extension of a gossip-based dissemination protocol, named gossip++, in the context of high-bandwidth content dissemination, suitable for live streaming in large-scale systems.

  The proposed solution integrates both well-known erasure coding techniques, in order to circumvent the probabilistic delivery guarantees of gossip, and a new retransmission technique, leveraging the many duplicates induced by gossiping. Effectively, by gossiping information to fanout communication partners chosen at random, participants can receive anything from multiple copies of the same information to none of it. Erasure coding ensures that the missing information can be reconstructed while the many duplicates act as different sources to request information from in case of communication problems, such as message losses or node failures. We extensively test the proposed gossip-based protocol in various scenarios and show that it tolerates catastrophic failures very well and a reasonable fraction of freeriders when the average capability of the system is larger than the demand, i.e., when the average upload bandwidth is larger than the stream rate.

Heterogeneous Bandwidth-constrained Environment

- We present a new heterogeneity-aware protocol, named HEAP, that adapts the contribution of participants according to the distribution of their capabilities.

  In practical systems, participants usually participate in the collaborative task with very different devices, spanning from mobile phones to desktop computers. Not only do the devices themselves have different characteristics but their connection to the Internet also shows very different capabilities be they at home behind broadband connections, at work with shared and protected accesses or on the move with mobile connections. On top of that, users can typically manually limit the amount of bandwidth they want to devote to their different bandwidth-greedy applications, resulting in very different capabilities given to the system in the end. Gossip has been recognized as a very practical and easy way to homogenize or balance the overall load equally on all participants. This is a very nice feature in an homogeneous system where all nodes devote the same amount of resources to the system but applies very poorly when participants have heterogeneous capabilities. Indeed, some participants see their resources underutilized, as only a very small fraction of the resources they are ready to provide to the system are actually used whereas some other participants cannot provide the amount of resources they are requested to. HEAP consists in asking nodes to contribute proportionally to their resources, irrespectively of the distribution of them, as long as the average amount of resources available is sufficient to sustain the dissemination of the stream rate.

Presence of Freeriders

- We present a lightweight freerider-tracking protocol, named LiFT, that detects and expels nodes that do not provide their fair share of work.

  In systems where the average capabilities of participants is much higher than the demand or where the benefit is not directly correlated to the contribution to the system as in *asymmetric*, push-based systems like the gossip-based protocols proposed, freeriders are tempted not to provide their fair share of work if they perceive a limited negative impact on their benefit. As the fraction of freeriders increases or the average capability of the system gets closer to the demand, the negative impact of such nodes increases until most nodes cannot actually play the stream. We analyze the proposed gossip protocol and describe the many attacks that such freeriding nodes can perform and propose LiFT: a protocol that provides lightweight verification mechanisms on top of gossip, for detecting nodes that freeride and even collude. As opposed to *incentives* encouraging nodes to contribute in order to benefit from the system, the approach considered is *coercive* in the sense that nodes that are caught misbehaving are expelled from the system, constituting by itself a way of encouraging nodes to collaborate as they are supposed to. LiFT is based on accountability in the sense that participants' actions are logged and cross-checked. Since communication partners change frequently and that a majority of participants have to be honest so that the stream is correctly disseminated, the cross-checked information converges eventually in detected misbehaving nodes that are gradually removed from the system, resulting in both a faster detection mechanism and a better dissemination, as it decreases the proportion of freeriders.

1.3 Thesis Organization

The thesis is split into 8 chapters, organized as follows.

Chapter 2 reviews the fundamental concepts behind high-bandwidth content dissemination, clearly positions the problem of live streaming, as opposed to file sharing and Video on Demand (VoD), and presents an overview of related work in the context of high-bandwidth content dissemination, focusing mainly on live streaming approaches.

Chapter 3 exposes the gossip paradigm in its different variants, proposes a gossip-based protocol designed for high-bandwidth content dissemination and describes the experimental setup used throughout the thesis.

The proposed three-phase gossip protocol is evaluated to understand the impact of its key parameters, namely the fanout and the proactiveness in the partner selection scheme, in Chapter 4.

Chapter 5 proposes two complementary mechanisms, *Claim* and *Codec*, to improve the protocol's efficiency, resulting in a protocol named gossip++.

Chapter 6 enables gossip to adapt to nodes' capabilities, that is, we propose a new gossip protocol, called HEAP, that adapts nodes' contribution proportionally to their capabilities, namely their upload bandwidth.

Chapter 7 exposes the issues due to freeriding in high-bandwidth content dissemination, how the behavior of freeriders can affect existing approaches and proposes a protocol to limit their impact on the proposed three-phase gossip protocol, namely a lightweight freerider-tracking protocol (LiFT) tailored for asymmetric systems.

Finally, Chapter 8 gives a conclusion, summarizing the main results of this work and indicating directions for future research.

The education of a man is never completed until he dies.

Robert E. Lee

2 Fundamental Concepts & Existing Approaches

This chapter reviews some fundamental concepts of content dissemination in large-scale systems and existing approaches that tackles the challenges of high-bandwidth content dissemination. In distributed systems, broadcasting data from a single source to a large number of clients represents a key challenge and could be compared to the problem of sorting in the context of data management as it is a key building block for collaborative tasks. The data to disseminate is usually present on a single source, either produced on the go or as an existing file, and must be distributed to a large set of clients. The size of that data can be very small, consider temperatures that have to be disseminated in a sensor network, or very large, consider applications, operating systems or movies that can span from hundreds of megabytes to several gigabytes that are typically exchanged on the Internet.

We focus on the problem of broadcasting large content in comparison to the bandwidth capabilities of the participants, namely high-bandwidth content dissemination. The problem of sending data from one to many can also be named multicast and commonly refer to *IP multicast* [Dee88, DC90], i.e., being able to send packets from a source and have them delivered to anyone that needs them. In practice, current networks cannot provide such a primitive for many reasons [Fra, CRSZ02], including protocol complexity at the network layer and pseudo-economical matters, e.g., Internet Service Providers (ISP) not wanting to duplicate content in their network, especially if this content was produced in a network managed by some other ISP. This lack of primitive at the network layer has led researchers and engineers to design protocols for high-bandwidth content dissemination at the *application layer*, named *end-system multicast* [CRSZ02].

2.1 High-bandwidth Content Dissemination

In high-bandwidth content dissemination, the data to broadcast is either a file that is large enough so that its transfer takes a significant time or the content to disseminate is *unbounded* and represents a significant fraction of the available bandwidth when streaming it. Unbounded content typically represents a stream that is produced at runtime and which duration or size cannot be determined. In that case, it is the amount of information per second produced, namely the *stream rate* which has to be compared to the bandwidth capabilities of the participants.

2.1.1 File Sharing

A file is an abstraction that represents data of various content, from programs to documents, music or movies. From the very beginning of the Internet, clients have always had the need to share content with each other thus sending files to each other. Clients are usually considered *impatient*, i.e., they want to receive data as soon as possible, that is, as fast as possible. The transfer speed is limited by the network characteristics of the two communicating clients, i.e., the bandwidth they can dedicate to transfer the file. The duration $d_{A\to B}$ to transfer a file of size S from a server A to a client B, knowing the upload bandwidth A_{up} of the server A and download bandwidth B_{down} of the client B is given as:

$$d_{A\to B} = \frac{S}{\min(A_{up}, B_{down})} + \kappa_{A\to B} \ \ ,$$

where $\kappa_{A\to B}$ is the time needed to establish a connection between A and B. Assuming S is very large compared to the upload and download bandwidths, the delay $\kappa_{A\to B}$ to establish a connection becomes negligible.

Client-Server Approach A naive solution to the problem of file sharing is to let the source send the whole data to all clients or equivalently let all interested clients request data from the single source. Assume the server A serves n clients with a file of size S, the duration $d_{A\to\{c_1,\ldots,c_n\}}$ for sharing the file from A to all users $u_1, \ldots, u_n$ becomes:

$$d_{A\to\{u_1,\ldots,u_n\}} = \sum_{i=0}^{n} d_{A\to u_i} = \sum_{i=0}^{n} \left(\frac{S}{\min(A_{up}, u_{i_{down}})} + \kappa_{A\to u_i} \right) \ .$$

Such a solution is still very common nowadays. Take the example of a server that provides a file to download, e.g., a new version of a device driver. All interested users connect, via HTTP or FTP, to the server and download the file. The server can provide data to users concurrently and usually serves many files, thus many users at the same time, decreasing the amount of bandwidth A_{up} it can dedicate to each client. Assuming the upload bandwidth of the server is the bottleneck, i.e., it is smaller than any of the download bandwidths of the clients, serving the n clients concurrently does not decrease the duration of the task since the upload bandwidth of A is concurrently shared between the n clients, i.e., A_{up}/n, resulting in:

$$\frac{S}{A_{up}/n} + \sum_{i=0}^{n} (\kappa_{A \to u_i}) = n \cdot \frac{S}{A_{up}} + \sum_{i=0}^{n} (\kappa_{A \to u_i}) = d_{A \to \{u_1, \ldots, u_n\}} \ .$$

Since clients want to finish their transfer as soon as possible, we say that they *compete* in order to have the largest amount of bandwidth dedicated to them. In that scenario, the server can either *(i)* equally share the load among all concurrent transfers by providing an equal fraction of its upload bandwidth A_{up} to all concurrent n clients (i.e., A_{up}/n, which in case of a very large number of concurrent transfers, will make the clients feel like their transfer is stalled, or *(ii)* allow a maximum number of concurrent clients U to which the server can guarantee a minimum quality of service (i.e., a minimum upload bandwidth of A_{up}/U) and thus deny other requests when $n > U$. Both solutions do not scale with the number of concurrent clients. Additionally, the server represents a single point of failure and in case it crashes, the service is temporarily down.

Replicating Servers with Brokers In order to partially solve the aforementioned issues, a single server can be replaced by *brokers*, i.e., a set of servers where the data to share is replicated, in other words acting as mirrors. The clients' requests are distributed among the different replicas, themselves serving clients with their own bandwidth. This solution does, indeed, solve the problem of failures but only partially solve the issue of scalability. Effectively, the number of concurrent requests is increased linearly with the number of mirrors at the cost of maintaining those multiple servers and paying the bandwidth usage of the different serving replicas.

Until now, we considered that the clients download the file sequentially or *in-order*, i.e., from the beginning of the file to the end of it. Note that in a non congested mode, a client could be, for example, concurrently served by two replicas: the first one serving the first half of the file and the other one serving the second half, concurrently. This very simple example opens new ways of sharing data since the data does not need to be downloaded in a particular order.

Peer-to-peer Approach In order to stop competing for resources, peer-to-peer (P2P) architectures have emerged so that clients *collaborate* to achieve a common goal: to create a replica of the original file onto all clients' machines. In that sense, each client is asked to dedicate some of its resources, its upload bandwidth in particular, for the good of the community, in order to itself transfer to others the data it has already received.

In the 1990s, the P2P approach was made very popular with systems like Napster, KaZaA or eDonkey for sharing – illegally or not – music content in the MP3 file format. The idea is pretty simple. At the beginning of the process, the file is present only on the source node (the server in the client-server architecture) and split into many subparts, namely *chunks*. Each chunk has an identifier corresponding, for instance, to its position in the file. Clients interested in that file connect to the source which starts to serve them with different chunks and also tells them who is part of the collaborative process. Since clients received different chunks from the source, they can now also act as source of those received chunks for the other clients and serve those that did not yet receive those chunks. In a sense, clients do not only benefit from the upload bandwidth of the source, but also from the upload bandwidth of the other clients, as soon as they have received some data to share. The more data clients receive, the more they can send until each client reaches the upload bandwidth it decided to devote to the collaborative task.

In theory, the overall upload bandwidth of the collaborative task is simply equal to the one of the source in a client-server architecture and can be as large as the sum of all participants' upload bandwidth in a P2P system.

In practice, many issues need to be solved. How do clients get to know each other and what clients should connect to what other clients? In what order and to whom should chunks be disseminated? What happens if clients start to join or leave the system? After having introduced Video on Demand (VoD) and live streaming in Sections 2.1.2 and 2.1.3 respectively, we describe how existing approaches tackle those issues by building *overlays*. The additional problems of heterogeneity in contribution and non-collaboration in existing approaches are reviewed in Sections 6.3 and 7.5 respectively.

2.1.2 Video on Demand (VoD)

When the content to share is audiovisual, typically a movie, and the clients want to watch the movie starting at any given point in time, the file sharing problem specializes into Video on Demand (VoD). A file of a given size S now represents the duration t of a movie. With $S = 700\,MB$ (to typically fit onto a CD) and $t = 90\,\text{minutes} = 5400\,s$ (average duration of a movie), the audiovisual content represents a stream rate s of roughly $1000\,kbps$ (i.e., S/t).

If a client is receiving the corresponding file at a rate r much larger than the stream rate s, the client can start to watch the movie right away and the movie will not be stopped before its end, i.e., after the duration t of the movie.

On the other hand, if the rate r at which the client is receiving the file is smaller than the stream rate s, the client has to wait, in order to accumulate enough data, in such a way that once it has started playing, it knows it has enough data buffered so that it can play the movie and the rest of the movie will be downloaded on time. The time δ the client needs to wait until it can start playing is named the *buffering delay*.

Assume the client receives data at a rate r on average, and smaller than the stream rate s (i.e., $r < s$). The minimal buffering delay δ_{min} is given as:

$$\delta_{min} = \left(1 - \frac{r}{s}\right) t = \left(1 - \frac{r}{s}\right) s \cdot S \ \ .$$

Assuming the download rate is three-fourths of the stream rate, the client needs to wait at least one-fourth of the length of the movie before playing to make sure it will not be interrupted. The calculated buffering delay is minimal only if the download rate r is used to retrieve data in the order it is played, e.g., the data buffered until starting to play represents the very first δ_{min} duration of the movie.

In practice, clients might want to start watching a movie from a given point in time of the movie (e.g., from the third chapter on), which makes the problem very challenging, especially when trying to make clients collaborate. The issue is that some clients are watching the beginning of the movie while some others are watching the end of it, thus not needing the first part of the movie that the first group could propose to send. We do not elaborate any further on the problem of VoD in order to concentrate on live streaming. We refer to [DLHC05, AGR06, LCC07, SDK$^+$07, VGK$^+$07] for further reading about VoD.

2.1.3 Live Streaming

The problem of live streaming shares similarities with the challenges of VoD since the data to disseminate represents audiovisual content and that there is thus a need for buffering data before starting to play it. It is still very different in the sense that the data to disseminate is not available *a priori* but produced at runtime. In essence, this means that clients can of course not choose at what time of the stream they want to start playing, but it also means that the rate at which the stream is received cannot be lower than the stream produced. In other words, since a client does not know *a priori* how long it will watch the stream (as opposed to the duration t of a movie), there exists no buffering delay δ for watching the stream without being interrupted if the reception rate r is smaller than the stream rate s.

If the reception rate r is much larger than s, the client can also not benefit from it to download data in advance, as this data is not produced yet.

Assuming the download capability of users is larger than their upload capability, typically the case for users behind cable or ADSL connections, the bottleneck for the collaborative task is considered to be the upload capabilities of nodes. In that sense, a live streaming system can be designed successfully if and only if the sum of all upload bandwidths contributed to the system by the source A and all users u_i (n of them) is larger than or equal to the stream rate to receive for each user (i.e., excluding the source), that is:

$$A_{up} + \sum^{n} u_{i,up} \geq s \cdot n \ \ .$$

Another way of looking at this inequality is the following [KLR07]. Given a sum of bandwidth capabilities $A_{up} + \sum^{n} u_{i,up}$, what is the optimal stream rate s that can be broadcast to all nodes?

$$s \leq \min \left(A_{up}, \frac{A_{up} + \sum^{n} u_{i,up}}{n} \right)$$

The maximum stream rate is obtained by the equality but is never reached in practice. Effectively, any protocol will induce overhead for either signaling data or at least transporting it to different nodes. A system thus aims at having this overhead as low as possible so that a stream of higher rate can be broadcast.

A live streaming system should aim at providing a high quality stream, i.e., the exact same copy of the stream produced by the source if possible, to a maximum number of clients concurrently, and letting them start viewing the stream as soon as possible. Effectively, for a live stream, the notion of *live* does not only mean the stream is produced at runtime but also that the client would like to experience a delay between reality and its viewing experience that is as short as possible. We define the *stream lag* as the time difference between the moment at which the stream is sent from the source and the moment at which it is played on the client.

The challenges in a live streaming system are thus summarized as:

- **Minimize the overhead of the protocol** Design a lightweight system so that the stream rate broadcast is as close as possible to the average upload bandwidth of participants. In other words, given a stream rate s, have a system that can both *(i)* adapt to the heterogeneity of upload bandwidth, and *(ii)* where the average upload bandwidth of clients can be decreased as close to s as possible,
- **Maximize the stream quality:** provide a stream that is as close as possible to the original stream produced by the source. We want the protocol to ideally provide an exact copy of the stream produced by the

source to all participants, and if not, a stream that contains a minimum number of missing data,

- **Minimize the buffering delay** Let clients start watching the stream as soon as possible after they start receiving data,
- **Minimize the stream lag** Provide a stream that is as live as possible to all nodes,
- **Maximize simplicity** Provide protocols that are as simple as possible to ease their implementation and deployment over a very large scale system.

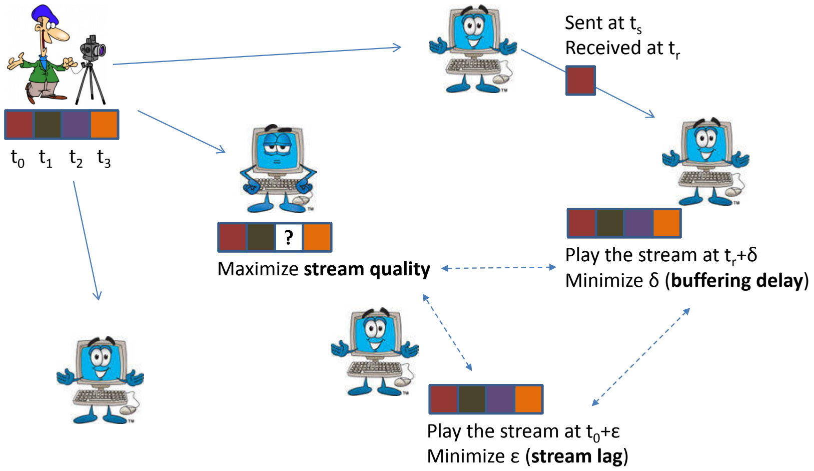

Sent at t_s
Received at t_r
t_0 t_1 t_2 t_3
?
Maximize **stream quality**
Play the stream at t_r+δ
Minimize δ (**buffering delay**)
Play the stream at t_0+ε
Minimize ε (**stream lag**)

Figure 2.1: The concepts of live streaming: a source node produces a stream from time t_0 on and wants to broadcast it to all participants.

Figure 2.1 presents the concepts of live streaming and illustrates the notions of stream quality, buffering delay and stream lag. Assume the source produces the very first stream data unit at time t_0, a participant will play it at time $t_0 + \varepsilon$ where ε represents the stream lag. Given that a participant receives its very first data unit at time t_r, it will play the stream at time $t_r + \delta$ where δ is the buffering delay. Both notions of buffering delay and stream lag are interconnected but still differ. Intuitively, the more a participant waits before playing the stream thus increasing its buffering delay, the less the stream can be live, i.e., the more the participant also increases its stream lag. On the other hand, imagine that in a system, a participant can wait sufficiently long until it is *directly served* only by the source. In that case, the participant has waited possibly quite a long buffering delay but since it is served directly by the source, its stream lag cannot is now very much reduced. Similarly, notions of stream quality, buffering delay and stream lag are also interconnected. In order to increase the stream quality, i.e., to reduce the amount of missing data, a participant would benefit in increasing its buffering delay so as to have more chances to recover the missing data, thus intuitively also increasing its stream lag.

2.2 Communication Graph & Overlay

The notion of which participant is served by which other participant is encompassed in the concept of dissemination *overlay*. Before diving into the concept of overlay, we first expose what a *communication graph* is since it restricts the overlay that can be built on top of it. It is important to clearly differentiate the overlay from the underlying communication graph. The communication graph consists in all possible communication channels between nodes, i.e., it states if it is possible, in the networking sense, to establish a communication between two nodes u_j and u_i. The communication graph represents the underlying network layer. The overlay represents a subset of the communication graph in such a way that a node is given a neighborhood or a view on the system from which it can pick nodes to communicate with. The overlay is therefore an application connectivity graph.

With a clique communication graph (i.e., a network where it is possible for all participants to communicate with any other participant in the system) nodes can either have a global view on the system (i.e., their neighborhood represents all other nodes in the system) or, in order to scale, nodes are restricted to communicate with only a subset of all participants, that is, it has a restricted neighborhood to which it is *connected*. The resulting connectivity graph is what we name an overlay. It is thus an application-level communication graph, on top of the communication graph.

2.2.1 Communication Graph

To model which nodes can directly communicate among each other, we use a communication graph. The communication graph of a system model is a graph $G(V, E)$ that consists of the set V of n nodes, each having a unique identifier (i.e., address), interconnected by a set of edges E. Further, a communication channel of bounded capability (i.e., limited upload/download bandwidth), latency and given reliability is associated with each edge.

The communication graph can change over time. If that is the case, we denote by $G(t)$, $V(t)$, and $E(t)$ the graph, active nodes, and available communication channels at time t, respectively. Furthermore, the characteristics of nodes and channels can also change over time. As a summary, the most important elements of this underlying system are:

Channels The communication channels associated with each edge $e \in E(t)$ have a limited upload/download bandwidth, latency and given reliability and direction. Consider a wireless sensor network, some devices might be capable of bidirectional transmissions whereas smaller/others not. Another parameter that describes the communication channels is whether the communication is point-to-point or a node can broadcast on all its channels in one single operation. Finally, communication channels can restrict the message length that nodes can use to exchange information. We focus on point-to-point communication channels where the bottleneck is the upload bandwidth $u_{i,up}$ of nodes.

Network Topology The network, by its nature (e.g., wireless), can restrict the communication patterns by allowing or not different nodes to be connected together via the edge set $E(t)$. Note that the above patterns can change and evolve over time, e.g., mobile nodes that have a limited wireless transmission range. We denote the possible communication partners of a node u at time t as $W_u(t) = \{p : (u \in V(t)) \wedge ((u, p) \in E(t))\}$.

Churn Nodes can appear and disappear (become active and inactive) from the system for application (or user-defined) reasons (e.g., join/leave) or for technological reasons (e.g., node crash, failures) at any time. As exposed, $V(t)$ is the set of active nodes at time t.

2.2.2 Overlay

It is usually the case that the set of possible communication partners $W_{u_i}(t)$ of every node is very large. This being, for scalability reasons, it is common that only a subset of the set $W_u(t)$ is known and used by u. This subset represents the *neighborhood* of the node in the overlay. In order not to think that neighborhood has anything to do with location or proximity in the underlying communication graph, the term *view* is preferred. The view that node u locally has of the whole system at a certain time t is denoted $view_u(t) \subseteq W_u(t) \subseteq V(t)$. The union of all these views form an overlay of the underlying network topology. The communication between nodes (i.e., which node(s) u can pick to communicate with) is thus restricted first by the network (to the set $W_u(t)$) and second by the overlay network (to the set $view_u(t)$).

The presence of nodes in each others' views is not symmetric. If a node A is present in the view of B, $view_B$, it does not mean B must be present in $view_A$, i.e., B can initiate a communication with A (both one-way or two-way) but A might not be able to initiate a communication with B since B might not be in A's neighborhood.

In the following, we assume that the communication graph is always a clique and review how the views of nodes, namely the sets $view_{u_i}(t)$, are used to form an overlay. Figure 2.2 gives an overview of the classification we propose in this thesis. The two main families are *structured* and *unstructured* overlays. In structured overlays, the views of nodes reflect a given hierarchy or well-defined classification or ordering, resulting in a structure that respects a specific semantics, e.g., some nodes are chosen to be closer to the source than others or nodes are connected according to some semantics. In unstructured overlays, the connectivity of nodes is arbitrary, i.e., nodes are assigned random views.

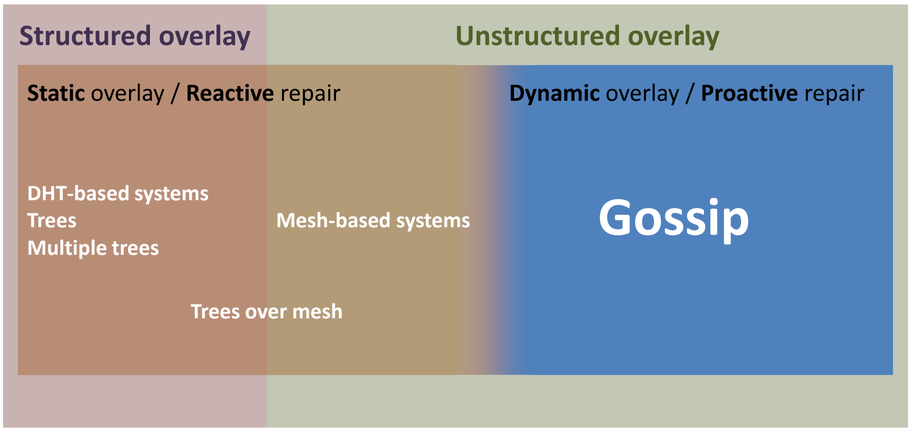


Figure 2.2: Classification of overlays.

Overlays can also be *static* or *dynamic*. If the views of nodes persist until some nodes join or some other leave, the overlay is said to be static (i.e., $\forall t : view_{u_i}(t) = view_{u_i}$ as long as there is no churn). In essence, if there is no churn, the views stay the same over time. If there is churn, the views have to adapt to new arrivals and the resulting overlay needs to repair itself in case of departures – it is thus a *reactive repair*. In a tree for instance, the term *repair* is illustrated by the fact that if any node that is not a leaf node leaves the tree, a branch of the tree (i.e., the subbranch originating from the leaving node) becomes isolated from the rest of the tree and the dissemination is interrupted. In that case, the structure needs reparation.

Dynamic overlays, on the other hand, have periodically changing views. While periodically changing the neighbors of a node in a structured overlay might not make much sense, since the overlay is usually built with a specific semantics attributing a position to each node following a hierarchy or classification, assigning new neighbors arbitrarily and dynamically in unstructured overlays does not break any semantics. We thus clearly differentiate between mesh-based systems and gossip systems. Mesh-based systems are unstructured and static whereas gossip systems are unstructured and dynamic. In the case of dynamically changing views, churn has very limited impact as long as the randomly and dynamically changing views themselves reflect the arrivals and departures of nodes as fast as possible. In opposition to static overlays where the reparation is reactive, we say the reparation process of dynamic overlays is *proactive*. In essence, the overlay itself, by dynamically changing, has to automatically add joining nodes and discard leaving ones to tolerate churn.

2.3 Existing Approaches

In this section, we review existing approaches implementing collaborative tasks with the help of different types of overlays.

2.3.1 Structured Overlays

In structured overlays, there exists either a hierarchy or a well-defined classification between nodes. For instance, a tree is a structured overlay where the source is the root of the tree, and has a set of children with which it communicates. The view of the source is thus a list of nodes which represent the first generation in the tree. These nodes in turn, have children that have children themselves until some nodes, representing the leaves of the tree, have no children to push data to. With this type of overlay, the data flows from the source to its set of children and from these children to theirs until data reaches the leaf nodes which do not forward data any further, being the last to be served.

DHT-based Overlays

Distributed hash tables [RD01, RFS+01, SMK+01, ZHS+04] provide nodes with a distributed lookup service on which different applications can be implemented, such as file sharing, multicast or content distribution systems [AGBH03,FFM04]. The DHT thus represents an infrastructure on which another overlay is built, typically a tree or a multiple-tree overlay [SGMZ04, BRP+05, BB05, LMSW07] in order to implement multicast and thus streaming.

Tree-based Dissemination

A tree overlay for multicast or streaming can be built on top of a DHT or on top of a mesh [Dee88, DC90, Fra, CRSZ02]. The challenge is to create a tree in such a way that the data disseminated reaches all nodes, i.e., the tree must cover the whole population of nodes, and be optimal for dissemination, e.g., minimize delays, both stream lag and buffering delay in the case of streaming.

In a tree approach, failing nodes have a huge impact on their children since they isolate their subbranch from the rest of the tree. A failing node that is a child of the source isolates a larger number of nodes than a leaf or a parent of leaves and thus has a comparatively much worse impact on dissemination. The isolated nodes have thus to repair the structure, that is, try to reconnect to it finding new parents and getting new children assigned. The larger the number of those repairing nodes, the larger the load on the system is for recovering, possibly also having an impact on the dissemination for surviving nodes too.

In addition, the stream that is broadcast from the source to its children and from these children to theirs is usually considered multiple times smaller than their upload bandwidth. The reason is that, in order to form a tree, the stream has to be served from a node to multiple children (e.g., at least two), which means a node that is supposed to serve f children must have at least $f \cdot s$ upload bandwidth (where s is the stream rate), e.g., [SGMZ04]. Last but not least, the leaf nodes do not contribute any upload bandwidth to the system, since they are not serving the stream to any children.

To illustrate this, assume every node has the same upload bandwidth of $f \cdot s$, in other words, each node can serve up to f children simultaneously. The total upload bandwidth available in the system is therefore $n(f \cdot s)$ where n is the number of nodes (including the source). With $f = 2$, a natural tree to build is a binary tree, i.e., each node has at most two children. Given the binary tree has a height $h = \lfloor log_2(n) \rfloor$, the number of leaf nodes is in the range $[2^{h-1} + 1; 2^h]$. Assuming $n = 2^{h+1} - 1$ with h a given height, the binary tree is *full* as every node at level $h-1$ has two children. In other words, all leaf nodes are at level h. With a full binary tree, the number of nodes that do not contribute anything to the system is therefore 2^h (roughly half of the nodes), that is, the total upload bandwidth contributed to the system is reduced to $(n - 2^h)(f \cdot s)$. With such a reduced total upload bandwidth in the system, the maximum stream rate that can be broadcast gets limited to $(n - 2^h)(f \cdot s)/(n - 1)$, that is, the sum of upload bandwidth of the nodes that can contribute divided by the number of nodes that need to receive the stream, excluding the source. In essence, by not giving a chance to the leaf nodes to contribute, the maximum stream rate gets very much limited.

Focusing on the upload bandwidth of nodes, it seems natural to build a tree such that all nodes serve the most number f_i of children, i.e., $f_i = \lfloor u_{i,up}/s \rfloor$. This way, the tree grows in breadth and results in a smaller number of hops between the source and the leaves, resulting in a reduced stream lag on average. However, increasing the breadth of the tree, and thus decreasing its height, also naturally increases the number of leaves in the tree which in turn reduces the maximum stream rate that can be broadcast.

The limitations in terms of stream rate and resilience to failures have led system designers to consider different overlays, one of them being to transform a single tree into multiple trees, as described in the following.

Multitree-based Dissemination

By creating multiple dissemination trees on top of a DHT [CDK$^+$03] or on top of a mesh [SGMZ04, VYF06], a node has a very low probability of being leaf in each of the multiple trees, thus it will contribute to some of them. In essence, consider that the stream is split into multiple *stripes* (or substreams) and that each stripe is sent from the source to a different set of children, possibly only one child per set. Each of the children acts as a source for its assigned stripe, resulting thus in different trees. In other words, a node receiving a given stripe from the source acts as a root for this subtree but not for the other stripes for which it can act as leaf or intermediary node. The use of different dissemination trees results in a much better overall utilization of resources since all nodes can effectively contribute upload bandwidth to the system. Following this approach, the maximum stream rate s is therefore set back to $\sum_i u_{i,up}/(n-1)$, put aside the overhead of distributing multiple substreams instead of a single one.

The splitting of the stream in multiple stripes is motivated by two facts. First, the use of different dissemination trees so that nodes that were leaf nodes in a single tree can now also contribute resources, namely upload bandwidth, allowing thus to broadcast a higher stream rate. And second, the multiple stripes can represent different descriptions of the stream in such a way that delivering only a subset of descriptions, namely stripes, roughly represents a proportional loss in terms of quality when delivering the stream. This technique is named *multiple description coding* (MDC) and was introduced in [GKAV98, PR99, Goy01].

A condition for this approach to function, for both resilience to churn and efficient use of bandwidth, is to ensure that the different subtrees are not built according to the exact same semantics, i.e., each node should be placed at different heights in each of the trees it belongs to. Otherwise, the multiple trees would share too many similarities by relying on the same nodes at the same height and therefore not use the upload bandwidth of the same leaves and finally suffer just as much as a single dissemination tree. In the particular case of SplitStream [CDK$^+$03], the original stream is split into 16 stripes resulting in 16 different dissemination trees. In the case of 25% of catastrophic failures, the remaining nodes receive an average of 6 stripes for 30 seconds after the failure and need up to 3 minutes, in the worst case, to recover the full stream, i.e., the 16 stripes. The underlying structure is still connected after the failure and it is thanks to Pastry [RD01] and Scribe [CDKR02] that SplitStream recovers, i.e., repairs the multiple trees for serving the surviving nodes. In Chunkyspread [VYF06], with 10% of the nodes failing, the surviving nodes recover the full stream (the 16 stripes) between 4 and 12 seconds, respectively 15 stripes out of 16 between 2 and 10 seconds and 13 stripes out of 16 between 0 and 8 seconds.

A side effect of multitree-based dissemination affects the buffering delay, thus stream lag. Within a single tree, a leaf can experience a very low buffering delay (dependent on the communication delay between the leaf and its parents) but a rather large stream lag (dependent on the height of the tree). With a multiple tree approach, on the other hand, a node that is very close to the source for a given stripe (at a low level of the tree) should be at a higher level (possibly a leaf) for another tree. If the node wants to view a stream of high quality (a majority of stripes, if not all), it has to wait until it receives data from multiple trees in order to start viewing the stream. Since nodes should be present as often at lower levels as in higher levels in different dissemination trees, they roughly all experience the same buffering delay, thus stream lag. In other words, multiple dissemination trees homogenize the buffering delay thus the stream lag of participating nodes.

To summarize, multiple-tree dissemination improves the single tree approach by enabling all nodes to contribute, including the leaf nodes, under the conditions that *(i)* nodes have an upload bandwidth at least larger than the stream rate of a single stripe and *(ii)* the sum of all nodes' upload bandwidth is larger than $s(n-1)$, that is, there is enough bandwidth to serve all nodes except the source.

Even though some approaches seem to provide a relative simplicity in building and maintaining the multiple trees [VYF06], the impact of churn or catastrophic failures, be it for complex tree(s) reparation [BRP$^+$05] or the fact that MDC is not democratized in practice [LRLZ08], has led researchers not only to try building dissemination trees on top of unstructured overlays in different ways, but to use those unstructured overlays directly for the dissemination itself for superior performance [MR07].

2.3.2 Unstructured Overlays

Unstructured overlays, also named random overlays because they usually represent a random graph [Bol01], roughly provide each node's view with a set of random nodes. There is no hierarchy between nodes and because of the presence of cycles, a node can be both ancestor of a node through a node and descendant of the same node through another node. The resulting overlay is a mesh that is commonly static and dynamic in case of gossip protocols.

We start by reviewing mesh-based systems and hybrid ones where the mesh is used both for dissemination and for creating a single tree or multiple-trees and finally give a flavor of gossip systems.

Mesh-based Dissemination

Whereas splitting the stream is explicit in order to create multiple dissemination trees, the idea underlying mesh-based dissemination is that the stream can potentially follow a different dissemination path for every single chunk of it, in an implicit manner. In essence, each node advertises the content it has to its view and gets pulled by its neighbors for the chunks in which they are interested. Since nodes will not request a chunk that they already have, the dissemination path for each chunk results indeed a single tree. By picking a large enough random view set, we know it is possible to construct a random graph that is connected with high probability [KMG03], thus the probability that each chunk flows from the source to each node in the mesh is very high. The dissemination pattern in a mesh is *epidemic* in the sense that *infected* nodes infect their neighbors which in turn infect theirs and so on, until the whole population of nodes is infected. These approaches have proven to be analytically optimal or close to optimal in terms of bandwidth and delay [ZZSY07, BMM$^+$08, GLL08, LGL08, PM08, OKJ09]. In such systems, the views are assigned to nodes at random and it is up to the nodes to decide *what* data to serve and *whom* to serve in their views. In other words, in such systems the problems to solve are *(i)* optimal node selection (also named peer selection) and *(ii)* optimal packet selection (also named packet scheduling) in order to maximize the efficiency of the dissemination, i.e., minimize protocol overhead, buffering delay and stream lag and maximize stream quality towards all nodes.

When a node delivers a chunk, that is, it pulled it from one of the nodes that advertised it, the node starts to advertise it to its view. The node therefore needs to send messages to its view for signaling that a given chunk is now available for download. The signaling can of course be periodic so that a single message can be used to advertise multiple chunks but this signaling represents an overhead that was not present in tree or multiple tree dissemination, where the major cost, in terms of networking overhead, was in the creation, maintenance and reparation of the structure. This signaling cost, however, is acceptable because negligible with respect to the stream rate. This generic approach is common to all mesh-based systems, be it for streaming e.g., [PKT$^+$05, ZLLY05, PPKB07, ZZSY07, MR09b] or file sharing with the very popular BitTorrent [Coh03] on which variations were proposed for streaming, focusing on VoD, e.g., [DLHC05].

Since views of nodes do not change (i.e., static overlay), nodes can keep TCP connections open with each other and thus rely on its retransmission mechanisms for recovering dropped packets, this advantage is recognized for congestion control as the streaming itself can be tuned to be TCP-friendly with respect to other bandwidth-consuming applications that are concurrently executed [WH01,MO07] but is also recognized as a drawback for time-critical applications like live streaming [WHZ$^+$00,WHZ$^+$01,ATW02]. The work in [ZZSY07] tunes TCP's sending buffer so that it drops packets when congested, definitely pushing for unreliable transmission such as UDP.

The fact that views are static also means that there is a relatively high probability that some nodes get a relatively good view, i.e., nodes with low delay and high bandwidth, whereas some other nodes might have a relatively bad view, and this, for the whole lifetime of those nodes in the system. As opposed to multiple-tree approaches, some nodes could thus have a very low stream lag (e.g., when served by nodes relatively close to the source) whereas some others might suffer a relatively high stream lag, and thus might not be able to contribute as much as they could. Optimal bandwidth utilization and stream lag mainly depend on node selection and chunk scheduling but also on the random overlay that was created, that is, how nodes were distributed in the mesh. In addition, mesh-based systems, even though they are very well connected and suffer less from disconnection than trees or multiple-trees, they still need to repair in case of catastrophic failures or churn. To illustrate this, the impact of the view size is evaluated in [LGL08] when given percentage of nodes leave the system. The remaining nodes are still well connected but suffer from 5% to 15% of missing chunks with 10% to 50% of leaving nodes assuming large views (each node is connected to 18% of the system) resulting in large signaling overhead. Some mesh-based systems (e.g., [ZLLY05]) rely on gossip-based membership protocols to provide them with a connected random graph and there exists protocols dedicated to repairing mesh-based overlays [MR09a], thus answering a real need.

In the following, we first review hybrid dissemination patterns that mix trees and mesh to reduce the signaling overhead when the system is considered healthy, i.e., in the absence of churn. And secondly, we review existing proactive repair approaches for doing high-bandwidth content dissemination, that is, disseminating chunks in a dynamic mesh overlay by focusing on the gossip paradigm.

2.3.3 Hybrid Dissemination

Bullet [KRAV03] is an hybrid high-bandwidth dissemination system in the sense that the main data dissemination is done with a tree and the remaining data is spread on top of a mesh. The data is split into multiple blocks from which only a subset is pushed from parents to their children, i.e., tree dissemination. The remaining blocks are then pulled from other nodes that advertise them in a random manner, i.e., with the help of a mesh overlay.

Mesh-based systems can also be dynamically tuned to create multiple-trees at runtime, when the system is considered healthy [ZZSY07, LXQ$^+$08, CJW09]. When nodes consider that they do not need to request packets from different nodes in their views because their view does not need to repair, they can *subscribe* to some of their neighbors asking them not only for chunks they advertise, but for a whole substream, thus reducing the signaling overhead. In other words, the stream is split into chunks that themselves belong to different substreams of the original stream. The views of nodes thus gradually result into a multiple-tree overlay on top of which the different substreams are spread. The advantage of such a solution is that the signaling overhead is reduced but for the streaming experience, it does not mean the quality is superior or the delay smaller. In fact, the user could fast forward the stream to be more live since the average dissemination delay becomes shorter in the tree than in the mesh dissemination, but in practice, the advantage is only to have the buffer fuller possibly resulting in more spare time to repair the mesh in case of catastrophic failures or churn. This aspect still needs to be evaluated, otherwise the use of the trees on top of the mesh brings a lot of complexity for no improvement.

Finally, [WJJ05] addresses the problem of building an optimized mesh in terms of network proximity and latency, in the presence of *guarded* nodes, i.e., nodes that are behind a NAT or firewall. This work led to mixing application level multicast with IP multicast whenever possible [ZWJ+06]. The core of this research is now commercially used in [Zat] and further described in [CJW09]. In order to compensate for the limited contribution of nodes in the system, there exists super peers (deployed on PlanetLab at the time of prototyping), named *repeater nodes* that act as bandwidth provider, i.e., they are dedicated servers that contribute data to other nodes. Complementarily, it is for instance known that the dissemination protocol of PPLive [PPL] substantially relies on a set of super peers and thus does not represent a purely decentralized solution [HLL+07].

Gossip-based Dissemination

The gossip paradigm was first introduced in computer science when tackling the problem of replicated database maintenance [DGH+87]. In essence, the main difference between a mesh-based dissemination system and a gossip-based dissemination system is that the underlying random graph overlay is static in a mesh and dynamic in gossip. In a mesh, a node is given a view of fixed size that will not change during its lifetime in the system as long as the node considers itself well connected, meaning its view size is above a certain threshold. In gossip, the view of a node periodically changes, resulting in a random graph that updates its edges periodically. In order to better distinguish between mesh and gossip systems, we usually call the dynamic view of a node in gossip the *view* the node has on the system. This perfectly represents the dynamic nature of the view since the view that a node has on the whole system periodically changes. The fact that edges are dynamically refreshed means that nodes having crashed or left the system will eventually be detected and not be present in any node's view whereas arriving nodes will populate nodes' view very fast.

The membership maintenance problem is exactly to provide each node with a random subset of other nodes (i.e., view) to communicate with while enabling arriving nodes to be present as fast as possible while removing left or crashed nodes in the views by being lightweight and scalable. This problem is usually solved using a random peer sampling service, itself using gossip protocols [GMS04, KS04, ADH05, VGvS05, JVG+07, BGK+09].

Gossip has been widely used for disseminating *small updates* [DGH+87, LOM94, BHO+99, EGH+03, KMG03]. The data to disseminate is so small that the bandwidth constraints of the nodes do not constitute a bottleneck (e.g., [EGH+03,KMG03,BHO+99]). More recently, some work have applied the gossip paradigm to high-bandwidth content dissemination, but in these particular cases, there was no bandwidth constraint on the nodes [DXL+06,LCW+06, LCM+08].

CREW [DXL+06] is the first gossip protocol focusing on high-bandwidth dissemination, file sharing in particular. BAR Gossip [LCW+06] and Flight-Path [LCM+08] tackle the issue of live streaming in the presence of byzantine nodes. The focus is not on providing the most efficient protocol in terms of dissemination but on tolerating malicious nodes that can attack the protocol. There is thus a need for research in the context of gossip, to devise a protocol that targets efficiency and constitutes an alternative if not an improvement over multiple-trees or static meshes.

We explain the gossip paradigm, its application to high-bandwidth content dissemination and the resulting challenges such as parameterization, heterogeneity adaptiveness and freerider detection in more details in the following chapter.

Gossip is charming! History is merely gossip. . . But scandal is gossip made tedious by morality.

Oscar Wilde

3 High-bandwidth Content Dissemination with Gossip

In high-content bandwidth dissemination, the trend has definitely moved from traditional client-server approaches to P2P approaches. With the absence of IP multicast at the network layer, application level multicast has emerged as a real alternative. Starting with natural and intuitive (single) tree structures, the trend has moved to multiple-trees and then mesh-based systems. Two orthogonal axes exist in *(i)* reintroducing multiple-trees on top of meshes (which could be considered as a step back) or *(ii)* finding the best strategies to optimize dissemination into meshes.

We believe that for large-scale systems where nodes can arrive and disappear without notice, the next step is to devise algorithms for high-content bandwidth dissemination following the gossip paradigm, with a proactive attitude towards churn (as opposed to reactively repairing the overlay) and simplicity in mind for ease of implementation and deployment [Ham07, Zho09].

This chapter presents the gossip paradigm along with a gossip protocol for high-bandwidth content dissemination that is at the core of existing gossip and mesh approaches. It is the building block that we *(i)* evaluate and parameterize with bandwidth constraints (Chapter 4), *(ii)* improve with simple mechanisms for efficient dissemination (Chapter 5), *(iii)* adapt to bandwidth heterogeneity so that gossip can act as a real alternative to existing systems (Chapter 6), and finally *(iv)* complement with a protocol for detecting and expelling nodes that do not provide their fair share of work (Chapter 7).

3.1 The Gossip Paradigm

The gossip paradigm was initially inspired by mathematical models that investigate everyday life phenomena: rumor mongering and epidemics. During the last century, mathematicians developed models to predict the rate of diseases spread, namely epidemics, using differential equations. In addition, researchers have developed discrete mathematic models to predict what we already know: rumors spread fast. It was thus natural to harness these models in order to design distributed systems that mimic the basic behavior of such fast spreading everyday life paradigms.

To begin with, gossiping and broadcasting were first described as two different information dissemination problems for a group of individuals connected by a communication network. More concretely, as defined in [HHL88], in the gossiping problem, every individual in the network initially knows a unique item of information and needs to communicate it to everyone else in the network, whereas the broadcasting problem is defined as the case in which one individual has an item of information which needs to be communicated to every other individual. The authors present solutions for both problems, producing a deterministic sequence of unordered pairs of communication partners. Each pair represents a *phone call* made between a pair of individuals, such that, during each call, the two people involved exchange all the information they know at that time. At the end of the sequence of calls, everybody knows all the information that had to be spread. The survey focuses on the number of calls among n people over arbitrary network topologies and variants of the problems.

A new variety of gossip-based algorithms have evolved as communication patterns for designing simple, scalable, and efficient communication protocols in large distributed systems. The first work proposing such gossip-based algorithms is [DGH$^+$87], proposing a family of gossip-based algorithms for maintaining replicated database systems. Since then, gossip-based algorithms have been proposed to solve central problems in numerous distributed systems deployed over wide range of networks. These problems include, for instance, replicated database maintenance [DGH$^+$87], Usenet news distribution [LOM94], ad-hoc routing [HHL06], distributed failure detection [vRMH98, DHJ$^+$07], network management [vR00], lightweight broadcast [EGH$^+$03], peer-to-peer membership maintenance [GKM03, VGvS05, JVG$^+$07], aggregation in large scale networks [JMB05] or in sensor networks [DSW06], building of overlay structure [GHH$^+$06], and topology management [JMB09].

The Gossip Overlay

Dissemination in gossip commonly follows an epidemic pattern, similarly to mesh-based dissemination, but for different reasons. In meshes, it is because of the random choices of neighbors in the overlay itself that the dissemination follows a random path. In gossip, it is the communication strategy that commonly follows a random fashion and the overlay simply needs to guarantee that a node is capable of picking nodes at random from the set of all nodes. To achieve this, nodes can either have knowledge of the whole system and randomly pick communication partners in this set or have partial knowledge of the whole system and make sure that this partial knowledge changes dynamically so that picking communication partners with this partial knowledge is equivalent to picking partners at random from the set of all nodes. The local views that each node maintains are said to be *global* if they contain all the possible communication partners of the node, and *partial* otherwise. Note that as defined in Section 2.2 a view contains only nodes that a given node can physically contact.

Global Views Global views have been considered in previous theoretical works in which a gossip-based approach relied on the assumption that each node locally knows every other node in the system. Consider, for example, the general structure of a gossip-based protocol discussed in the seminal paper of Demers et al. [DGH$^+$87]. The system consists of a set V of n nodes interconnected by a complete graph (clique). In other words, each node has a global knowledge of the system, with a view that contains every other node in the system. The choice of communication partners, usually randomized, can thus rely on this global view of the system. In this case, the view maintained by a node u is equal to the whole network at all times, $view_u(t) = V(t)$, meaning that the nature of the network assumed can only be a complete graph as each node u can potentially contact any node in the view, thus network.

Partial Views Providing each node with a global view is unrealistic in a large distributed system for scalability reasons. First the data structure for storing the view should not grow linearly with the system size and second, maintaining such information in the presence of churn incurs considerable communication costs. The resulting problem is a need for protocols that maintain partial views, keeping given desired properties of the overlay network (e.g., connectivity [ADH05, EGH$^+$03]). Interestingly, the problem of overlay maintenance can itself be solved by gossip-based algorithms (e.g., [GMS04, KS04, ADH05, VGvS05, JVG$^+$07, BGK$^+$09]).

Peer Sampling Service

A peer sampling service provides nodes with samples of the set $V(t)$, which have some probabilistic guarantees (typically, is a uniformly chosen random sample). The sampling has to consider the churn rate experienced by the system, trying to prevent including inactive nodes in the samples. The peer sampling service assumes an underlying complete communication graph such that $view_u(t) \subseteq W_u(t) = V(t)$. We assume that each node in the system is reachable from each other node or there exists means to circumvent firewalls and NAT, e.g., [WJJ05, KPQS09].

3.1.1 Gossip-Based Algorithms

In this section we present the structure of a generic gossip-based algorithm and identify its most significant parameters.

Structure of Gossip-Based Algorithms

Gossip-based algorithms have a very simple and regular structure. The code of the gossip-based algorithm is executed by each node in rounds. Every round r, a node performs *(i)* a *communication phase*, followed by *(ii)* a *processing phase*.

- **Communication Phase** In the communication phase of a round, a network node v chooses a subset of communication partners (called *neighbors*) from its local view, and exchanges with them information it holds. The message sent by the node is synthesized from the current state of the node, and possibly includes information from several previous exchanges.
- **Processing Phase** After the communication phase, a network node applies a state transition function to its current state to obtain the new state. The transition depends on the current state and the information obtained form the set of neighbors that have been contacted. In the case of information spread or broadcast, the state transition function defines *(i)* what information should be delivered (i.e., if the node received new information) and *(ii)* what information has to be gossiped in the next round(s).

We illustrate a common structure of a generic gossip-based algorithm in Algorithm 3.1. The communication phase begins in line 2, as the neighbors set is filled by a select method. This method implements the *communication strategy* of the algorithm, defined by the underlying system parameters and adjusted by the *algorithm parameters*. For each of the chosen communication partners, a communication channel is opened and data can be exchanged (line 4 in Algorithm 3.1). The gossip message is exchanged depending on the *transmission model* chosen for the algorithm (lines 6, 9, 16 and 20 in Algorithm 3.1), and after the information is received (lines 9 and 16), the communication phase is finished, and the processing phase starts. The update method implements the state transition function that will effectively solve the problem, delivers the information and chooses what information to gossip in the following round(s) (if applicable) and finally does some maintenance tasks (e.g., *buffer management*).

Algorithm 3.1 Generic gossip-based algorithm pseudocode. *The booleans push and pull are true in case of a pushpull transmission model.*

```
Initialization:
  gossipPeriod := $T_g$ time units
  start(GossipTimer)
upon (GossipTimer $\bmod$ gossipPeriod) = 0 at node $u_i$ do
  neighbors := selectNodes($f$) communication partners from $view_{u_i}$
  for all $p \in$ neighbors do
    communicateWith($p$)  {A communication channel is open between $u_i$ and p, data can now be exchanged}
    if (push) then
      send gossip message to $p$                    {$u_i$ is pushing data to $p$}
    if (pull) then
      receive gossip message from p
      update local state
upon communicateWith() from $u_j$ at node $u_i$ do
  if (push) then
    receive gossip message from $u_j$               {$u_j$ is pushing data to $v_i$}
    update local state
  if (pull) then
    send gossip message to $u_j$                    {$u_j$ is pulling data from $u_i$}
```

Algorithm Parameters

The above basic pseudocode of a gossip-based algorithm has to be instantiated depending on the problem that has to be solved and the underlying available system. Among others, this instantiation depends on the following basic parameters.

Transmission Model As can be observed, in Algorithm 3.1, there are two flags, *push* and *pull*, that strongly characterize the behavior of the algorithm. They determine if the communication of the node with the neighbors only transfers information from the node to the neighbor, only transfers information from the neighbor to the node, or transfers information in both directions. The first two cases are one-way transmission. In the first, we say that information is *pushed* from the node to the neighbor, while in the second we say that the information is *pulled* by the node from the neighbor. In the general case of two-way transmission, where the two nodes exchange their information, the transmission model is named *pushpull.*

Communication Strategy and Fanout The communication strategy defines how a node u chooses the subset of communication partners, i.e., its *neighbors*, for the current round. The first decision the algorithm designer has to make is the size of the neighbors' set, f, known as the *fanout*. The set of neighbors is chosen from the $view_u(t_r)$ (t_r is the time at which round r is executed), thus $neighbors \subseteq view_u(t_r) \subseteq W_u(t_r) \subseteq V(t_r)$. The way a set of neighbors is chosen is usually random. Following the communication strategies defined, it is indeed sometimes the underlying system that dictates or influences the choice of neighbor(s) with whom to communicate by reflecting the nature of the underlying network (e.g., network-driven communication strategy [LM99, KKD04]) or to reflect a given arbitrary topology (e.g., application-driven topology [JMB09]).

Buffer Management Inherently to all gossip-based algorithms, duplication of messages can happen. Furthermore, it is common that every message carries several units of information, typically named *events*. Hence, it is highly possible that a node receives information (events) that it already had, which is one of the reasons gossip-based algorithms are considered fault-tolerant. With random communication, there is a fair chance that the same neighbor is chosen more than once. Moreover, even without repeating partners, it is possible that two nodes p and q, which both communicated with r in the past, can now exchange r's information.

For scalability reasons (e.g., overall number of messages exchanged or the size of these messages) an algorithm usually specifies that a node should not forward all events forever and decide to stop gossiping this or that particular event (i.e., by removing it from the gossip message) at some point. This *buffer management* problem is exposed in [EGKM04] and solutions are proposed in, e.g., [EGH$^+$03, RHP$^+$03].

Message Size Within the restrictions imposed by the underlying system, the algorithm has to decide how to forward the events that it has in its buffer. It may decide that it will forward all events forever, which means that unless events are purged from the memory the message size will grow arbitrarily large. Another option is to decide that only a fixed number of events will be forwarded in each message, which implies that messages will have a fixed size, but open the question of how to select the set of events to transmit (e.g., age-based purging [EGH$^+$03]). Finally, it can decide to forward each event a maximum number of times, which implies that the message size depends on the number of events to gossip in the following round(s). In epidemic terminology, each different event represents an infectious disease and sending or receiving messages translates to infecting and being infected by other nodes. Once a node gets infected by a disease, it can basically *(i)* decide it can only be infectious by a fixed number of diseases and thus instantly heal other diseases (i.e., having a maximum message size), *(ii)* be infectious forever (i.e., forwarding each event an infinite number of times), or *(iii)* be infectious during a given time, named *contagion period* (i.e., forwarding the same event a given number of times since reception of it), a special case is named *infect-and-die* when the contagion period is 1 for a given disease.

History Buffer Size A related issue is to manage the history of delivered events. The history of delivered events is maintained on each node in order to decide if a received information is new (i.e., has to be delivered in case of information spread), or has already been received in the past (i.e., the node received a duplicate). The management of the delivered buffer must be scalable as more and more events are received (and delivered) over time. Modeling and analysis of the history buffer can be found in [Kol03].

3.2 Gossip for High-bandwidth Content Dissemination

Among its various applications, gossip is also very appealing for bandwidth-intensive applications such as file-sharing, or video streaming. In this case, the source splits the data to share into chunks that constitute the events to be transmitted. Then it begins dissemination using a gossip-based protocol. The problem is that nodes generally cannot afford gossiping these chunks themselves due to their large size. Gossip creates many duplicates and nodes usually have limited bandwidth: sending two copies of the same chunk to the same node would thus waste too many resources.

As a way to address this problem, the work in [CKS09] proposes a gossip algorithm for file sharing using fountain codes. The source splits the file to share into k chunks that are gossip-pushed to $n/2$ nodes, using an experimental Time to live (TTL). It additionally and continuously encodes the file and sends new encoded chunks with the same TTL. Once a node (i.e., including the source) has received enough chunks to decode the file (i.e., k random chunks) it decodes it, re-encodes the content and starts to gossip newly encoded chunks itself, preferring nodes that are close to having k chunks, in order to increase the number of coding sources in the system. The gossips stop once every node has received at least k chunks.

The use of fountain codes allows the proposal in [CKS09] to exploit the first exponential-growth phase of gossip which has been shown to be more efficient, due to the presence of fewer duplicates, than the second *shrinking phase*. However, even in the first phase, a node may still receive the same chunk multiple times, leading to inefficient bandwidth utilization. Moreover, the protocol requires that some nodes, i.e., those that completed the first phase, contribute more than others. In some cases, this may lead to freeriding behavior.

3.2.1 Three-phase Gossip

To address the problem of bandwidth utilization more effectively, an appealing solution is to use a three-phase gossip protocol [DXL$^+$06, LCW$^+$06, LCM$^+$08] inspired from mesh-based protocols [PKT$^+$05, ZLLY05, ZZSY07] as exposed in Algorithm 3.2. In short, the first phase gossips content location by sending proposals for chunk ids to fanout nodes. The generic gossip in Algorithm 3.1 is instantiated with *message* being a proposal (or propose message), the unit of information to gossip (abstracted to the notion of event) is a chunk identifier and the gossip is configured to *push* only. A propose message thus contains multiple chunk identifiers (chunk ids). This first gossip phase serves to advertise to fanout nodes that the sender has the corresponding chunks. The second phase consists in requesting the chunks needed and the third is a reply with the chunk itself, in other words the payload. Using these three phases, gossip is creating many duplicates on small size propose messages whereas the payload is received only once. This makes the model very appealing in the context of high-bandwidth content dissemination such as streaming.

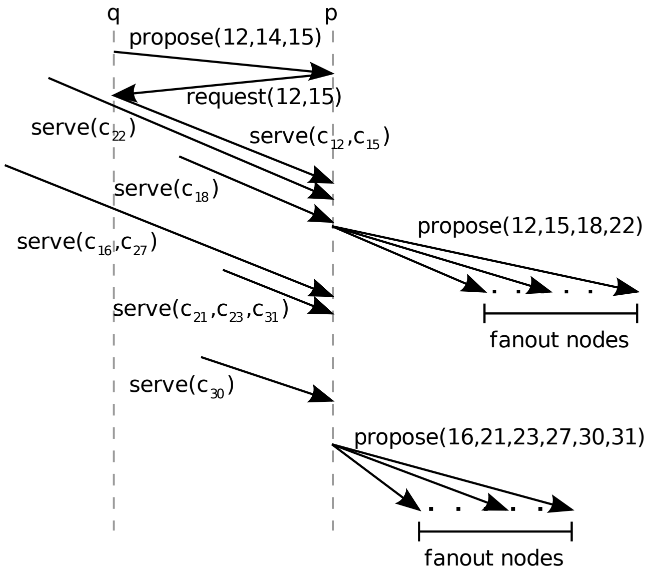


Figure 3.1: Three-phase gossip protocol with an infect-and-die behavior.

The three phases are depicted in Figure 3.1 and are as follows:

- *Propose phase.* Periodically, i.e., every gossip period T_g, each node picks a set of f (fanout) other nodes uniformly at random. This is illustrated in Figure 3.1, where node p proposes chunks $12, 15, 18$ and 22 during the first displayed gossip period, and chunks $16, 22, 23, 27, 30$ and 32 in the subsequent one.
- *Request phase.* Upon receipt of a proposal for a set of chunk identifiers,

Algorithm 3.2 Three-phase gossip protocol.

```
Initialization:
  $f := \ln(n) + c$                                   {$n$ is the system size and $c$ a constant}
  chunksToPropose := chunksDelivered := requestedChunks := $\varnothing$
  start(GossipTimer)
Phase 1 – Gossip chunk ids
procedure publish($c$) is
  deliverChunk($c$)
  gossip($\{c.id\}$)
upon (GossipTimer mod gossipPeriod) = 0 do
  gossip(chunksToPropose)
  chunksToPropose := $\varnothing$                                            {Infect-and-die}
Phase 2 – Request chunks
upon receive [PROPOSE, chunksProposed] do
  wantedChunks := $\varnothing$
  for all $c.id \in$ chunksProposed do
    if ($c.id \notin$ requestedChunks) then
      wantedChunks := wantedChunks $\cup$ $c.id$
  requestedChunks := requestedChunks $\cup$ wantedChunks
  reply [REQUEST, wantedChunks]
Phase 3 – Push payload
upon receive [REQUEST, wantedChunks] do
  askedChunks := $\varnothing$
  for all $c.id \in$ wantedChunks do
    askedChunks := askedChunks $\cup$ getChunk(c.id)
  reply [SERVE, askedChunks]
upon receive [SERVE, chunks] do
  for all $c \in$ chunks do
    if ($c \notin$ chunksDelivered) then
      chunksToPropose := chunksToPropose $\cup$ $c.id$
      deliverChunk($c$)
Miscellaneous
function selectNodes($f$) returns set of nodes is
  return $f$ uniformly random chosen nodes in the set of all nodes
function getChunk(chunk id) returns chunk is
  return the chunk corresponding to the id
procedure deliverChunk($c$) is
  deliveredChunks := deliveredChunks $\cup$ $c$
  deliver($c$)
function getFanout() returns Integer is
  return $f$
procedure gossip(chunk ids) is
  communicationPartners := selectNodes(getFanout())
  for all p $\in$ communicationPartners do
    send(p) [PROPOSE, chunk ids]
```

a node determines the subset of chunks it needs, and requests them from the sender. Clearly, the needed chunks are those that the node has not yet received. In Figure 3.1, node p requests chunks 12 and 15 from q.

- *Serving phase.* When a proposing node receives a request message, it replies with the corresponding chunks, that is by sending their actual payloads. Nodes only serve chunks that they previously proposed. In Figure 3.1, node q serves p with chunks c_{12} and c_{15}.

Infect-and-die Gossip In the proposed three-phase algorithm, we opted for a number of contagion period of 1 (line 7 in Algorithm 3.2). In other words, the gossip protocol follows an infect-and-die model, as opposed to [LCW$^+$06, LCM$^+$08] where the number of contagion periods is larger, using a sliding window of chunk ids (e.g., [ZLLY05, ZZSY07]).

Theoretical results [KMG03] show that in an infect-and-die model, the fanout has to be chosen as $ln(n) + c$ where n is the system size and c a constant defining the probability of the gossip protocol to result in a connected graph as $\exp(-\exp(-c))$. In other words, f can be chosen such that all events are gossiped to all nodes *with high probability*. It is rather intuitive that infecting f nodes in one round and infecting 1 node in f rounds should be equivalent (considering the nodes chosen in the fanout consecutive rounds are different) in terms of reliability. To roughly achieve the same dissemination speed, the gossip period in the second case should therefore be divided by the fanout. Infecting fanout nodes in a time T_g is roughly equivalent to infecting fanout times 1 node every T_g/f time.

Technically speaking and put aside the recognized disadvantages of TCP in time-critical applications such as live streaming [WHZ$^+$00, WHZ$^+$01, ATW02], a low fanout with a relative large gossip period, thus possibly exchanging a larger set of data between the sender and the fanout nodes, is an advantage when using TCP connections since the cost of connection establishment is amortized by the fact that a relatively large data is sent and thus there is no need for explicit retransmission, but a lower fanout and a larger period definitely infects less nodes per period of time. In order to compare with an infect-and-die model, lowering the fanout also means reducing the gossip period (by a factor fanout) and in practice with an infect-and-die model, we experienced optimal performance with gossip periods between 200 ms and 500 ms and fanout in the order of $\ln(n)$ (Chapter 4). It is thus hardly possible to establish TCP connections with one or multiple nodes every $500/ln(n)$ ms especially when considering that the time to exchange data should be larger than the cost of connection establishment to be considered profitable.

In addition, assuming that increasing the number of contagion periods is similar to increasing the fanout, Chapter 4 gives evidence that increasing the fanout should be done with caution, indicating also that the number of contagion periods (or sliding window size) of data to communicate should also have an optimal range.

We therefore concentrate on the infect-and-die model throughout this work and assume unreliable communication between nodes, using UDP.

3.3 Live Streaming with Gossip

In order to disseminate real audiovisual content, we designed a simple system composed of the two following roles: being the source and being regular nodes. On the source node, we execute a movie player (e.g., VLC [Vid]) which produces a stream of a chosen quality. This stream feeds the source node exactly as if a camera was filming a live event. VLC produces a MPEG transport stream with each chunk being of size 1316 bytes. This chunk perfectly fits in a single UDP packet as the common maximum transmission units (MTU) are almost always above 1500 bytes. Whenever a new chunk is produced by VLC, the source assigns a sequence number (the chunk id) and a timestamp to it, and gossips a propose message to fanout nodes with this chunk id. When a node receives such a propose message from the source, it requests the source for the proposed chunk and the source replies with the chunk in a brief message (along with the timestamp of the creation of the chunk). By gossiping to fanout nodes at every production of a chunk, the source creates a different dissemination path for each of the chunks. Even though the source could also be acting as a regular node watching the stream, it is excluded from all experiments so that its extremely good performance (negligible buffering delay and stream lag and perfect quality) do not bias the evaluations.

Nodes execute a movie player for outputting the stream. When receiving chunks from the source or from other nodes, a node buffers the received data until it decides to start forwarding the data to the movie player. When it starts forwarding data, it calculates both how live the stream is (i.e., the stream lag) with the help of the timestamp of the source (the time on nodes is synchronized) and the duration between reception of the very first chunk and the forwarding has started (i.e., the buffering delay).

Every gossip period, nodes gossip the ids of the chunks they received since the last gossip period in a propose message to fanout nodes chosen at random. When receiving a propose message containing chunk ids, the node requests the ones it did not receive nor request yet. Doing this, the node is ensured not to receive duplicates of chunks and thus also not to impose a higher load on proposing nodes than needed. Since we rely on UDP for all three types of messages and that the gossip dissemination of chunk ids reaches all nodes with high probability only, it is very possible for a node *(i)* not to receive proposals for a given chunk, *(ii)* that its request is lost, or *(iii)* that the brief serving the requested chunk is lost.

To overcome those effects we complement the three-phase gossip with retransmission, as exposed in Algorithm 3.3, and erasure coding, i.e., Forward Error Correction (FEC). The source groups packets in windows of $k + c$ chunks, representing k source chunks and c encoded chunks, meaning that a node receiving at least k (random) chunks in a group is able to decode the whole group. Nodes

Algorithm 3.3 Three-phase gossip protocol with retransmission.

```
Initialization:
  $f := \ln(n) + c$
  chunksToPropose := chunksDelivered := requestedChunks := $\varnothing$
  start(GossipTimer)
Phase 1 – Gossip chunk ids
procedure publish($c$) is
  deliverChunk($c$)
  gossip($\{c.id\}$)
upon (GossipTimer mod gossipPeriod) = 0 do
  gossip(chunksToPropose)
  chunksToPropose := $\varnothing$                                   {Infect-and-die}
Phase 2 – Request chunks
upon receive [PROPOSE, chunksProposed] do
  wantedChunks := $\varnothing$
  for all $c.id \in$ chunksProposed do
    if (($c.id \notin$ requestedChunks) or (isBeingRetransmitted($c.id$)) then
      wantedChunks := wantedChunks $\cup$ $c.id$
  requestedChunks := requestedChunks $\cup$ wantedChunks
  reply [REQUEST, wantedChunks]
  if ($c$ requested less than $r$ times) then
    start(RetTimer(chunksProposed))                   {Schedule retransmission}
Phase 3 – Push payload
upon receive [REQUEST, wantedChunks] do
  askedChunks := $\varnothing$
  for all $c.id \in$ wantedChunks do
    askedChunks := askedChunks $\cup$ getChunk(c.id)
  reply [SERVE, askedChunks]
upon receive [SERVE, chunks] do
  for all $c \in$ chunks do
    if ($c \notin$ chunksDelivered) then
      chunksToPropose := chunksToPropose $\cup$ $c.id$
      deliverChunk($c$)
  cancel(RetTimer(chunks))             {Cancel retransmission of delivered chunks}
Retransmission
upon (RetTimer(chunksProposed) mod retPeriod) = 0 do
  receive [PROPOSE, chunksProposed]   {Re-request proposing node for missing chunk}
function isBeingRetransmitted(chunk id) returns boolean is
  return true if a timer is scheduled with id, false otherwise
Miscellaneous
function selectNodes($f$) returns set of nodes is
  return $f$ uniformly random chosen nodes in the set of all nodes
function getChunk(chunk id) returns chunk is
  return the chunk corresponding to the id
procedure deliverChunk($c$) is
  deliveredChunks := deliveredChunks $\cup$ $c$
  deliver($c$)
procedure gossip(chunk ids) is
  communicationPartners := selectNodes($f$)
  for all p $\in$ communicationPartners do
    send(p) [PROPOSE, chunk ids]
```

try to maximize the number of chunks received. The retransmission is inspired from ARQ [FW02] (i.e., *à la* TCP) where a node stubbornly requests the node that sent a proposal until it receives the requested chunks. The requesting node emits up to r additional requests from the missing chunks. These changes define the baseline protocol on top of which *(i)* we justify the need for FEC coding (as well as tailor it specifically for our needs and finely parameterize) and *(ii)* we improve the retransmission mechanism for leveraging the many duplicates created by the first proposing/gossiping phase of the protocol (Chapter 5). Before this, we expose the experimental setup (Section 3.4) and evaluate the impact on live streaming of key parameters such as the fanout and the proactiveness of gossip, that is, the frequency at which communication partners change (Chapter 4).

3.4 Experimental setup

All algorithms presented in this thesis are implemented in Java on top of a framework we developed for deployment and execution over large testbeds such as PlanetLab [Pla] or Grid'5000 [Gri].

Throughout this thesis, we consider that nodes can fail by crashing, or exhibit freeriding behavior, that is, decrease their contribution while still benefiting from the system. Byzantine attacks on the peer sampling service are covered in [BGK+09]. Pollution attacks, i.e., nodes that inject junk content in the system, are orthogonal to the problems we tackle and are considered in [LCW+06, DHRS07].

3.4.1 Bandwidth Constraints

PlanetLab PlanetLab nodes, located mostly in research and educational institutions, benefit from high bandwidth capabilities. As such, PlanetLab is not representative of a typical collaborative peer-to-peer system [SPBP06], in which most nodes would be sitting behind ADSL connection, with an asymmetric bandwidth and limited upload/download capabilities. We thus artificially limit the upload capability of PlanetLab nodes so that they match the bandwidth usually available for home users. We focus on upload as it is a well-known fact that download capabilities are much higher than upload ones. In practice, nodes never exceed their given upload capability, but some nodes (between 5% and 12%), contribute way less than their capability, because of high CPU load and/or high bandwidth demand by other PlanetLab experiments, e.g., [LGL08]. In other words, the average used capabilities of nodes is always less or equal to their given upload capability limit.

We implemented, at the application level, two ways of limiting the bandwidth of nodes:

1. Bandwidth Throttling. All data that is about to cross the bandwidth limit is queued and sent as soon as there is enough available bandwidth. This

guarantees that nodes never send bursts of data, but incurs delay in data transmission.

2. Token Bucket Limiter. A token bucket is defined by the maximum number of tokens it can contain, representing the maximum amount of data it can send at once and a rate at which tokens are added, representing the upload bandwidth of the limiter. When data is tentatively sent, the token bucket has to contain a number of tokens greater or equal to the amount of data sent to be effectively sent. If the bucket does not contain enough token to send the data, the data is discarded.

Grid'5000 The use of a cluster platform like Grid'5000 allows us to have a controlled and reproducible setting, while at the same time simulating realistic network conditions. Besides the bandwidth limitation, we implemented a communication layer that provides delays and message loss as connectivity in a cluster is not representative of the Internet.

3.4.2 Metrics

We evaluate the performance of the considered protocols according to two metrics: stream lag and stream quality. We define the *stream lag* as the difference between the time at which the stream was sent by the source and the time at which a node could actually view it. Intuitively it measures *how live* the stream is. We, then, define the *stream quality* as the percentage of nodes that receive a completely clear stream, that is one in which 100% of the chunks can be played correctly. The rationale behind this choice is that a stream where more than 1% of the chunks are missing can be shown to be already very disturbing [BPLCH09]. Video formats differ in the way they compress the stream and losing data representing B-frames, for instance, can be less disturbing than other types of frames. In practice, however, it is very difficult to prioritize data in the stream as very few applications, if any, provide this kind of information at the stream packet level [LCC07].

When such a 100% quality is impossible to reach, we define a maximum jitter percentage in the stream: a 1% jittered stream means it contains at least 99% of the original stream, i.e., at most 1% of the groups were not complete. If a group is jittered, it does not mean the whole group is not viewable, but simply that the node delivered less than k chunks leading to audio glitches or video artifacts (using systematic coding, a node receiving $k-1$ out of the k original packets, experiences a $(k-1)/k\%$ delivery in that window). When not using FEC, the property is weaker since if there is at most 1% missing chunks, there is no information on how scattered they can be, i.e., it simply represents the *delivery ratio* defined as the percentage of chunks received [ZZSY07].

Pour ébranler une hypothèse, il ne faut quelquefois que la pousser aussi loin qu'elle peut aller.

Denis Diderot

4

Stretching Gossip with Live Streaming

Gossip protocols have been shown to be effective for challenging applications like file sharing [DXL$^+$06] and live streaming [LCW$^+$06, BMM$^+$08, LCM$^+$08]. Yet, most of the work evaluating gossip has considered ideal settings, e.g., unconstrained bandwidth, no (or uniform) message loss, global knowledge about the state of all nodes. Evaluations conducted through simulation [DXL$^+$06, MTGR07, BMM$^+$08] also assume that key parameters of gossip, such as fanout and gossip rate, can be arbitrarily tuned to improve robustness and to adequately adapt to network dynamics. Moreover, exceptions of gossip experiments in real settings assume infinite bandwidth [LCW$^+$06, LCM$^+$08], or consider applications with low bandwidth needs, such as membership maintenance [DOvNB07].

In the presence of constrained bandwidth and greedy applications, we have no evidence that gossip will fulfill its promises. The impact of the key parameters of gossip protocols in such contexts remains unexplored. For instance, the impact of varying the fanout, an obvious knob to tune the robustness of a gossip protocol, has never been determined.

The rate at which the gossip partners are changed reflects the *proactiveness* of a gossip protocol. The impact of varying this rate has never been studied either. At one extreme, one might consider a gossip protocol where nodes change their neighbors at every communication step (this is the scheme typically considered in theoretical studies). At the other extreme, nodes would never change their communication partners unless they notice malfunctions. This is typically the approach underlying structured or mesh-based systems where the unstructured overlay used to create random or deterministic dissemination trees, once constructed, is kept as is until the overlay connectivity is reduced (e.g., a node has less than a given number of neighbors), or a node feels it is incorrectly fed [LXQ+08]. A wide range of schemes can be considered between these two extremes.

4.1 Gossip's Key Parameters

We focus on the following two key parameters:

- **Fanout** The fanout is defined as the number of communication partners each node contacts in each gossip operation. It has been theoretically shown that a fanout greater than $\ln(n)$ in an infect-and-die model [KMG03] ensures a highly reliable dissemination. Theory also assumes that increasing the fanout results in an even more robust (as the probability to receive a chunk id increases) and faster dissemination (as the degree of the resulting dissemination tree increases). In practice, however, too high a fanout can negatively impact performance as heavily requested nodes may exceed their capabilities in bandwidth constrained environments.
- **Proactiveness** We define proactiveness as the *rate* at which a node modifies its set of communication partners. We explore two ways of modifying this set.

  First, the node may locally refresh its set of communication partners and change the output of selectNodes every X calls (line 31 in Algorithm 3.3, page 38). In short, when $X = 1$ the gossip partners of the node change at every call to selectNodes (i.e., every gossip period), whereas $X = \infty$ means that the communication partners of a node never change.

  Second, every Y gossip periods, the node may contact f random communication partners asking to be inserted in their views. When $Y = 1$, a node A sends a *feed-me* message to f random partners every gossip period asking them to feed it. Each of the random f partners replaces a random node from its current set of f partners with the node A.

4.2 Evaluation

We evaluate the impact of the fanout and proactiveness of a gossip protocol by deploying a streaming application based on Algorithm 3.3 (page 38) over a set of 230 PlanetLab nodes.

Bandwidth Constraints PlanetLab nodes benefit from high bandwidth capabilities and therefore are not representative of nodes with limited capabilities. We thus artificially constrain the upload bandwidths of nodes with three different caps: 700 kbps, 1000 kbps and 2000 kbps using bandwidth throttling.

Streaming Configuration The source node generates a stream of 600 kbps and proposes it to 7 nodes in all experiments. To provide further tolerance to message loss (combined with retransmission), the source groups packets in windows of 110 packets, including 9 *FEC encoded packets*. The gossip period T_g is set to 200 ms.

4.2.1 Impact of Varying the Fanout

We start our analysis by measuring how varying the fanout impacts each of the two metrics, stream lag and stream quality, with a proactiveness degree of 1 ($X = 1$). Results are depicted in Figures 4.1–4.3. Figure 4.1 shows the percentage of nodes that can view the stream with less than 1% jitter for various stream lags in a setting where all nodes have their upload capabilities capped at 700 kbps.

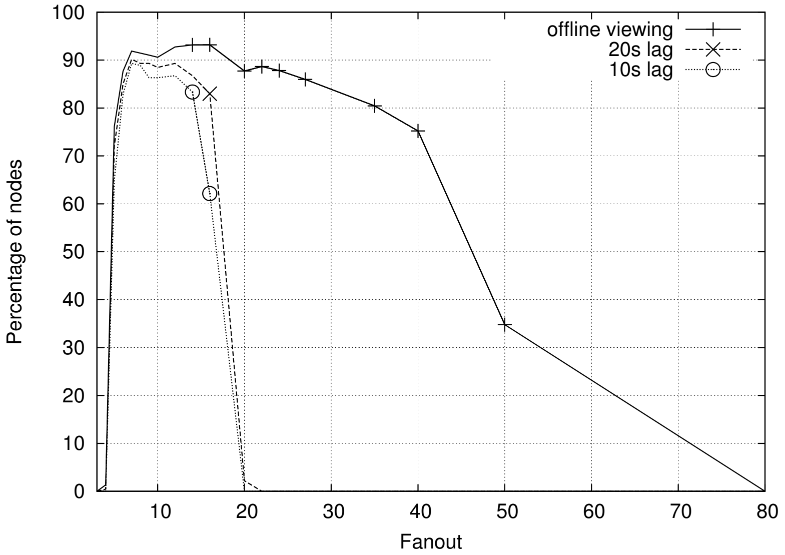


Figure 4.1: Percentage of nodes viewing the stream with less than 1% of jitter (upload capped at 700 kbps).

Optimal Fanout Range The plot clearly highlights an optimal range of fanout values (from 7 to 15 in Figure 4.1) that gives the best performance independently of the lag value considered. Lower fanout values are insufficient to achieve effective dissemination, while larger values generate higher network traffic and congestion, thus decreasing the obtainable stream quality.

The plot also shows that while the lines corresponding to finite lag values have a bell shape without any flat region, the one corresponding to offline viewing does not drop dramatically until a fanout value of 40. The bandwidth throttling mechanism is in fact able to recover from the congestion generated by large fanout values once the source has stopped generating new packets. For fanouts above 40, on the other hand, such recovery does not occur.

Critical Lag Value A different view on the same set of data is provided by Figure 4.2. For each value t, the plot shows the percentage of nodes that can view at least 99% of the stream with a lag shorter than t. A fanout in the optimal range (e.g., 7) causes almost all nodes to receive a high quality stream after a critical lag value ($t = 5\,\mathrm{s}$ for a fanout of 7). Moderately larger fanout values cause this critical value to increase ($t = 22\,\mathrm{s}$ for a fanout of 20), while for fanouts above 35, no critical value is present. Rather, congestion causes significant performance degradation. With a fanout of 40, only 20% of the nodes can view the stream with a lag shorter than 60 s, and a lag of 90 s is necessary to reach 75% of the nodes.

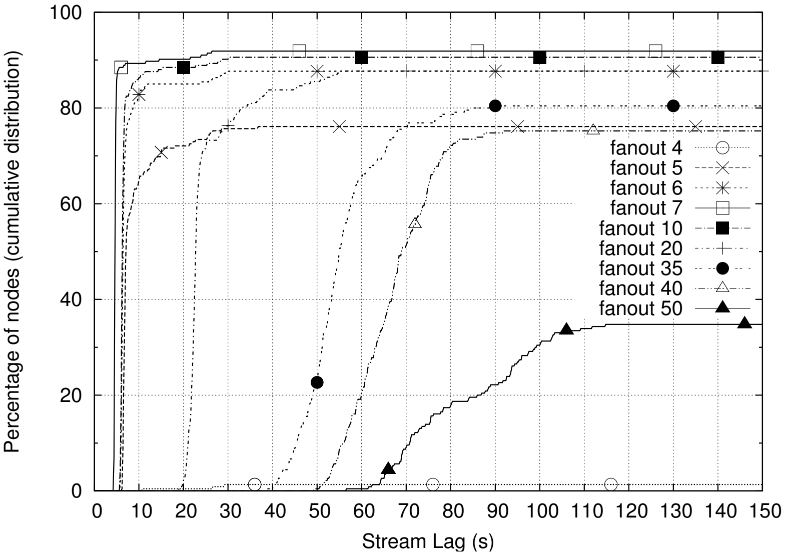


Figure 4.2: Cumulative distribution of stream lag with various fanouts (upload capped at 700 kbps).

Behavior with Less Tight Distributions The presence of an optimal fanout value is clearly a result of operating under limited bandwidth. Figure 4.3 complements the picture by showing how fanout affects performance under less critical conditions: 1000 kbps and 2000 kbps of capped bandwidth. As available bandwidth increases, the range of good fanout values clearly becomes larger and larger and tends to move to the right. With an available bandwidth of 1000 kbps, which is more than 1.67 times the stream rate, it is still possible to identify a clear region outside of which performance degrades significantly. With 2000 kbps of available bandwidth, both the 10 s-lag and the offline performance figures appear to remain high even with very large fanout values. This behavior may be better understood by examining how bandwidth is actually used by the PlanetLab nodes involved in the dissemination.

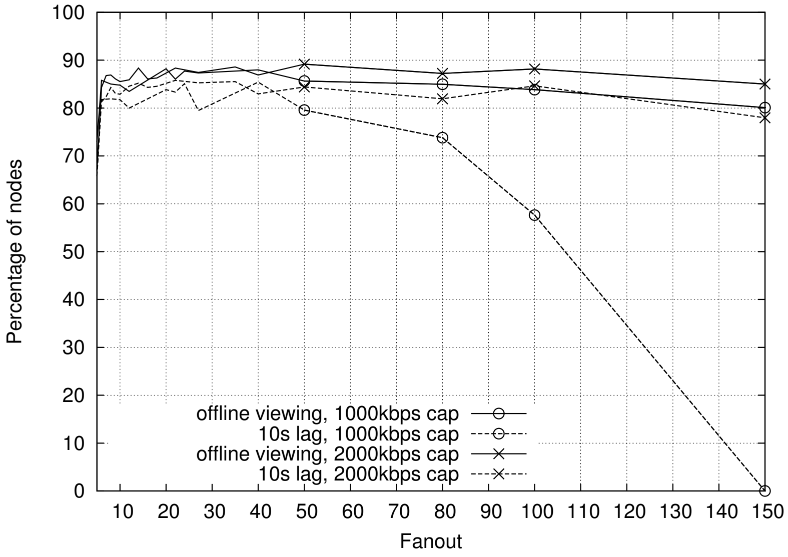


Figure 4.3: Percentage of nodes viewing the stream with less than 1% of jitter with upload caps of 1000 kbps and 2000 kbps, and different fanout values.

Bandwidth Usage with Different Fanouts Values Figure 4.4 shows the distribution of bandwidth utilization over all the nodes involved in the experiments sorted from the one contributing the most to the one contributing the least. The plot immediately highlights an interesting property: even though all the considered scenarios have a homogeneous bandwidth cap, the distribution of utilized bandwidth is highly heterogeneous. This behavior is a direct result of the three-phase protocol employed for disseminating large content.

According to Algorithm 3.3 (page 38), all nodes contribute to the gossip dissemination by sending their proposal messages to the same fanout number of nodes. However, the contribution of a node in terms of serve messages depends

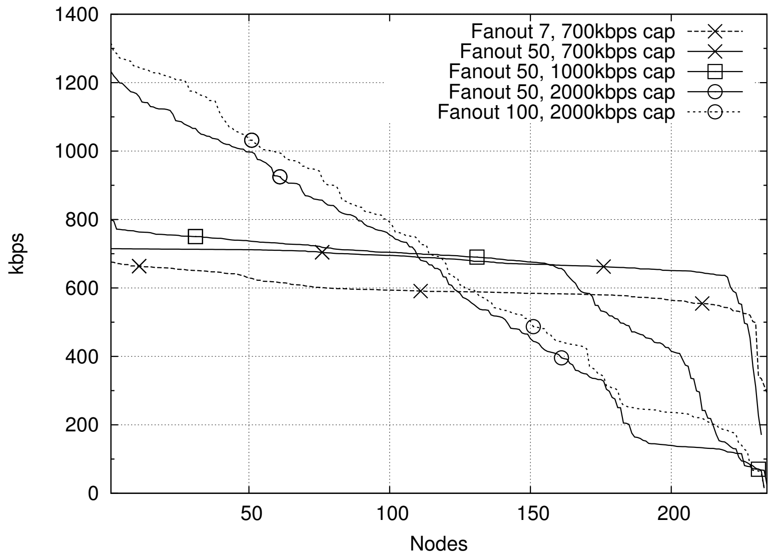


Figure 4.4: Distribution of bandwidth usage among nodes with different fanout values and upload caps.

on the probability that its proposal messages are accepted by other nodes. In general, nodes with low-latency and reliable connections (i.e., *good* nodes) have higher probabilities of seeing their proposals accepted. This is confirmed by Figure 4.4. The plot also shows that the heterogeneity in bandwidth utilization increases with the amount of available bandwidth. For example, the lines for the 700 kbps bandwidth cap show an almost homogeneous distribution apart from a small set of *bad* nodes. This is because the higher latencies exhibited by good nodes when their bandwidth utilization is close to the limit causes other nodes to work more, thus equalizing bandwidth consumption. On the other hand, if we observe the lines corresponding to a fanout of 50 in the 1000 kbps and 2000 kbps scenarios and to a fanout of 100 in the 2000 kbps scenario, we see that good nodes have enough spare capability to operate without saturating their bandwidths. As a result, the contribution of nodes remains heterogeneous.

4.2.2 Proactiveness

We present our analysis of gossip proactiveness by showing how refreshing the set of communication partners affects application performance. Figure 4.5 presents the results obtained by varying the *view refresh rate*, X, in a scenario with a 700 kbps bandwidth cap. The plot shows three lines corresponding to stream lags of 10 s and 20 s as well as to offline viewing. In all cases, results confirm that the best performance is obtained by varying the set of gossip partners at every communication round. If on the other hand, the set of communication partners remains constant for long periods of time, a small set of nodes end up having the responsibility of feeding large numbers of nodes for as long as they keep being selected early in the dissemination process. This means that their upload rates remain constantly higher than their allowed bandwidth limits, ultimately resulting in high levels of congestion, huge latencies, and message loss.

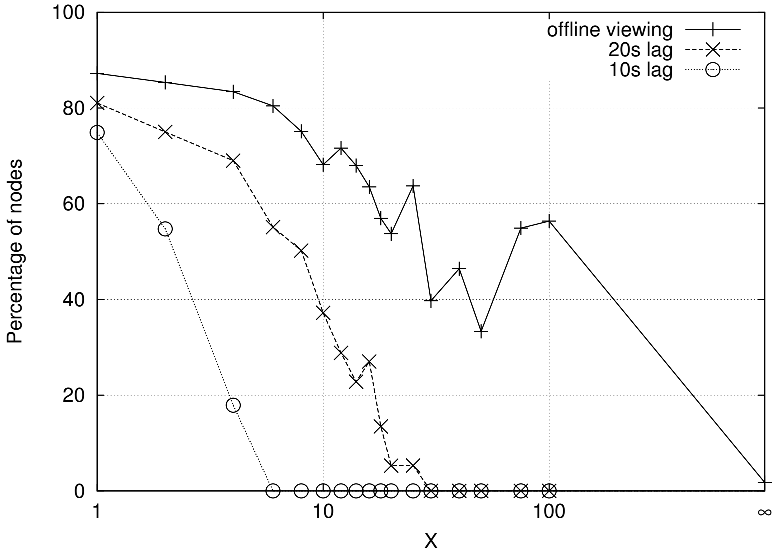


Figure 4.5: Percentage of nodes viewing the stream with at most 1% jitter as a function of the refresh rate X.

In accordance with these observations, Figure 4.5 shows that the slope at which the curves decrease with X is most negative for a lag of 10 s. This is because longer values of lag allow the bandwidth throttling mechanism more time to recover from the bursts generated by a constant set of communication partners. Nonetheless, a completely static dissemination mesh invariably yields bad performance even for offline viewing as the load becomes then concentrated on a very small set of nodes for the entire experiment.

Requesting Nodes to Update their Views A second way to modify the proactive behavior of the considered streaming protocol is for nodes to periodically request a new set of nodes to feed them, i.e., every Y dissemination rounds. In Figure 4.6, we show the results obtained with different values of Y. Results show that this technique remains inferior to the simpler approach of choosing a *view refresh rate* of $X = 1$, as discussed above. The additional messages used by this approach may in fact be lost or delayed while the node is congested, resulting in a larger Y than planned.

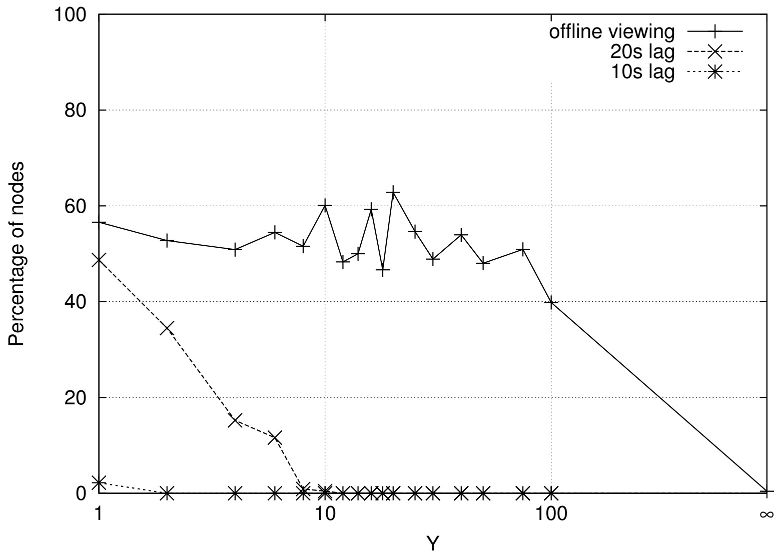


Figure 4.6: Percentage of nodes viewing the stream with at most 1% jitter as a function of the request rate Y.

4.2.3 Performance in the Presence of Churn

Finally, we evaluate the impact of proactiveness in the presence of churn with $Y = \infty$. Results are depicted in Figures 4.7 and 4.8. The experiments consist in randomly picking a percentage of nodes and make them fail simultaneously. We compare the baseline results (e.g., when no nodes fail) together with increasing percentages of failing nodes (from 10% to 80%). Figure 4.7 shows the percentage of remaining nodes experiencing less than 1% jitter after the catastrophic failure, for values of X of 1, 2, 20 and ∞.

Figure 4.7 clearly shows that a completely dynamic mesh offers the best performance in terms of ability to withstand churn. With 35% of churn, a proactiveness of $X = 1$ is able to deliver an unaffected stream to 60% of the remaining nodes, while the percentage drops to 32% for $X = 2$ and a stream lag of 20 seconds. The results obtained with large values of X and in particular when $X = \infty$ show very high degrees of variability from experiment to experiment. The resulting static or semi-static random graph may become completely unable to disseminate the stream if failing nodes are close to the source or it may appear extremely resilient to churn if failing nodes are located at the edge of the network. On average, performance is therefore in favor of a completely dynamic graph ($X = 1$).

It should be noted that Figure 4.7 only shows how many nodes manage to remain completely unaware of the churn event. To characterize the extent of the performance decrease experienced by all surviving nodes, Figure 4.8 shows

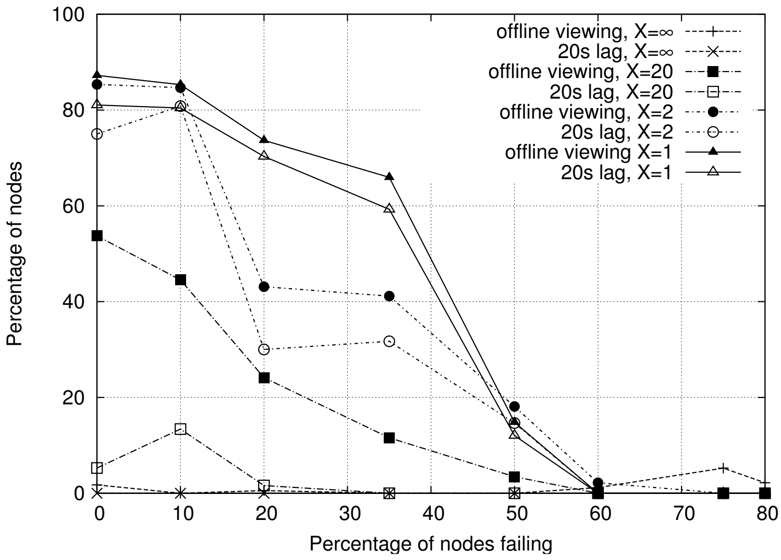


Figure 4.7: Percentage of surviving nodes experiencing less than 1% jitter for different values of X.

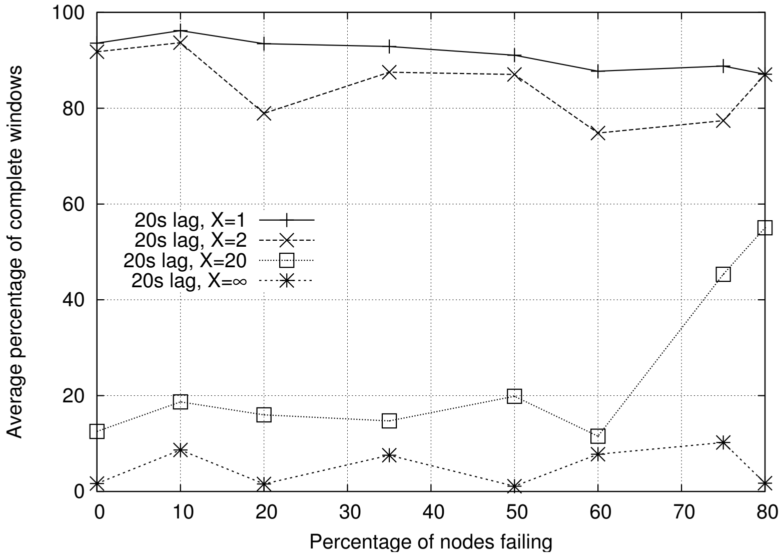


Figure 4.8: Average percentage of complete windows for surviving nodes.

the average percentage of decoded windows over the total number of windows streamed by the source. With $X = 1$, the protocol is almost unaffected by churn, and nodes correctly receive over 90% of the windows for all churn percentages lower than 80%.

4.3 Summary

The evaluations of gossip protocols for dissemination usually back up the associated theoretical claims: gossip can be tuned to achieve reliable dissemination in the presence of churn. In this chapter, we challenged these results, obtained mostly through simulations in close-to-ideal settings, and evaluated a gossip-based live streaming application in a real deployment over 230 PlanetLab nodes.

Our results show that message loss and limited bandwidth significantly restrict the range of parameter values in which gossip can successfully operate. First, the fanout cannot be increased arbitrarily to improve reliability and latency, but must remain small enough to prevent bandwidth saturation. Second, the set of communication partners should be changed frequently, at every communication round, in order to minimize congestion and provide an effective response to churn.

Better a diamond with a flaw than a pebble without.

Commonly attributed to Confucius

5 Boosting Gossip for Live Streaming: Gossip++

In this chapter, we justify the need for both erasure coding (FEC) and retransmission, propose a novel content request scheme that is specially tailored for gossip, and show how these mechanisms can effectively complement each other in a new gossip protocol, called gossip++.

As described in the previous chapter, the best performance of gossip-based streaming, based on the proposed infect-and-die gossip, is achieved with fanout values that are only slightly larger than $\ln(n)$. Such small values allow the dissemination protocol to operate without saturating the bandwidth of nodes, but they also limit the redundancy of data dissemination. This, coupled with message losses during the second and third phases of the protocol, makes it almost impossible for such a naive three-phase gossip protocol to deliver the entirety of the available content to all participating nodes.

To exemplify this effect, we expose in Figure 5.1 a set of experiments disseminating a video stream of 680 kbps to 200 nodes (Grid'5000 testbed) and evaluate the behavior of the three phases of the initial gossip protocol (Algorithm 3.2, page 35) with several fanout values. All nodes except the source have a bandwidth cap of 800 kbps and the fanout of the source is set to 5.

On average each chunk identifier sent in the first phase of the protocol was received by an average of 99% of the nodes, while the number of nodes receiving all advertisements varied from 0% with a fanout of 7 to 60% with a fanout of 10. While this seems to match the theoretical results about the reliability of

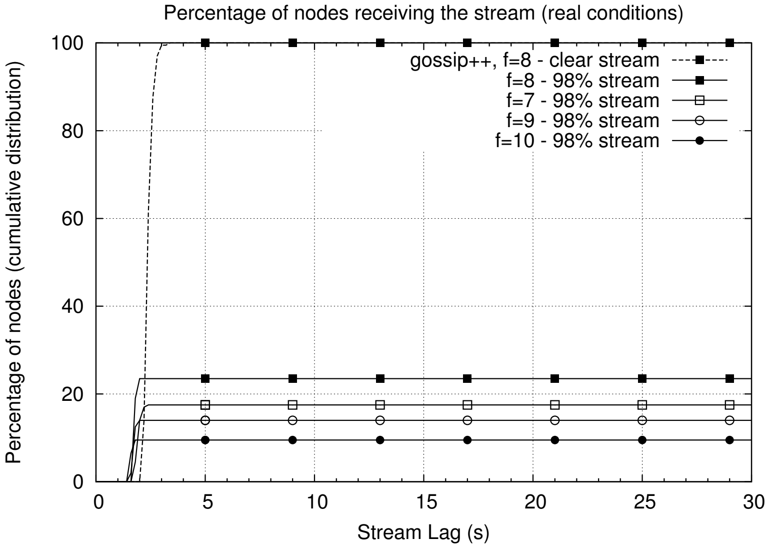


Figure 5.1: In a realistic scenario with constrained bandwidth (800 kbps bandwidth cap) and additional message losses (1%), increasing the fanout of standard gossip does not help (Chapter 4). On the other hand, gossip++ can deliver a clear stream to all nodes.

gossip [KMG03], the situation changes dramatically if we analyze the number of actual chunks received at the end of phase 3. In this case, the average delivery ratio (i.e., the percentage of chunks received of the original stream) drops to about 97% with no node being able to receive all or even 99% of the stream, regardless of the fanout. Moreover, the best performance is achieved with a fanout of 8, which provides 23.5% of the nodes with between 98% and 99% of the stream with a stream lag shorter than 3 seconds as shown in Figure 5.1.

This level of performance may be sufficient for some applications (e.g., news or query dissemination). However it is unacceptable for an application like video streaming [BPLCH09].

5.1 Gossip++

Gossip++ is a combination of two simple mechanisms: *(i) Codec*, an erasure coding scheme, and *(ii) Claim*, a content request scheme that leverages gossip duplication to diversify the retransmission sources of missing information. *Codec* operates by adding redundant encoded chunks to the stream so that it can be reconstructed after the loss of a random subset of its chunks. *Claim*, on the other hand, allows nodes to re-request missing content by recontacting the nodes from which they received advertisements for the corresponding chunks, leveraging the duplicates created by gossip. Our experiments show that neither mechanism alone can guarantee reliable dissemination of all the streaming data to all nodes. On the other hand, their combination is particularly effective and is able to provide all nodes with a clear stream even in tight bandwidth scenarios, in the presence of crashes, or up to 20% of freeriding nodes with proportional slack in average bandwidth capability.

Intuitively this can be explained by observing that each of the two proposed mechanisms addresses a different problem of gossip dissemination. *Codec* manages to reconstruct the chunks that could not be delivered by gossip due to its probabilistic guarantees. On the other hand, *Claim* is able to recover from message loss occurring at any point during the last two phases of the dissemination process, that is the *request* and the *serve*.

In order to provide reliable dissemination of streaming content, we augment the three-phase gossip protocol with two components: *Codec* and *Claim* as described in the following.

5.1.1 Codec

Codec is a forward error correction (FEC) mechanism which feeds information back into the gossip protocol to decrease the overhead added by the FEC. This mechanism increases the efficiency of the dissemination achieved by three-phase gossip in three major ways. First, since each chunk is proposed to all nodes with high probability, some nodes do not receive proposals for all chunks, even when there is no message loss. FEC allows nodes to recover these missing chunks even if they cannot actually be requested from other nodes. Second, FEC helps in recovering from message losses occurring in all three phases. Finally, decoding a group of chunks to recover the missing ones often takes less time (i.e., in the order of 40 ms) than actually requesting or re-requesting and receiving the missing ones (i.e., at least a network round-trip time).

Erasure Coding (FEC) The source of the stream uses a *block-based* FEC implementation [Riz97] to create, for each group of k source chunks, c additional encoded ones. A node receiving at least k random chunks from the $k + c$ possible ones is thus able to decode the k source chunks and forward them to the video player (step (i) in Figure 5.2). If the node received less than k chunks, the group is considered *jittered.* Nevertheless, using systematic coding (i.e., the source chunks are not altered), a node receiving $j < k$ chunks can still deliver the $i \leq j$ source chunks that it received. In other words, assuming the k source chunks represent a duration t of audiovisual stream, the jittered group does not inevitably represent a blank screen without sound for t time. If i is close to k, the decreased performance can be, in the best case, almost unnoticeable to the user (e.g., losing a B-frame).

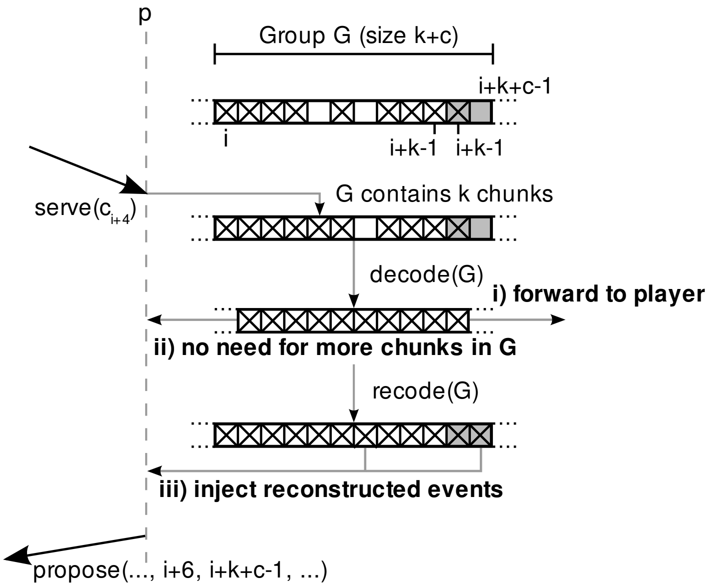


Figure 5.2: *Codec*: a node p receiving k chunks in G decodes the group to reconstruct the k source chunks and sends them to the player (step (i)). Node p then signals the protocol not to request any more chunks in G (step (ii)). Optionally (step (iii)), p re-encodes the k source chunks and injects reconstructed chunks into the protocol.

The Cost of FEC The cost of using FEC mainly consists of network costs. The CPU cost of coding and decoding was a concern 15 years ago but is negligible nowadays with the type of FEC we are using. On the other hand, the source needs to send $k + c$ chunks for each group of k. This constitutes an overhead of $\frac{c}{k+c}$ in terms of outgoing bandwidth. The remaining nodes, however, can cut down this overhead as described in the following.

Codec Operation The key property of *Codec* is the observation that a node can stop requesting chunks for a given group of $k + c$ as soon as it is able to decode the group, i.e., as soon as it has received $k' \geq k$ of the $k + c$ chunks (step (ii) in Figure 5.2). The decoding process then provides the node with the k source chunks needed to play the stream. This means that it does not need to request more chunks in this group from other nodes. This allows the node to save incoming bandwidth and most importantly it allows other nodes to save their outgoing bandwidth that they can thus use for serving useful chunks to nodes in need. Optionally (step (iii) in Figure 5.2), in order not to stop abruptly the dissemination of the reconstructed chunks (source or encoded chunks: chunks $i + 6$ and $i + k + c - 1$ in that case), nodes can re-inject decoded chunks into the protocol.[1] The performance improvement of this step is evaluated and discussed in Section 5.2.8.

5.1.2 Claim

While *Codec* can reconstruct missing chunks, it still needs at least k chunks from each group. *Claim* uses retransmission to make it possible to recover these k chunks even when message loss affects more than c chunks per group. In doing this, it takes full advantage of the redundancy of gossip by leveraging the duplicate proposals received for a chunk. Instead of stubbornly requesting the same sender, the requesting node re-requests nodes in the set of proposing nodes in a round-robin manner as presented in Figure 5.3.

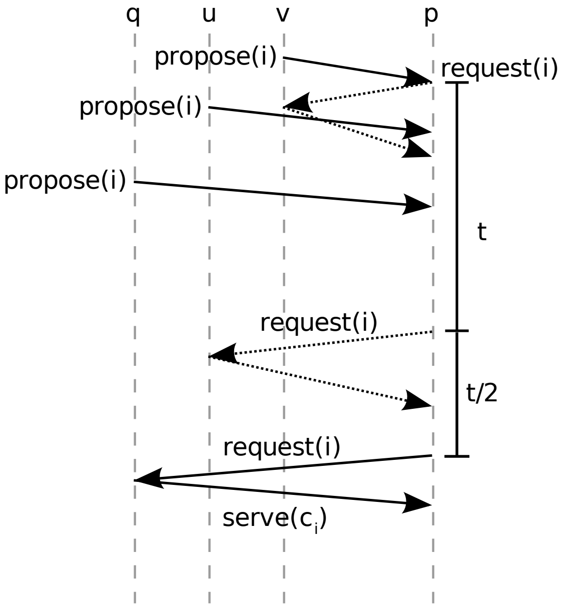


Figure 5.3: *Claim*: Node v has proposed chunk i to node p which requested it. Either the request or the serve was lost and p, instead of re-requesting from v, now requests u, that also proposed chunk i. If u fails to serve c_i, p requests chunk i again to another node that proposed i. Node q finally serves p with c_i.

Nodes can emit up to a number r of re-requests for each chunk. The first re-request for a given chunk is scheduled to be sent after a timeout defined as $\mu + 3.29\sigma$ where μ and σ are respectively the average and standard deviation of

[1] This is possible because FEC coding is deterministic, meaning that the k source chunks produce the exact same c encoded chunks independently of the encoding node and thus the injection of reconstructed source or encoded chunks will be identical as the ones produced by the source.

the roundtrip times experienced by the node.[2] Further re-requests, if needed, are scheduled to be sent each time after half of the previous timeout until a minimum fixed timeout is reached.

5.2 Evaluation

We evaluated gossip++ on 200 nodes, deployed over 40 Grid'5000 machines. The use of a cluster platform like Grid'5000 allows us to have a controlled and reproducible setting, while at the same time simulating realistic network conditions. To achieve this, we implemented a communication layer that provides bandwidth limitation, delays and message loss. Specifically, we give the source enough bandwidth to serve 5 nodes in parallel, and we limit the upload bandwidth of each other node to 800 kbps unless otherwise specified using a token-bucket mechanism with a bucket size of 200 KB. This means that all burst of cumulated size over 200 KB are automatically dropped. On top of this, we introduce a 1% message loss rate unless otherwise indicated. Finally, we add a random delay between 0 and 200 ms to all sent messages.

All nodes, except the source, gossip with a fanout of 8, unless otherwise specified. This proved to be the value providing the best performance with a 800 kbps bandwidth limit. The source, on the other hand, uses a fanout of 5 to gossip a stream fed by VLC at 679.79 kbps on average. Before gossiping, the source encodes groups of $k = 100$ chunks of 1316 bytes, and creates $c = 5$ additional coded chunks except when otherwise specified. The average stream sent by the source is thus 713.75 kbps, adding 5% overhead to the stream. The gossip period is set to 200 ms and all messages are sent over UDP.

5.2.1 Metrics

In this evaluation, we particularly focus on the percentage of nodes that receive a completely clear stream, that is one in which 100% of the chunks can be played correctly. The rationale behind this choice is that a stream where more than 1% of the chunks are missing can be shown to be already very disturbing [BPLCH09]. Video formats differ in the way they compress the stream and losing data representing B-frames, for instance, can be less disturbing than other types of frames. In practice, however, it is very difficult to prioritize data in the stream as very few applications, if any, provide this kind of information at the stream packet level [LCC07].

When using plain gossip without *Codec*, playing 100% of the chunks requires receiving every single chunk disseminated by the source. When using *Codec*,

[2] Note that trying to keep track of roundtrip times from past communication partners individually does not scale, since partners are taken at random from the set of all nodes.

on the other hand, a node can play all chunks if it receives at least k random chunks per group of $k + c$ chunks.

5.2.2 Overview

In the following, we present the results of our evaluation. We first show in Section 5.2.3 that plain gossip is insufficient in video streaming applications, thereby motivating the use of *Codec*. Second, we observe that when bandwidth is limited, *Codec* alone is not sufficient and that it can provide satisfactory performance only in combination with a retransmission mechanism like *Claim* (Section 5.2.4). We then evaluate *Claim* alone as well as in combination with *Codec*. Again, *Claim* alone proves to be insufficient, but its combination with *Codec* provides all nodes with a clear stream even in highly constrained settings. In Section 5.2.5 we evaluate the impact of different FEC percentages in combination with *Claim*. This allows us to identify 5% FEC as the best trade off. Finally, we push the analysis further and examine performance *(i)* in the presence of catastrophic crash failures (Section 5.2.6), *(ii)* when the bandwidth gets more and more constrained (Section 5.2.7), and *(iii)* in the presence of nodes that cannot or do not provide their fair share of work (Section 5.2.8).

5.2.3 Need for Codec

We start our analysis by motivating the need for *Codec* as part of our improved gossip-based solution for live streaming. According, to the analysis in Chapter 4), we know that in constrained bandwidth scenarios, it is not possible to increase the fanout of nodes arbitrarily. Too large values end up saturating the available bandwidth causing drops in performance. As presented in the beginning of this chapter (page 51), in a network of 200 nodes, and an 800 kbps limit on the upload bandwidth, the fanout offering the best performance is 8. Even with this fanout, however, no node is able to obtain a perfectly clear stream and only 23.5% of the nodes are able to experience a 1% jittered stream.

To get a better understanding of this poor performance, we ran an experiment with a fanout of 8 in an unconstrained bandwidth scenario. Results are depicted in Figure 5.4: without bandwidth constraints, all nodes can view 99% of the stream but only 8.5% can receive a clear one. This is because a fanout of 8 is too low to guarantee reliable dissemination of chunk advertisements. On the other hand, larger values prove to be too high when bandwidth is constrained to 800 kbps.

The natural solution to these problems is therefore the introduction of *Codec*. With its use, the situation radically changes and all nodes are able to view a completely clear stream in this ideal network scenario with a stream lag lower than 3.3 s.

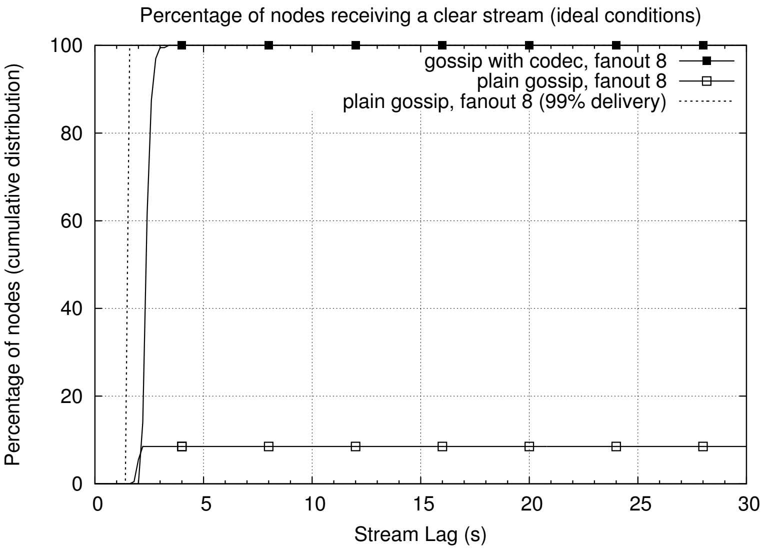


Figure 5.4: In an ideal scenario where bandwidth is unconstrained and without message loss, plain gossip delivers a clear stream to only 8.5% of nodes since all nodes do not receive a proposal for each chunk. Adding FEC on the other hand, 100% of the nodes can view the original stream.

Still, we recognize that in this very favored environment (i.e., no message loss nor bandwidth cap), plain gossip is still quite efficient since all the nodes suffer at most 1% missing chunks in their stream (dashed line in Figure 5.4), comforting the theoretical results that each chunk proposal (followed by its actual serve) reaches all nodes with high probability.

5.2.4 Realistic Conditions: Need for Claim

When moving to realistic conditions, however, it becomes clear that using *Codec* alone is not sufficient to provide reasonable streaming performance. With a constraint on the upload bandwidth of 800 kbps and a message loss rate of 1%, no node is able to receive a clear stream even when *Codec* is used. Nonetheless, *Codec* allows 64.5% of the nodes to view 99% of the original stream against a flat 0% provided by plain gossip.

These results show that, while *Codec* is able to address the inability of gossip to reach all nodes with a fanout of 8, its use is not sufficient to recover missing chunks that were lost, because of losses in the communication layer and the associated bandwidth constraints. The situation, instead, improves dramatically if we add *Claim* to the picture. Figure 5.6 shows that *Claim* combined with

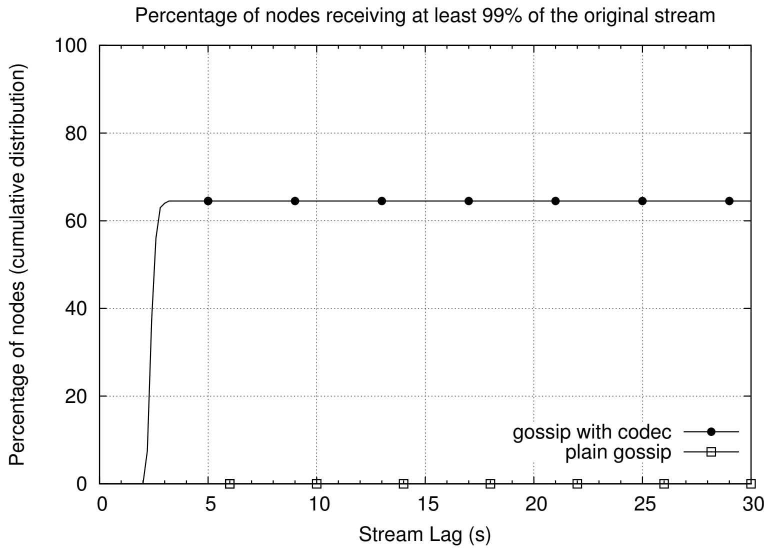


Figure 5.5: Without FEC, no node can even receive 99% of the original stream with constrained bandwidth and message losses. Using *Codec*, there is a large improvement since 64.5% of the nodes receive at least 99% of the original stream with a stream lag shorter than 3.3 s.

Codec is able to provide each node with a clear stream with a stream lag shorter than 3.5 s.

Increasing the percentage of FEC should intuitively help recover an increased proportion of missing chunks, be they not proposed or lost. We thus also show in Figure 5.6 results for 50% coding (resp. 100% coding), i.e., 2/3 (resp. 50%) of the received chunks are enough to decode a clear stream. We only show results for optimal fanouts, 4 and 3 respectively. The presented results are optimal in the sense that a lower fanout prevents gossip from disseminating proposals (and thus possibly chunks) to a large number of nodes and that larger fanouts create bursts such that more and more messages are dropped by the bandwidth limiter. Still, *Codec* alone can provide a clear stream to only 21.5% with 50% coding (resp. 3.5% for 100% coding).

The figure also shows that, interestingly enough, *Claim* alone is also unable to provide a significant improvement over plain gossip, with only 4.10% of the nodes receiving a clear stream. The reason is that *Claim* guarantees that a node that received proposals for a chunk will eventually receive the chunk when reclaiming it from one of the proposing nodes. However, with *Claim*, a node will never be able to retrieve chunks for which it never got a proposal.

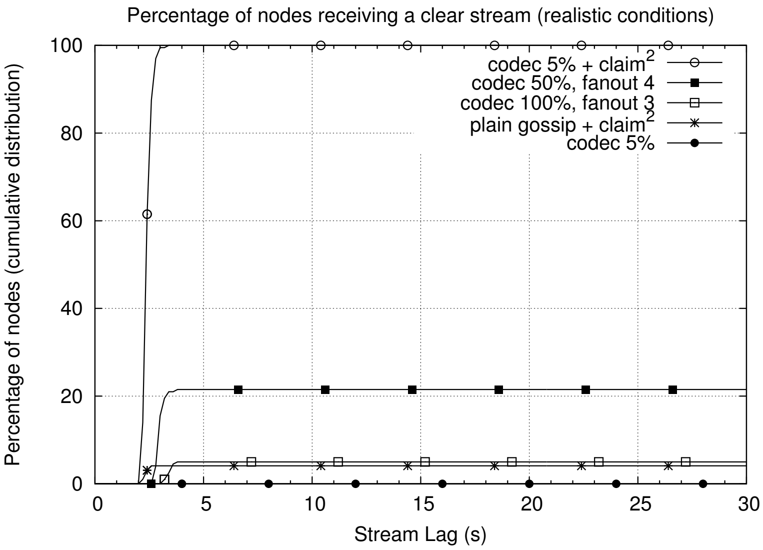


Figure 5.6: While *Codec* (5% coding) can provide a 1% jittered stream to 64.5% of nodes (Figure 5.5) it cannot, alone, provide a clear stream to any node. *Claim* improves plain gossip only a very little but significantly boosts the performance of *Codec*. When applied together the two techniques provide a clear stream to all nodes. Finally, increasing the percentage of FEC without retransmission does not help in recovering missing chunks.

5.2.5 Impact of Message Loss

The main contribution of *Codec* is the ability to recover chunks for which no proposal was received. The importance of this feature depends mainly on two factors: the fanout, and the message loss rate of the network.

To better understand the trade-offs associated with the configuration of *Codec*, we consider the performance of the combination of *Codec* and *Claim* in several scenarios with a bandwidth cap of 800 kbps, and message loss rates ranging from no message loss to 5% of message loss. In each of these scenarios we configured *Codec* with different levels of coding: 2%, 5%, 10%, 30% and 50%.

The results, depicted in Figure 5.7, show that *Codec* with 2% coding is not able to compensate the missing proposals for all nodes even in the absence of message loss. Higher coding percentages, on the other hand, provide very similar and good results up to 4% of message loss. With 5% of message loss, 50% coding performs slightly worse than the other percentages, providing only 94.9% of nodes with a clear stream compared to at least 98.7% for the others.

These figures can be better understood by analyzing the data in Figure 5.8. The plot shows the cumulative distribution of stream lag for the various FEC

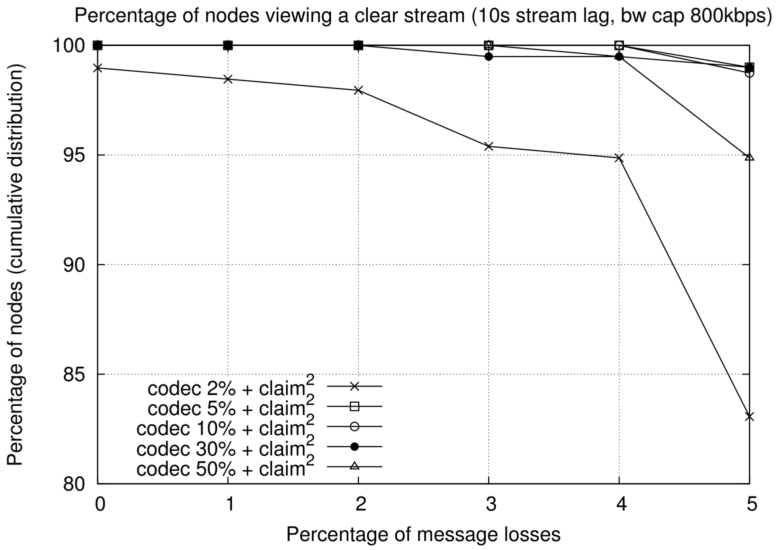


Figure 5.7: *Codec* with 2% and 50% coding provide a clear stream to a lower number of nodes than other *Codec* percentages as the percentage of message losses increases.

percentages in the presence of 5% message loss. *Codec* with 50% coding is indeed able to provide all nodes with a clear stream, but with a much longer stream lag. The reason is that adding a large percentage of coding represents a large overhead in terms of bandwidth meaning nodes try to send more data than they are allowed to by their bandwidth limiter. This leads to dropped messages and retransmissions ultimately explaining the poor results with respect to stream lag of *Codec* 30% and *Codec* 50%.

These observations are confirmed by Figure 5.9. Here we show the bandwidth usage of the source and the average requested bandwidth usage of nodes, before the bandwidth limit is applied. The picture shows that only 2% and 5% coding do not attempt to send more data than allowed to by the bandwidth limit. This means that they are the only two versions of the protocols that will not experience message loss as a result of the token bucket. This is confirmed by the fact that 2% coding and 5% coding exhibit the shortest stream lag in Figure 5.8.

In practical terms, the choice of the FEC percentage should therefore fall on the largest percentage that remains within the bandwidth constraints. This allows *Codec* and *Claim* to combine the fast dissemination with the ability to recover from missing proposals.

A final observation, on this set of experiments can be made by looking at Figure 5.10. The plot shows the percentage of chunks that *Codec* was able to reconstruct with each of the five coding percentages. The depicted percentage

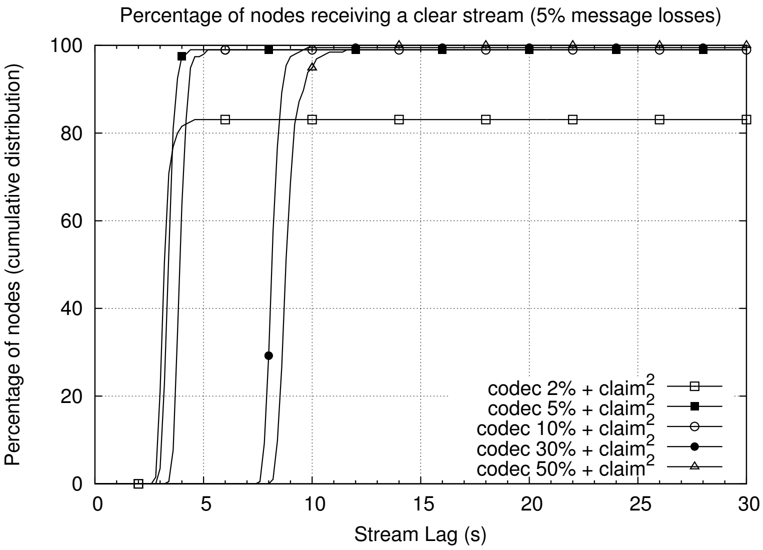


Figure 5.8: With 5% message loss, *Codec* with 2% coding provides a clear stream to a maximum of 83% of nodes. Other percentages reach at least 98.7% but with different stream lags. The bandwidth used by *Codec* with 30% and 50% coding is larger than with lower coding percentages (Figure 5.9). The traffic excess is limited by the token bucket, which results in needing *Claim* to recover missing chunks, which in turn, explains the larger stream lag.

includes both the chunks for which no proposal was received and the requests/re-requests that were saved, i.e., the missing chunks were reconstructed before a request/re-request was sent.

It is interesting to see that the recovery percentages grow linearly with message loss and correspond to around half of the percentages of FEC coding being considered. This means that the overhead added by *Codec* is partly recovered and is ultimately around half of the expected overhead value. The reason is that the more the message loss, the more the missing proposals and the need for retransmission for chunks that were proposed. Since each retransmission is associated with a timeout, the more the message loss and the longer the time available to *Codec* to reconstruct chunks in incomplete groups before actually sending a re-request.

5.2.6 Crashes

In this section, we consider the behavior of gossip++ in a catastrophic-failure scenario. Figure 5.11 reports a zoomed-in view of the percentage of nodes re-

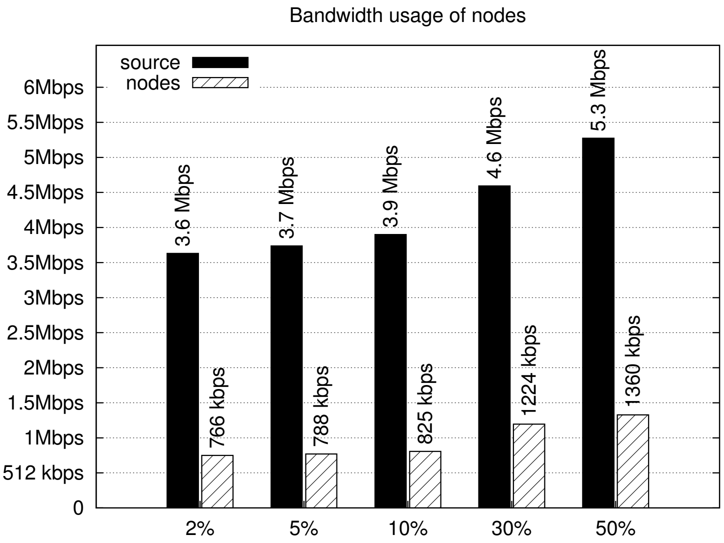


Figure 5.9: Bandwidth usage of both source and tentative bandwidth usage of nodes for different percentages of coding. The source bandwidth usage increases according to the FEC overhead and the tentative bandwidth usage of nodes quickly exceed the bandwidth cap of 800 kbps, explaining the bad performances in Figure 5.7 for 30% and 50% coding.

ceiving or decoding each chunk around the moment at which 20% or 50% of the nodes crash simultaneously.

The two vertical lines in the plot show respectively the minimum and the maximum chunk ids received by the nodes in the system at the moment of the crash. The distance between the two lines shows that the nodes fail across an interval of 54 chunks, corresponding to 0.7 s.

The performance lines, instead, start to drop at around chunk 3570. This is because some of the nodes that were supposed to serve that chunk and the following were among the crashed nodes. Overall, the picture shows that the crash only results in a minor performance glitch that lasts less than 1 s, before the remaining nodes can continue to view the stream undisturbed.

5.2.7 Constrained Environments

We evaluate the performance of gossip++ with variable amounts of available upload bandwidth. In doing this, we also consider a variant of *Codec*, *decode to player*, in which nodes continue requesting the proposed chunks that they have not yet received, even if they have been successfully decoded by the FEC (step (i) of *Codec* only, i.e., without signaling to the protocol that the group has

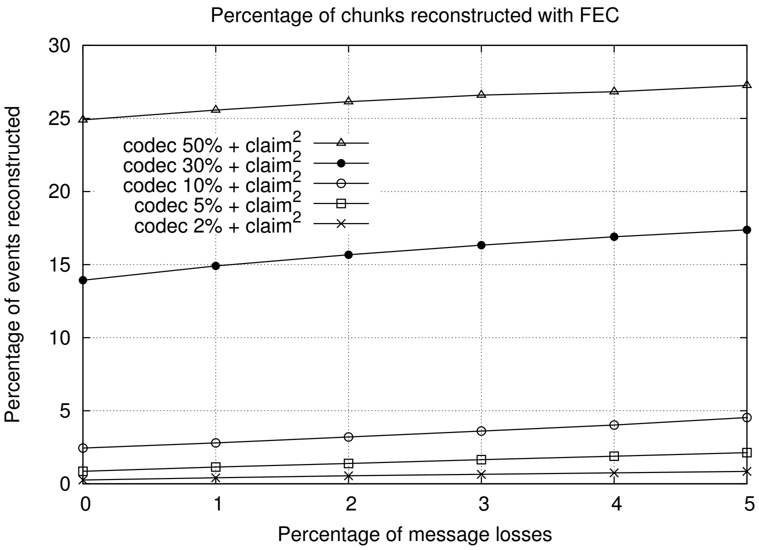


Figure 5.10: The percentage of chunks recovered with erasure coding increases similarly for the different FEC values and the FEC recovery grows as the percentage of message losses increases.

been successfully decoded). The plot in Figure 5.12 shows that such a variant is significantly less robust than *Codec* when the available bandwidth is limited. The continual requests for already decoded chunks cause its performance to drop when the available bandwidth is below 780 kbps. *Codec*, on the other hand, is almost unaffected by the bandwidth constraint up to approximately 760 kbps.

The plot also shows the percentage of chunks reconstructed by *Codec* in the same conditions. Decreasing values of available bandwidth causes an increase in the number of messages dropped by the bandwidth limiter, which in turn causes a larger proportion of chunks to be recovered by the *Codec* mechanism.

5.2.8 Freeriders

Freeriders are nodes that want to benefit from the system without contributing their share of work or resources. In the context of gossip-based streaming we can distinguish two classes of freeriders: *(i) passive freeriders* that do not propose chunks, and *(ii) active freeriders* that actively discard requests.

Passive freeriders will never serve any node since they never get any requested chunk. They can also be seen as nodes that simply cannot have outgoing communication, either deliberately or because of network constraints (e.g., firewall, closed ports). Passive freeriders benefit from the system as they receive propose messages and thus request data from other nodes while not contributing. This

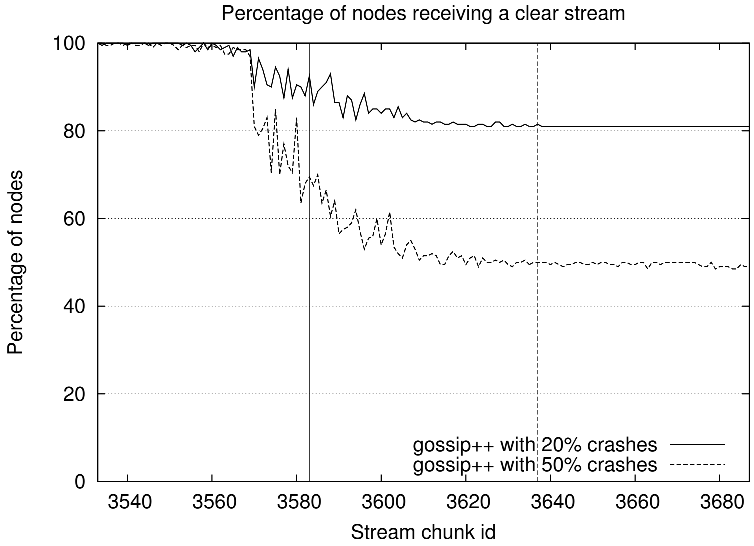


Figure 5.11: While 20% (resp. 50%) of the nodes crash between chunks 3583 and 3637 (700 ms duration), the percentage of nodes that receive each chunk starts to drop from chunk 3570 and the drop lasts less than 1 s.

causes the average fanout of the system to be lowered since all passive freeriders act as if they had a fanout of 0.

Active freeriders, instead, follow the protocol until they are expected to contribute. Once they are requested to serve chunks, they decide not to serve what was requested. This implies that a fraction of the many duplicates of propose messages will not lead to subsequent serves. In other words, nodes requesting chunks from active freeriders will not be served, just as if they requested a chunk from a node that crashed.

In the following, we evaluate the ability of gossip++ to tolerate both types of freeriders while providing all nodes with a clear stream. Clearly, this is only possible if non freeriding nodes are allowed some extra bandwidth to compensate for the bandwidth that the freeriding nodes are not using. For this reason we ran the following experiments with a larger upload bandwidth of 1000 kbps.

Passive Freeriders Figure 5.13 shows the performance obtained by gossip++ with variable percentages of passive freeriders. In this set of experiments we also consider a second variant of *Codec*, called *Codec*②. Specifically, as soon as a node is able to reconstruct a group, *Codec*② injects the reconstructed source chunks it did not receive *regularly*, and *re-encodes* the k source chunks into c encoded chunks that it also injects in the dissemination process (step (iii) in Figure 5.2).

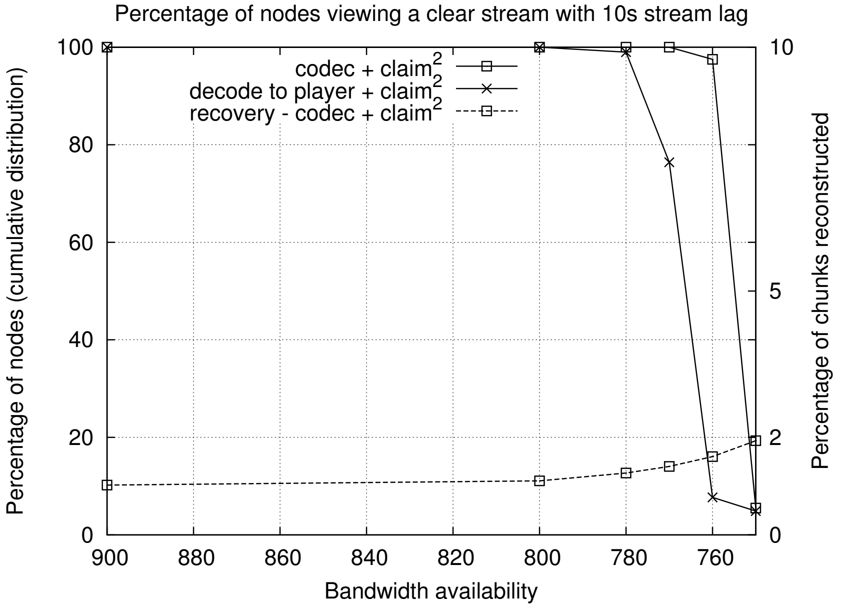


Figure 5.12: By decreasing the available bandwidth of nodes, nodes tend to lose more messages due to bursts and therefore need both *Codec* and *Claim* to deliver a clear stream. Once the available bandwidth gets below 770 kbps not all nodes can view a clear stream with gossip++ anymore. Interestingly, *Codec* is more and more effective in avoiding requests and re-requests until the system collapses.

The plot in Figure 5.13 shows that both *Codec* and its variant *Codec*② are effective in managing up to 20% of freeriding nodes. However, *Codec*② is slightly more effective with passive freerider percentages above 20%. The reason for this performance difference is that the injection mechanism of *Codec*② is able to compensate for the decrease in effective fanout resulting from the freeriding behavior. In other words, nodes injecting a reconstructed chunk into the protocol create duplicate advertisements that would not have otherwise existed.

The third line in Figure 5.13 complements these results by showing the average amount of data that non-freeriding nodes attempt to send. It is interesting to observe that the slope of the line increases dramatically as soon as the line crosses the value of 1000 kbps. At this point, the bandwidth limiter starts dropping messages, causing *Claim* to issue more and more re-requests, which in turn are more and more likely to be dropped. The data that nodes attempt to send therefore increases without a corresponding increase in the data that is actually sent. This causes the dramatic performance decrease occurring when the bandwidth line crosses the 1000 kbps threshold.

It is also worth observing that approximately 20% of freeriding nodes can be

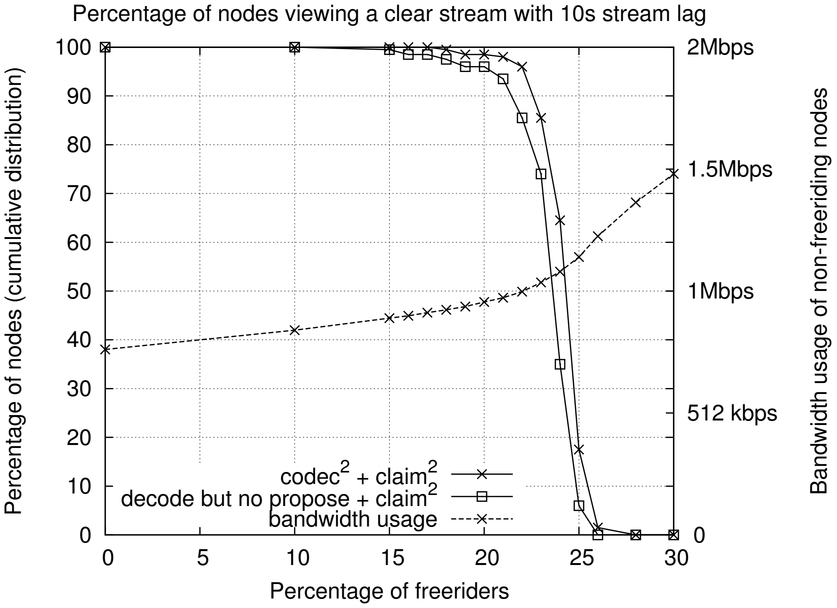


Figure 5.13: Increasing the percentage of freeriders has the effect of decreasing the average fanout. *Codec*② performs slightly better than *Codec.*

tolerated with approximately 20% of slack in the upload bandwidth: 1000 kbps instead of 800 kbps. This highlights the scalability in terms of bandwidth consumption of the combination *Codec*② + *Claim.*

Active Freeriders Figure 5.14 shows instead the performance obtained by gossip++ in the presence of variable percentages of active freeriders. Interestingly enough, in this case the performances of *Codec* and *Codec*② exhibited no differences. However, the plot shows a distinction between *Claim* and a standard retransmission mechanism such as ARQ [FW02], labeled as "retransmission" in the plot (Algorithm 3.3, page 38). Specifically, while *Claim*'s retransmission mechanism chooses to contact any of the nodes that offered a propose message for the desired chunk, the standard approach repeatedly requests a single node for each missing chunk. This means that, if the proposing node is a freerider, the standard mechanism will be unable to obtain the chunk which will instead be quickly obtained by *Claim.*

The results confirm this reasoning. The standard retransmission mechanism is unable to tolerate any amount of active freeriders despite the slack in available bandwidth. On the other hand, it keeps using the baseline bandwidth of approximately 800 kbps regardless of the percentage of freeriders. This causes its performance to drop to 0 with as little as 4% of active freeriding nodes.

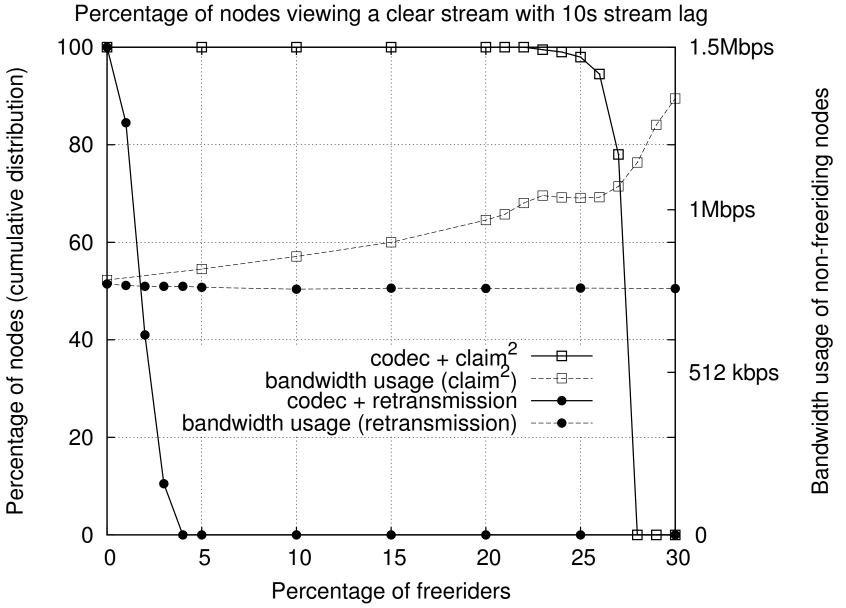


Figure 5.14: While increasing the percentage of freeriders, *Claim* is able to shift the contribution that freeriders should provide to non-freeriding nodes, providing good performance as long as their bandwidth usage remains under the bandwidth cap. A simple retransmission mechanism, on the other hand, is not able to shift the load to non-freeriding nodes. Its bandwidth usage therefore remains constant and significantly lower than the allowed bandwidth.

Claim on the other hand is very effective in having non-freeriding nodes compensate for up to 25% of freeriding nodes. The plot shows that the desired bandwidth increases quasi linearly until it reaches the threshold of 1000 kbps, i.e., the bandwidth cap. After the threshold is hit, there is first a short plateau during which performance decreases only slightly and then a sudden increase in required bandwidth which causes an equally sudden decrease in performance because the desired bandwidth is above the available limit.

5.3 Summary

In this chapter we presented gossip++, an integration to gossip consisting of *Codec*, a FEC encoding mechanism, and *Claim*, a retransmission approach leveraging gossip duplication. Gossip++ significantly improves the performance of plain gossip, making gossip-based video streaming possible in realistic scenarios.

Our experiments showed that plain gossip is not efficient in delivering large content in large-scale systems, especially in scenarios where bandwidth is constrained and in presence of message loss. Gossip++ on the other hand is able to provide nodes with a clear stream in the presence of up to 5% message loss, and up to 20% of freeriding nodes with a proportional slack in the average bandwidth capability.

The work we presented in this chapter aims to bring clarity in the design of reliable gossip-based streaming systems for real-world environments. By means of thorough experiments we demonstrated that gossip alone is unable to offer satisfactory performance in the context of video streaming applications. Moreover, we showed that applying FEC or retransmission separately is far from being an effective solution to the streaming problem. Rather, the two mechanisms must be carefully combined in order to provide a scalable streaming solution.

We also introduced a novel retransmission mechanism called *Claim*, which is explicitly designed to leverage the redundancy that is inherent in gossip dissemination. We showed that, when combined with *Codec*, *Claim* is effective in building a scalable streaming system in which correctly operating nodes are able to compensate for the presence of a significant percentage of freeriders.

Still, the proposed mechanisms do not account for heterogeneity in capabilities, since all nodes have to contribute the same amount. We have shown that gossip++ is able to accommodate freeriding behaviors in a proportion that is similar to the slack of bandwidth. Nonetheless, if there exists slack in bandwidth, we would like to use it in order to increase the stream quality rather than potentially not using it in the absence of freeriders, and gossip++ cannot do anything in the case of scarce bandwidth, i.e., when the stream rate is very close to the average outgoing capability. Therefore, the following chapters will focus on *(i)* devising an algorithm that can adapt to heterogeneous capabilities and *(ii)* complementing the gossip dissemination algorithm with a protocol for detecting and expelling freeriders so that no available bandwidth is wasted.

One of the strangest things about life is that the poor, who need money the most, are the very ones that never have it.

Finley Peter Dunne

6 Heterogeneous Gossip: HEAP

The variety of devices available today leads to the creation of systems composed of highly heterogeneous nodes. One aspect of this heterogeneity is the upload bandwidth available to nodes. Nodes on corporate networks generally have much wider bandwidths than nodes using cheaper home-based connections. Similarly, the bandwidth available to mobile devices such as Internet-enabled mobile phones depends on the cells available in the area as well as on the number of users connected to those cells. In essence, assuming that the download bandwidth of each node is larger than the stream rate s, the collaborative task that seeks to provide the stream to all nodes must adapt to the upload bandwidth heterogeneity of participating nodes.

Assume a stream of rate s produced by the source A and n nodes to broadcast the stream to, we have seen in Chapter 2 that s is upper bounded by $\left(A_{up} + \sum^{n} u_{i,up}\right)/n$ when A_{up} is large enough not to be considered the bottleneck. In a system where the n nodes can devote very different amounts of resources to the system, e.g., $u_{1,up} \ll u_{i,up} \ll u_{n,up}$, load-balancing, i.e., asking each node (except the source) to contribute the same amount s will overload the nodes with capabilities lower than s and underutilize the nodes with higher capabilities.

Given a stream rate s and $m < n$ nodes with an upload bandwidth smaller than s, the total contribution C_n of the n nodes is therefore $C_n = (n-m)s + \sum u_{i,up}$ $(\forall u_{i,up} < s)$. Since all nodes need to receive the stream, the total demand is $D = D_n = ns$, whereas the total contribution including the source is $C = C_n + A_{up}$. A problem arises as soon as C becomes smaller than D, that is, when the source is not able to compensate for the lack of contribution of the m nodes or the underutilization of the $n - m$ nodes.

To be very conservative, s can be chosen as $s = \min(u_{i,up})$ so that all nodes can contribute such a rate, thus $C = C_n + A_{up}$ with $C_n = ns$. The contribution C of the source and all nodes is thus always larger than the demand $D_n = ns$. Choosing s this way is thus very far from the optimal rate $\left(A_{up} + \sum^n u_{i,up}\right)/n$.

In order to maximize s, the nodes must therefore all contribute an amount that is proportional to their upload bandwidth, breaking the inherent load-balancing property of gossip where all nodes have the same gossip period and fanout, and therefore are expected to contribute similarly to the system.

A Case for Adaptation Consider a stream of 600 kbps produced by a single source and disseminated to 270 PlanetLab nodes. We present the results with two different distributions: dist1 is composed of 5% of nodes with 3 Mbps upload bandwidth, 10% of nodes with 1 Mbps upload bandwidth and 85% of nodes with 512 kbps upload bandwidth whereas with dist2, all nodes have an upload bandwidth of 691 kbps (similar to the average of dist1). Several fanouts are tested given the two distributions of upload capabilities, in Figure 6.1.

The major reason for the mixed behavior of gossip in a heterogeneous setting is its homogeneous and load-balanced nature. All nodes are supposed to disseminate the same number of messages for they rely on the same fanout and gossip period. However, this uniform distribution of load ignores the intrinsic heterogeneous nature of large-scale distributed systems where nodes may exhibit significant differences in their capabilities. Interestingly, and as conveyed by our experiments (Section 4.2.1, Section 6.2 and pointed out in [DXL+06]), a gossip protocol does indeed adapt to heterogeneity to a certain extent. Nodes with high bandwidth gossip rapidly, thus get pulled more often and can indeed sustain the overload to a certain extent. Nevertheless, as the bandwidth distribution gets tighter (closer to the stream rate) and more skewed (rich nodes get richer whereas poor nodes get poorer), there is a limit on the adaptation that standard homogeneous gossip can achieve.

Heterogeneous Gossip Echoing recent work, including [BRS06, DXL+06, SBR06,VYF06,VF07,LXQ+08], we recognize the need to account for the heterogeneity between nodes in order to achieve a more effective dissemination. This poses important technical challenges in the context of a gossip-based streaming application. First, an effective dissemination protocol needs to dynamically track and reflect the changes of available bandwidth over time. Second, the robustness of gossip protocols heavily relies on the proactive and uniform random selection of new target nodes: biasing this selection could impact the average quality of dissemination and the robustness to churn. Finally, gossip is simple

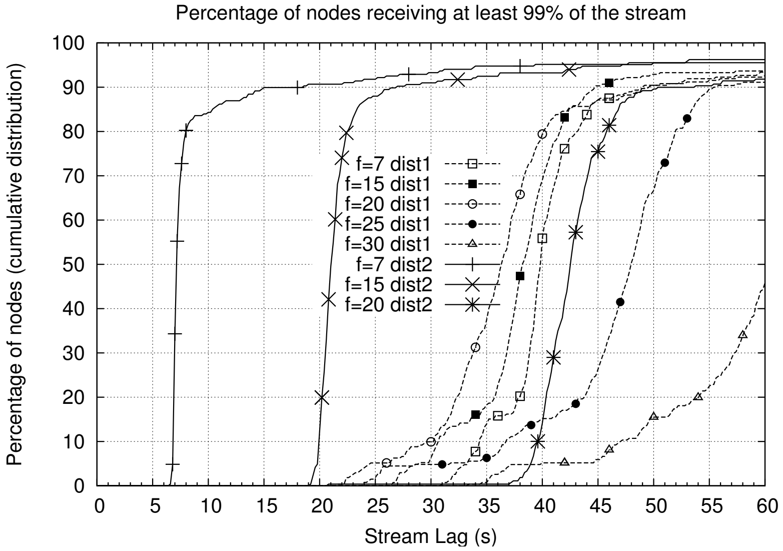


Figure 6.1: When constraining the upload capability in a heterogeneous manner (dist1), the stream lag of all nodes significantly deteriorates. Adjusting the fanout (e.g., between 15 and 20) slightly improves the stream lag but a blind fanout increase (e.g., if it goes over 25) degrades performance. Moreover, the good fanout range in this case (fanouts of 15, 20 in dist1) reveals being inadequate with a different distribution (dist2) having the same average upload capability. With dist2, a fanout of 7 is optimal and much more effective than fanouts of 15 and 20.

and thus easy to deploy and maintain; sophisticated extensions that account for heterogeneity could improve the quality of the stream but would render the protocol more complex and thus less appealing [Ham07, Zho09].

We propose a new gossip protocol, called *HEAP*, *HEterogeneity-Aware Gossip Protocol*, whose simple design is the result of two observations. First, mathematical results on epidemics and empirical evaluations of gossip protocols convey the fact that the robustness of the dissemination is ensured as long as the *average* of all fanouts is in the order of $\ln(n)$ [KMG03] (assuming the source has at least a fanout of 1). This is crucial because the fanout is an obvious knob to adapt the contribution of a node and account for heterogeneity. A node with an increased (resp. decreased) fanout will send more (resp. less) information about the chunks it can provide and in turn will be pulled more (resp. less) often. Second, using gossip dissemination, one can implement an aggregation protocol [vRBV03, JMB05] to continuously provide every node with a pretty accurate approximation of its relative bandwidth capability. Using such a protocol, HEAP dynamically leverages the most capable nodes by increasing their fanouts, while decreasing by the same proportion those of less capable nodes.

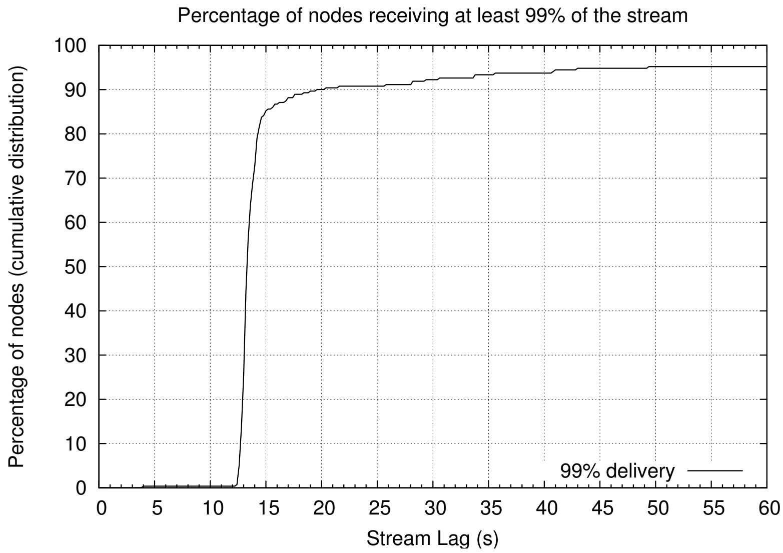


Figure 6.2: With the same constrained and heterogeneous distribution (dist1), HEAP significantly improves performance over a traditional homogeneous gossip.

HEAP preserves the simplicity and proactive (churn adaptation) nature of traditional homogeneous gossip, while significantly improving its effectiveness.

Applying HEAP in the PlanetLab context of Figure 6.1, i.e., assuming a heterogeneous bandwidth distribution conveying users using ADSL, we significantly improve the stream lag and quality using HEAP (Figure 6.2) with an average fanout of 7: 50% of nodes receive 99% of the stream with 13.3 s lag, 75% with 14.1 s and 90% with 19.5 s. More generally, we report on an exhaustive evaluation which shows that, when compared to a standard gossip, HEAP: *(i)* matches better the contribution of nodes to their bandwidth capabilities; *(ii)* enables a better usage of the overall bandwidth thus significantly improving the stream quality of all nodes and; *(iii)* significantly improves the resilience to churn.

6.1 HEAP

HEAP addresses the limitations of standard gossip by preventing congestion at low capability nodes through the adaptation of each node's workload. Consider two nodes u_i and u_j with upload capabilities $u_{i,up}$ and $u_{j,up}$. HEAP adapts the contribution of each node to its capability and thus causes the upload rate resulting from node u_i's serve messages to be $u_{i,up}/u_{j,up}$ times as large as that of node u_j.

Key to HEAP's adaptation mechanism is the fact that, in a non-congested setting, each propose message has roughly the same probability p to be accepted (thereby generating a subsequent serve message) regardless of the bandwidth capability of its sender[1]. HEAP exploits this fact to dynamically adapt the fanouts of nodes so that their contribution to the stream delivery remains proportional to their available bandwidth. Specifically, because the average number of proposals accepted in each gossip round can be computed as $p \cdot f$, f being the fanout of the proposing node, we can derive that the fanout f_{u_i} of node u_i should be $u_{i,up}/u_{j,up}$ times the fanout of node u_j.

$$f_{u_i} = \frac{u_{i,up}}{u_{j,up}} \cdot f_{u_j} \tag{6.1}$$

Preserving Reliable Dissemination Interestingly, Equation (6.1) shows that determining the ratios between the fanouts of nodes is enough to predict their average contribution. However, simply setting the fanouts of nodes to arbitrary values that satisfy Equation (6.1) may lead to undesired consequences. On the one hand, a low average fanout may hamper the ability of a gossip dissemination to reach all nodes. On the other hand, a large average fanout may unnecessarily increase the overhead resulting from the dissemination of propose messages and create undesired bursts (Chapter 4).

HEAP strives to avoid these two extremes by relying on theoretical results showing that the reliability of gossip dissemination is actually preserved as long as a fanout value of $\overline{f} = \ln(n) + c$, n being the size of the network, is ensured *on average* [KMG03], regardless of the actual fanout distribution across nodes. To achieve this, HEAP exploits a simple gossip-based aggregation protocol (see Algorithm 6.1) which provides an estimate of the average upload capability $\overline{b}$ of network nodes. A similar protocol can be used to continuously approximate the size of the system [JMB05]. For simplicity, we consider here that the initial fanout is computed knowing the system size in advance. The aggregation protocol works by having each node periodically gossip its own capability and the freshest received capabilities. We assume a node's capability $b = u_{i,up}$ is either *(i)* a maximal capability given by the user at the application level (as the maximal outgoing bandwidth the user wants to give to the streaming application) or *(ii)* computed, when joining, by a simple heuristic to discover the nodes upload capability, e.g., starting with a very low capability while trying to upload as much as possible in order to reach its maximal capability as proposed in [ZWJ$^+$06]. Each node aggregates the received values and computes an estimate of the overall average capability. Based on this estimate, each node, u_i,

[1] In practice, proposals from low capability nodes incur in higher transmission delays and thus have a slightly lower probability of acceptance, but this effect is negligible when dealing with small propose messages in a non-congested setting.

regulates its fanout, f_{u_i}, according to the ratio between its own and the average capability, i.e., $f_{u_i} = \overline{f} \cdot b/\overline{b}$.

HEAP thus consists in adding to Algorithm 3.3 a fanout adaptation mechanism and an aggregation protocol, as exposed in Algorithm 6.1.

Algorithm 6.1 HEAP protocol details.

```
Initialization:
 1: capabilities := $\varnothing$
 2: $b$ := own available bandwidth
 3: start(AggregationTimer)

Fanout Adaptation
function getFanout() returns Integer is
 4: return $b/\overline{b} \cdot \overline{f}$

Aggregation Protocol
upon (AggregationTimer mod aggPeriod) = 0 do
 5: commPartners := selectNodes($\overline{f}$)
 6: for all p $\in$ commPartners do
 7:     fresh = 10 freshest values from capabilities
 8:     send(p) [AGGREGATION, fresh]

upon receive [AGGREGATION, otherCap] do
 9: merge otherCap into capabilities
10: update $\overline{b}$ using capabilities
```

The fanout (function getFanout(), line 4 in Algorithm 6.1) is computed as $b/\overline{b} \cdot \overline{f}$ where $\overline{f}$ is the average fanout of nodes, i.e., the original fanout of Algorithm 3.3, roughly $\ln(n) + c$. The aggregation protocol consists in gossiping to $\overline{f}$ partners a set of capability values so that the average capability $\overline{b}$ of the system can be computed on each node. Note here that the regular $\overline{f}$ is chosen (and not the adapted fanout), so that every node is uniformly represented in other nodes' capabilities set for computing the average capability $\overline{b}$.

6.2 Evaluation

We evaluate HEAP on a testbed of 270 PlanetLab nodes. This includes a head-to-head comparison with a standard homogeneous gossip protocol, i.e., Algorithm 3.3. In short, we show that, when compared to a standard gossip protocol: *(i)* HEAP adapts the actual load of each node to its bandwidth capability (Section 6.2.2), *(ii)* HEAP consistently improves the streaming quality of all nodes (Section 6.2.3), *(iii)* HEAP improves the stream lag from 40% to 60% over standard gossip (Section 6.2.4), *(iv)* HEAP resists to extreme churn situations where standard gossip collapses (Section 6.2.5).

6.2.1 Experimental setup

The source generates chunks of 1316 bytes at a stream rate of 551 kbps on average. Every *window* is composed of 9 FEC encoded chunks and 101 original stream chunks resulting in an effective rate of 600 kbps.

The gossiping period of each node is set to 200 ms, which leads to grouping an average of 11.26 chunk ids per propose message. The fanout is set to 7 for all nodes in the standard gossip protocol, while in HEAP, the average fanout $\overline{f}$ is 7 across all nodes. The aggregation protocol gossips the 10 freshest local capabilities every 200 ms, costing around 1 KB/s and is thus completely marginal compared to the stream rate.

We consider three different distributions of upload capabilities, depicted in Table 6.1 and inspired from the distributions used in [ZZSY07]. The *capability supply ratio* (CSR, as defined in [ZZSY07]) is the ratio of the average upload bandwidth over the stream rate. We only consider settings in which the global available bandwidth is enough to sustain the stream rate. Yet the lower the capability ratio, the closer we stand to that limit. The ms-691 distribution was referred to as dist1 in the beginning of this chapter.

			Fraction of nodes		
Name	CSR	Average	2 Mbps	768 kbps	256 kbps
ref-691	1.15	691 kbps	0.1	0.5	0.4
ref-724	1.20	724 kbps	0.15	0.39	0.46
Name	CSR	Average	3 Mbps	1 Mbps	512 kbps
ms-691	1.15	691 kbps	0.05	0.1	0.85

Table 6.1: The reference distributions ref-691 and ref-724, and the more skewed distribution ms-691.

Each distribution is split into three classes of nodes. The skewness of an upload distribution is characterized by the various percentages of each class of nodes: in the most skewed distribution we consider (ms-691), most nodes are in the *poorest* category and only 15% of nodes have an upload capability larger than the stream rate.

In the following, we first show that HEAP adapts the contribution of nodes according to their upload capability, and then we show that HEAP provides users with a good quality stream.

6.2.2 Adaptation to Heterogeneous Upload Capabilities

We evaluate how HEAP and a standard homogeneous gossip adapt to heterogeneous upload capabilities in all three configurations. In ref-691, ref-724 and ms-691, resp. 60%, 54% and 15% of the nodes have an available bandwidth larger than the one required on average for the stream rate. We report on ref-691 in Figure 6.3, on ref-724 in Figure 6.4, and on ms-691 in Figure 6.5. The figures depict the breakdown of the contributions among the three classes of nodes. For example, the striped bar for standard gossip in Figure 6.3 means that nodes having an upload capability of 768 kbps use 97.17% of their available bandwidth.

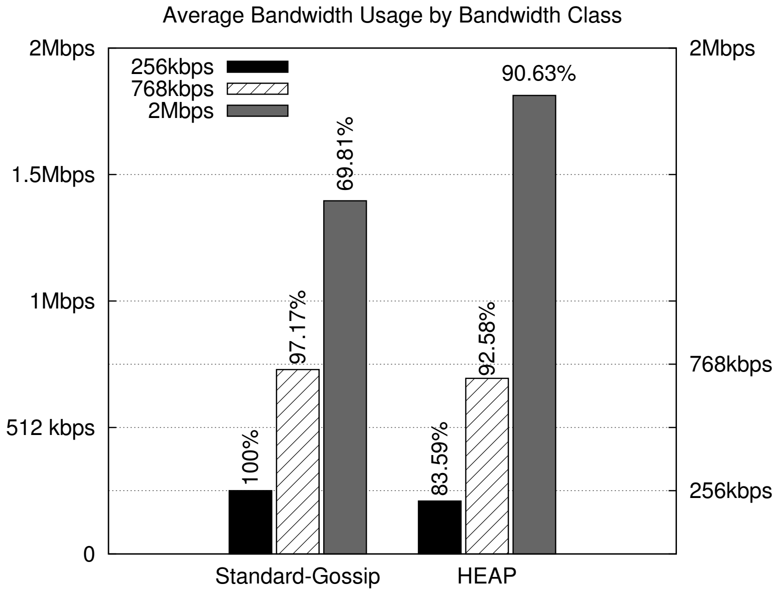


Figure 6.3: Bandwidth consumption, ref-691.

Figure 6.3 and Figure 6.4 show similar results as standard homogeneous gossip seems to be able to adapt to some extent. It is indeed interesting to observe that nodes contribute somewhat proportionally to their upload capabilities even in standard gossip. This is because of the correlation between upload capability and latency: chunk ids sent by high-capability nodes are received before those sent by lower-capability ones. Consequently, the former are requested first and serve the stream to more nodes than the latter. In addition, nodes with low capabilities are overloaded faster and therefore naturally serve fewer nodes (either because they are slower or because they are subject to more packet drops). Yet, despite this natural self-adaptation, we observe that high-capability nodes are underutilized in standard gossip.

HEAP balances the load on all nodes proportionally to their upload bandwidth, by correctly adapting their gossip fanout: all nodes approximately consume 90% of their bandwidth. This highlights how the bandwidth consumption of standard gossip and HEAP on Figure 6.3 are caused by opposite reasons: congestion of low-capability nodes in standard gossip and fanout adaptation, which prevents congestion, in HEAP.

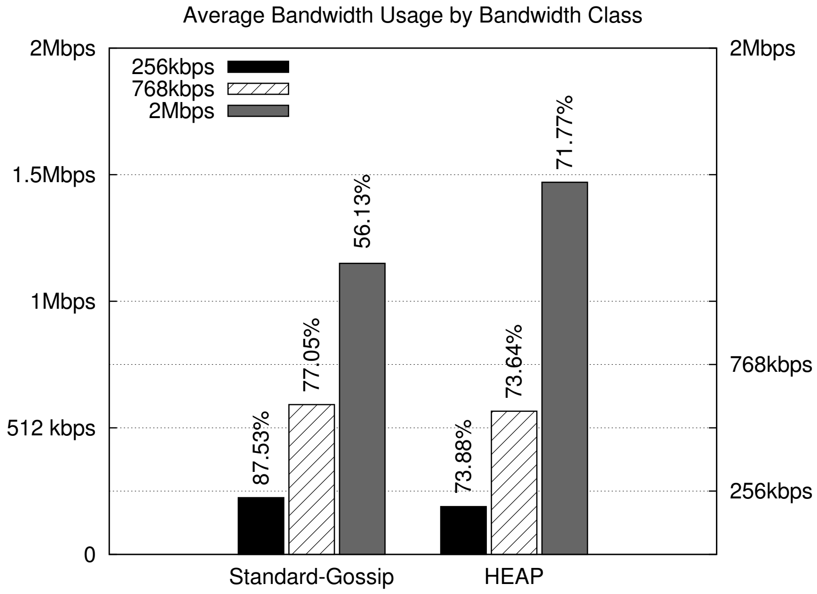


Figure 6.4: Bandwidth consumption, ref-724.

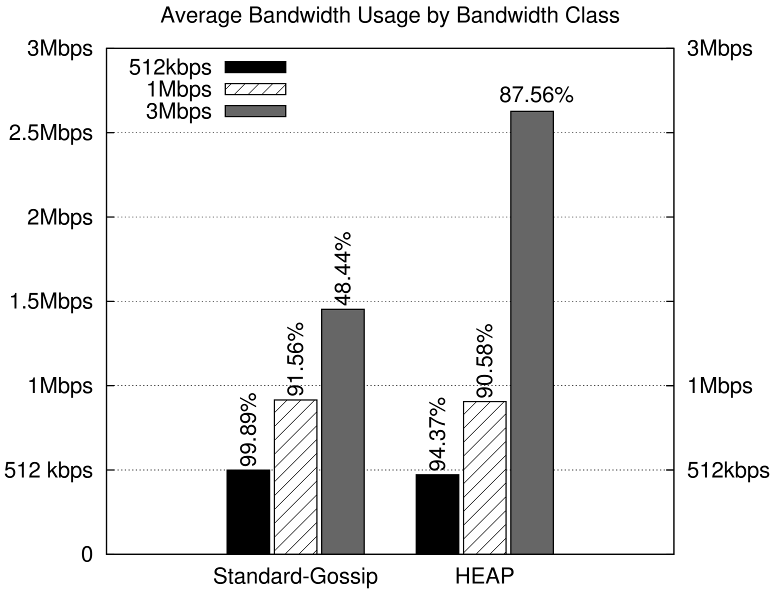


Figure 6.5: Bandwidth consumption, ms-691.

Figure 6.5 conveys the limits of the self-adaptation properties of standard gossip with an upload distribution in which only 15% of the nodes have an upload capability higher than the stream rate (ms-691). We observe that with standard gossip, the 5% of nodes with high capabilities only use 48.44% of their bandwidth because their limited fanout does not allow them to serve more nodes. In HEAP, on the other hand, the 5% of high-capability nodes can serve with up to 87.56% of their bandwidth, lowering the congestion of the low-capability nodes and providing a much better performance than standard gossip in terms of quality as we show in next section.

6.2.3 Stream Quality

Our next experiment compares the percentages of jitter-free windows received by nodes in the three considered scenarios. Results are depicted in Figures 6.6, 6.7 and 6.8. For instance, the black bar in Figure 6.6 for standard gossip indicates that nodes with low capabilities in ref-691 have only 18% of the windows that are not jittered (considering chunks received with a stream lag of up to 10 s). The same figure also shows that HEAP significantly improves this value, with low-capability nodes receiving more than 90% of jitter-free windows. This reflects the fact that HEAP allows high-capability nodes to assist low-capability ones. Results in Figure 6.7 are even more dramatic: high-capability nodes receive less than 33% of jitter-free windows in standard gossip, whereas all nodes receive more than 95% of jitter-free windows with HEAP.

Figure 6.8 clearly conveys the collaborative nature of HEAP when the global available bandwidth is higher (ref-724). The whole system benefits from the fact that nodes contribute according to their upload capability. For instance, the number of jitter-free windows that low-capability nodes obtain increases from 47% for standard gossip to 93% for HEAP. These results are complemented by Table 6.2, which presents the average delivery ratio in the jittered windows for both protocols, for each class of nodes in the three considered distributions. Again, results show that HEAP is able to provide good performance to nodes regardless of their capability classes. It should be noted, however, that the table provides results only for the windows that are jittered, jitter that appears more in standard gossip than in HEAP. This explains the seemingly bad performance of HEAP in a few cases such as for high-bandwidth nodes in ref-724.

	Standard gossip			HEAP		
upload capability	256 kbps	768 kbps	2 Mbps	256 kbps	768 kbps	2 Mbps
ref-691	63.4%	87.1%	89.3%	80.4%	77.1%	89.8%
ref-724	75.6%	88.6%	89.6%	87.9%	87.7%	64.4%
upload capability	512 kbps	1 Mbps	3 Mbps	512 kbps	1 Mbps	3 Mbps
ms-691	42.8%	56.5%	64.5%	83.7%	80.7%	90.9%

Table 6.2: Average delivery rates in windows that cannot be fully decoded.

Figure 6.9 conveys the cumulative distribution of the nodes that view the stream as a function of the percentage of jitter. For instance, the point ($x = 0.1$,

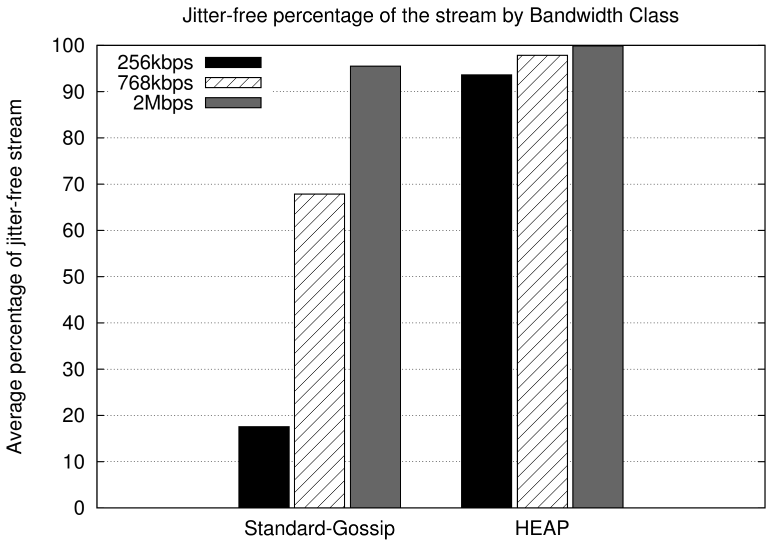


Figure 6.6: Stream quality (ref-691).

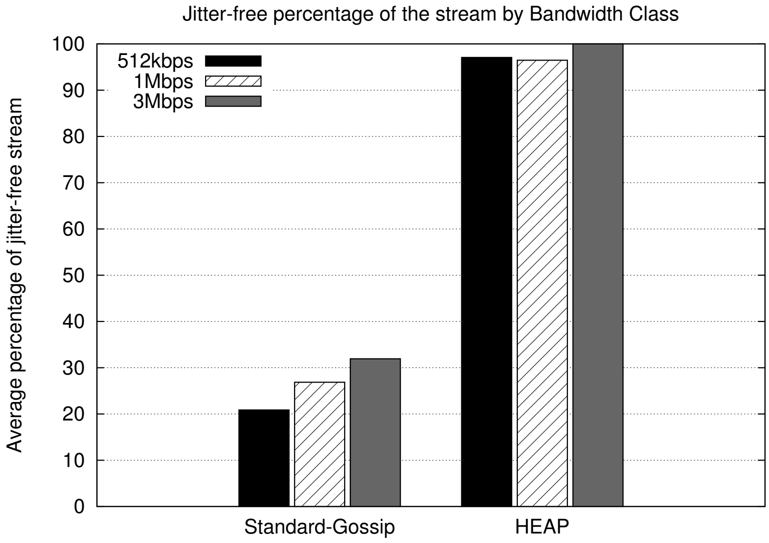


Figure 6.7: Stream quality (ms-691).

$y = 85$) on the HEAP - 10 s lag curve indicates that 85% of the nodes experience a jitter that is less than or equal to 10%. Note that in this figure, we do not differentiate between capability classes. We consider standard gossip and HEAP in two settings: offline and with 10 s lag. We present offline results in order to show that, with standard gossip, nodes eventually receive the stream. However, with a 10 s lag, standard gossip achieves very poor performance: most windows

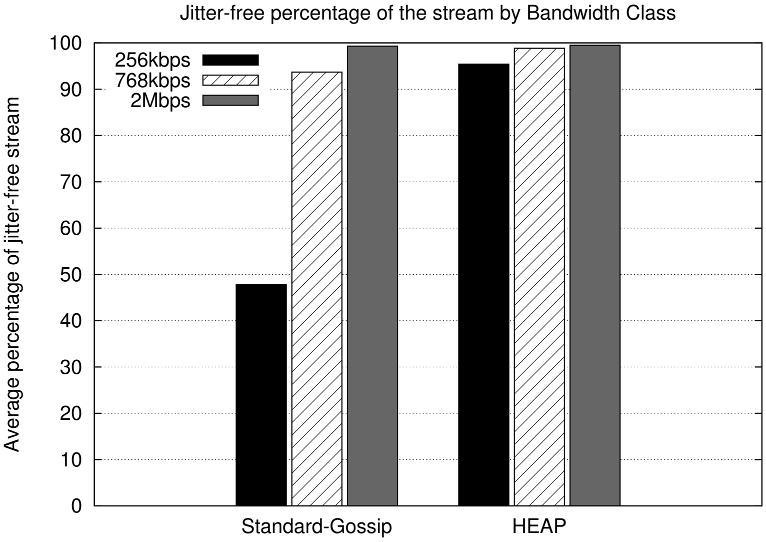


Figure 6.8: Stream quality (ref-724).

are jittered. In contrast, HEAP achieves very good performance even with a 10 s lag.

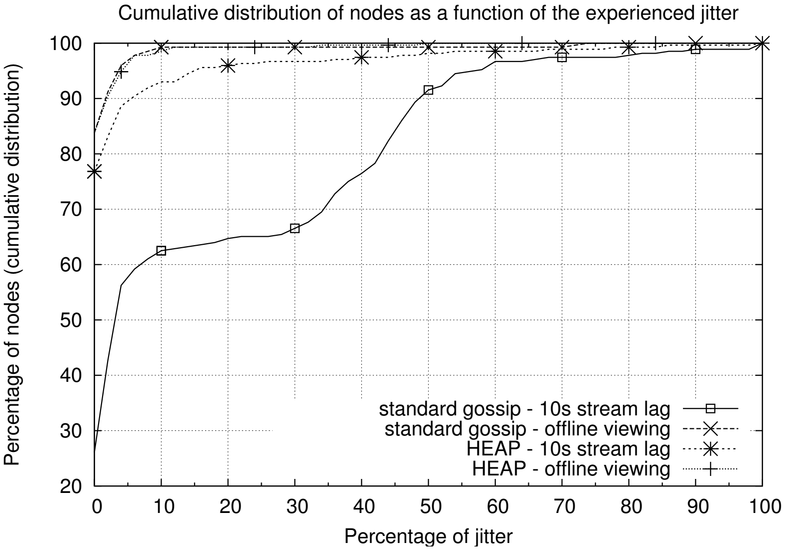


Figure 6.9: Cumulative distribution of experienced jitter (ref-691). With HEAP and a stream lag of 10 s, 93% of the nodes experience less than 10% jitter.

6.2.4 Stream lag

Next, we compare the stream lag required by HEAP and standard gossip to obtain a non-jittered stream. We report the results for ref-691 and ms-691 on Figures 6.10 and 6.11, respectively. In both cases, HEAP drastically reduces the stream lag for all capability classes. Moreover, as shown in Figure 6.11, the positive effect of HEAP significantly increases with the skewness of the distribution.

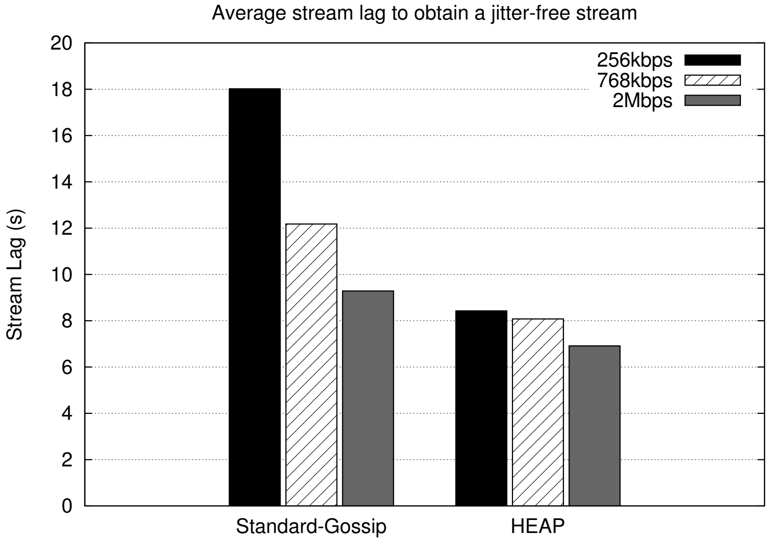


Figure 6.10: Stream lag (ref-691).

Figures 6.12 and 6.13 depict the cumulative distribution of nodes viewing the stream as a function of the stream lag, without distinguishing capability classes. We compare standard gossip and HEAP in two configurations: without jitter and with less than 1% of jitter. Sporadically, some PlanetLab nodes seem temporarily frozen, due to high CPU load and/or suffer excessive network problems explaining why neither protocol is able to deliver the stream to 100% of the nodes. Still, both plots show that HEAP consistently outperforms standard gossip. For instance, in ref-691, HEAP requires 12 s to deliver the stream to 80% of the nodes without jitter, whereas standard gossip requires 26.6 s.

Table 6.3 complements these results by showing the percentage of nodes that can view a jitter-free stream for each bandwidth class and for the three described distributions. In brief, the table shows that the percentage of nodes receiving a clear stream increases as bandwidth capability increases for both protocols. However, HEAP is able to improve the performance experienced by poorer nodes without any significant decrease in the stream quality perceived by high-bandwidth nodes.

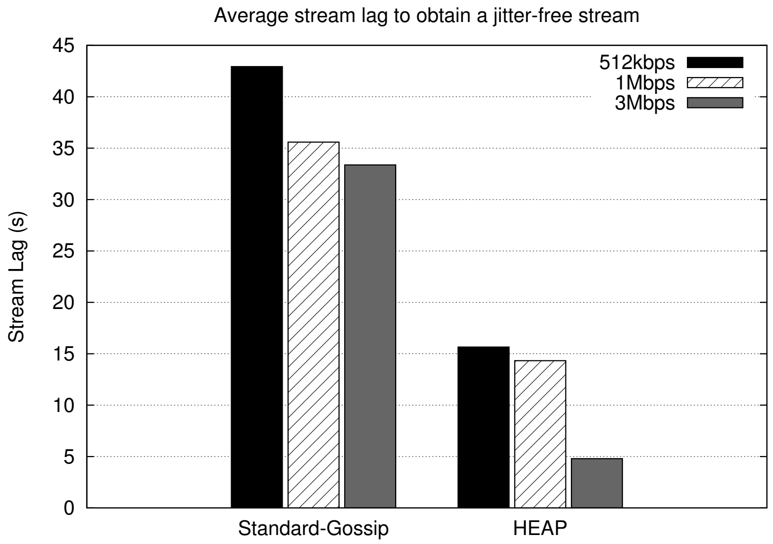


Figure 6.11: Stream lag (ms-691).

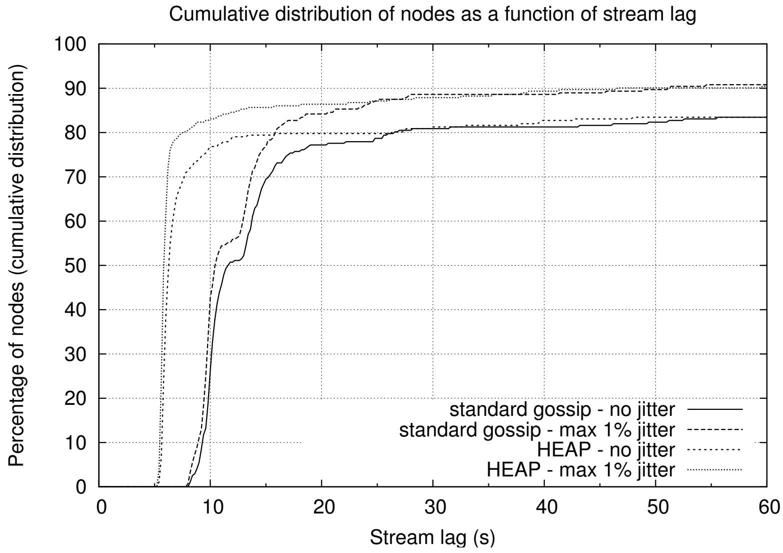


Figure 6.12: Cumulative distribution of stream lag values (ref-691).

6.2.5 Resilience to Catastrophic Failures

Finally, we assess HEAP's resilience to churn in two catastrophic-failure scenarios where 20% and 50% respectively of the nodes fail simultaneously 60 s after the beginning of the experiment. The experiments are based on the ref-691 bandwidth distribution, while the percentage of failing nodes is taken uniformly

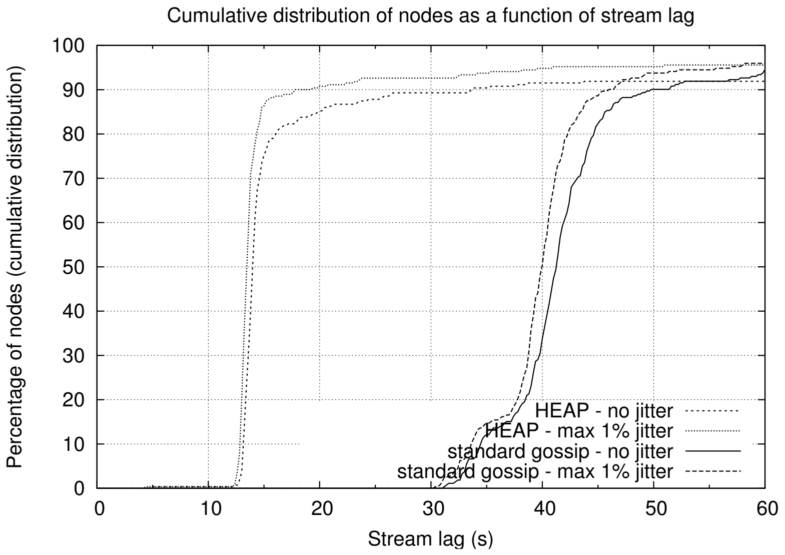


Figure 6.13: Cumulative distribution of stream lag values (ms-691).

	Standard gossip			HEAP		
bandwidth	256 kbps	768 kbps	2 Mbps	256 kbps	768 kbps	2 Mbps
ref-691 (10 s lag)	0	29.80	86.67	65.93	79.61	96.55
ref-724 (10 s lag)	0	67.52	97.73	61.95	74.34	93.02
bandwidth	512 kbps	1 Mbps	3 Mbps	512 kbps	1 Mbps	3 Mbps
ms-691 (20 s lag)	0	0	0	84.58	89.66	85.71

Table 6.3: Percentage of nodes receiving a jitter-free stream by capability class.

at random from the set of all nodes, i.e., keeping the average capability supply ratio unchanged. In addition, we configure the system so that surviving nodes learn about the failure an average of 10 s after it happened.

Figure 6.14 depicts, for each encoded window in the stream, the percentage of nodes that are able to decode it completely, i.e., without any jitter. The plot highlights once more the significant improvements provided by HEAP over standard gossip-based content dissemination. The solid line showing HEAP with a 12 s lag shows that the percentage of nodes decoding each window is always close to 100% (or to the 80% of nodes remaining after the failure) except for the chunks generated immediately before the failure. The reason for the temporary drop in performance is that the failure of a node causes the disappearance of all the chunks that it has delivered but not yet forwarded. Clearly, windows generated after the failure are instead correctly decoded by almost all remaining nodes. The plot also shows two additional lines depicting the significantly worse performance achieved by standard gossip-based dissemination.

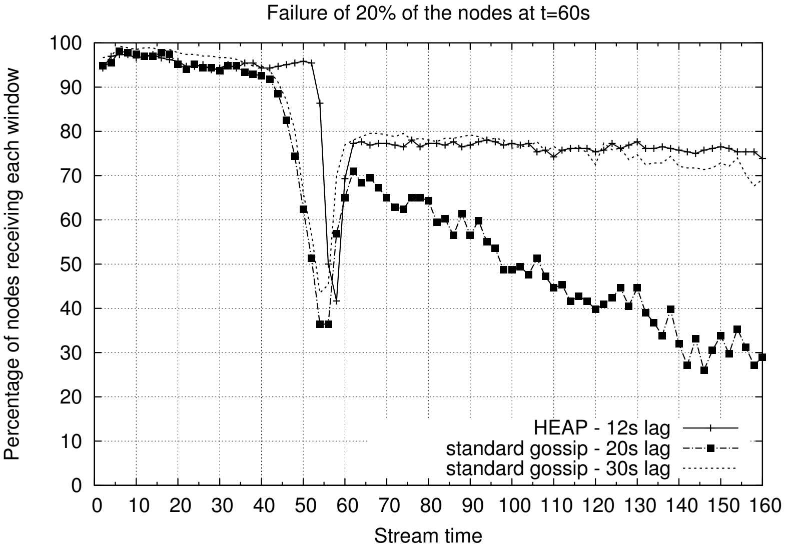


Figure 6.14: Resilience in the presence of 20% of nodes crashing.

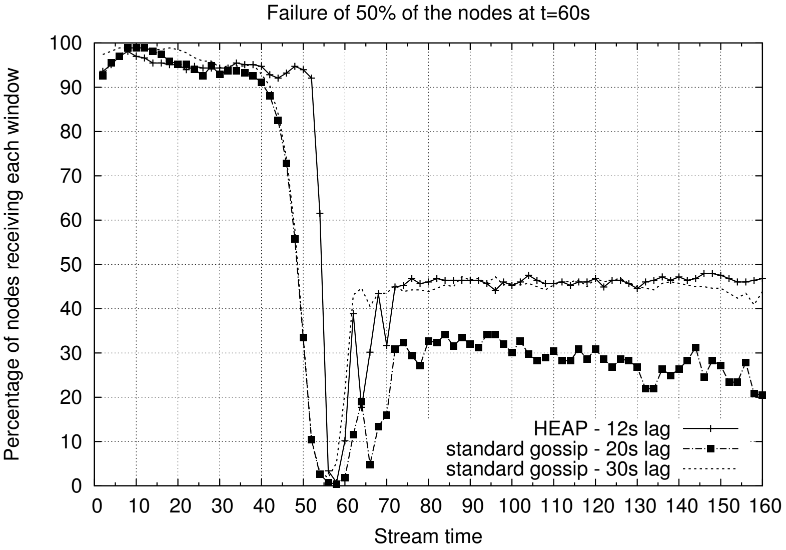


Figure 6.15: Resilience in the presence of 50% of nodes crashing.

The number of nodes receiving the stream with a 20 s lag in standard gossip is, in fact, much lower than that of those receiving it with only 12 s of lag in HEAP. Only after 30 s of lag is standard gossip able to reach a performance that is comparable to that of HEAP after 12 s. The figure also highlights that the number of chunks lost during the failure is higher in standard gossip than in HEAP (the width of the drop is larger). The reason is that in standard gossip upload queues tend to grow larger than in HEAP. Thus chunks that are lost as a result of nodes that crash span a longer time interval in standard gossip than they do in HEAP. Finally, the continuous decrease in the 20 s lag line for standard gossip shows that the delay experienced by chunks in standard gossip increases as time elapses: this is a clear symptom of congestion that is not present in HEAP.

Figure 6.15 provides similar information for a scenario in which 50% of the nodes fail simultaneously. HEAP is still able to provide the stream to the remaining nodes with a lag of less than 12 s. Conversely, standard gossip achieves mediocre performance after as many as 20 s of lag.

6.3 Alternative Solutions

The approaches of [BRS06, SBR06] propose a set of heuristics that account for bandwidth heterogeneity (and node uptimes) in tree-based multicast protocols. This leads to significant improvements in bandwidth usage. These protocols aggregate global information about the implication of nodes across trees, by exchanging messages along tree branches, in a way that relates to our capability aggregation protocol.

Multi-tree dissemination schemes split streams over diverse paths to enhance their reliability. This comes for free in gossip protocols where the neighbors of a node continuously change. In a sense, a gossip dissemination protocol dynamically provides different dissemination paths for each chunk, providing the ultimate splitting scheme. Chunkyspread accounts for heterogeneity using the SwapLinks protocol [VF06]. Each node contributes in proportion to its capacity and/or willingness to collaborate. This is reflected by heterogeneous numbers of children across the nodes in the tree.

Mesh-based systems are similar to gossip in the sense that their topology is unstructured. Some of those, namely the latest version of Coolstreaming [LXQ+08] and GridMedia [ZZSY07] dynamically build multi-trees on top of the unstructured overlay when nodes perceive they are stably served by their neighbors. Typically, every node has a *view* of its neighbors, from which it picks new partners if it observes malfunctions. In the extreme case, a node has to seek for more or different communication partners if none of its neighbors is operating properly. Not surprisingly, it was shown in [LXQ+08, LGL08] that increasing the view size has a very positive effect on the streaming quality and is more robust in case of churn. Gossip protocols like HEAP are extreme cases of these phenomena because the views they rely on keep continuously changing.

6.4 Summary

We presented HEAP, a new gossip protocol which adapts the dissemination load of the nodes to account for their heterogeneity. HEAP preserves the simplicity and proactive (churn adaptation) nature of traditional homogeneous gossip, while significantly improving its effectiveness. Experimental results on PlanetLab convey the improvement of HEAP over a standard homogeneous gossip protocol with respect to stream quality, bandwidth usage and resilience to churn. When the stream rate is close to the average available bandwidth and the capability distribution is more skewed, the improvement is even more significant.

We considered bandwidth as the main heterogeneity factor, as it is indeed crucial in the context of streaming. Other factors might reveal being important in other applications (e.g., node interests, available CPU). We believe HEAP could easily be adapted to such factors by modifying the underlying aggregation protocol accordingly. Also, we considered the choice of the fanout as the way to adjust the load of the nodes. One might also explore the dynamic adaptation of the gossip targets, the frequency of the dissemination or the memory size devoted to the dissemination.

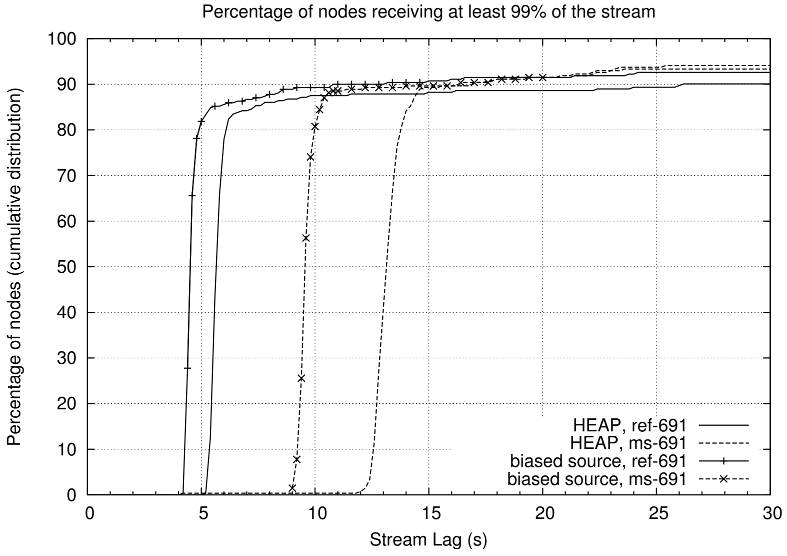


Figure 6.16: Biasing the source improves the overall stream lag by an average of 3.6 s for ref-691 and 2.3 s in ms-691.

A natural way to further improve the efficiency of the dissemination is to increase the upload capability of the source itself [PM08] or add multiple sources in the system [LXQ+08, CJW09]. A complementary way is to bias the neighbor selection towards rich nodes in the early steps of dissemination [KSSV00, CGN+04, PPKB07]. Our experiments reveal that this can be beneficial at the first step of the dissemination (i.e., from the source) but reveals not being trivial if performed in later steps. In Figure 6.16, the source picks its gossip target from the 50% highest capability nodes in the system. We observe that the overall perceived stream lag is significantly improved. High-capability nodes may see their load increase, however, they are more likely to be fed directly by the source and this significantly improves their stream lag. In addition, all nodes benefit from this bias as they have a higher probability of being fed by high capability nodes.

There are some limitations to adaptation and these provide interesting research tracks. While adapting to heterogeneity, a natural behavior is to elevate certain wealthy nodes to the rank of temporary super peers, which could potentially have a relatively large impact in case of failures. Moreover, an attacker targeting highly capable nodes could degrade the overall performance of the protocol. Likewise, the very fact that nodes advertise their capabilities may trigger freeriding vocations, where nodes would pretend to be poor in order not to contribute to the dissemination. This last point is the subject of the next chapter, on detecting freerider behaviors in gossip protocol.

Love all, trust a few.

William Shakespeare

7 Lightweight Freerider-Tracking Protocol: LiFT

The various gossip-based protocols presented in this thesis are *asymmetric*: nodes propose chunk identifiers to a dynamically changing random subset of other nodes. These, in turn, request packets of interest, which are subsequently pushed by the proposer. The nodes proposing and finally pushing content have no immediate return of contribution from the nodes they serve.

The efficiency of such protocols highly relies on the willingness of participants to collaborate, i.e., to devote a fraction of their resources, namely their upload bandwidth, to the system. Yet, some of these participants might be tempted to *freeride* [AH00, KSTT04, LCW$^+$06, LCM$^+$08], i.e., not contribute their fair share of work, especially if they could still benefit from the system. Freeriding is common in large-scale systems deployed in the public domain [AH00] and may significantly degrade the overall performance in bandwidth-demanding applications such as streaming [KSTT04]. In addition, freeriders may collude, i.e., collaborate to decrease their individual and common contribution to the system and cover each other up to circumvent detection mechanisms.

By using the Tit-for-Tat (TfT) incentives (inspired from file-sharing systems [Coh03]), TfT-based content dissemination solutions (e.g., [LCW$^+$06, PPKB07, LCM$^+$08]) force nodes to contribute as much as they benefit by means of balanced *symmetric* exchanges. As we review in related approaches (Section 7.5), those systems do not perform as well as *asymmetric* systems in terms of efficiency and scalability.

In practice, many proposals (e.g., [DXL+06, VYF06, ZZSY07, LXQ+08]) consider instead asymmetric exchanges where nodes are supposed to altruistically serve content to other nodes, i.e., without asking anything in return, where the benefit of a node is not directly correlated to its contribution but rather to the global *health* of the system. The correlation between the benefit and the contribution is not immediate. However, such correlation can be artificially established, in a coercive way, by means of verification mechanisms that ensure that nodes which do not contribute their fair share do not benefit anymore from the system. Freeriders are then defined as nodes that decrease their contribution as much as possible while keeping the probability of being expelled low.

We consider a generic three-phase gossip protocol (Algorithm 3.3) where data is disseminated following an asymmetric push scheme. In this context, we propose LiFT, a lightweight mechanism to track freeriders. To the best of our knowledge, LiFT is the first protocol to secure asymmetric gossip protocols against possibly colluding freeriders.

At the core of LiFT lies a set of deterministic and statistical distributed verification procedures based on *accountability* (i.e., each node maintains a digest of its past interactions). Deterministic procedures check that the content received by a node is further propagated following the protocol (i.e., to the right number of nodes within short delay) by cross-checking nodes' logs. Statistical procedures check that the interactions of a node are evenly distributed in the system using statistical techniques. Interestingly enough, the high dynamic and strong randomness of gossip protocols, that may be considered as a difficulty at first glance, happens to help tracking freeriders. Effectively, LiFT exploits the very fact that nodes pick neighbors at random to prevent collusion: since a node interacts with a large subset of the nodes, chosen at random, this drastically limits its opportunity to freeride without being detected, as it prevents it from deterministically choosing colluding partners that would cover up its bad behavior.

LiFT is lightweight as it does not rely on heavyweight cryptography and incurs only a low overhead in terms of bandwidth. This overhead can be dynamically adjusted and potentially reduced to zero when the system is healthy. In addition, LiFT is fully decentralized as nodes are in charge of verifying each others' actions and monitoring each others' behavior. Finally, LiFT provides a good probability of detecting freeriders while keeping the probability of false positive, i.e., inaccurately classifying a correct node as a freerider, very low.

To evaluate LiFT, we give analytical results backed up with simulations, providing means to set up the parameters of LiFT in a real environment. Additionally, we deployed LiFT on PlanetLab, where a stream of 674 kbps is broadcast to 300 PlanetLab nodes having their upload bandwidth capped to 1000 kbps. In the presence of freeriders, the health of the system (i.e., the proportion of nodes able to receive the stream in function of the stream lag) degrades significantly compared to a system where all nodes follow the protocol. Figure 7.1 shows a clear drop between the plain line (no freeriders) and the dashed line (25% of freeriders). With LiFT and assuming that freeriders keep their probability of being expelled lower than 50%, the performance is close to the baseline.

In this context, LiFT incurs a maximum network overhead of only 8%. When freeriders decrease their contribution by 30%, LiFT detects 86% of the freeriders and wrongly expels 12% of honest nodes, after only 30 seconds. Most of wrongly expelled nodes deserve it, in a sense, as their actual contribution is smaller than required. However, this is due to poor capabilities, as opposed to freeriders that deliberately decrease their contribution.

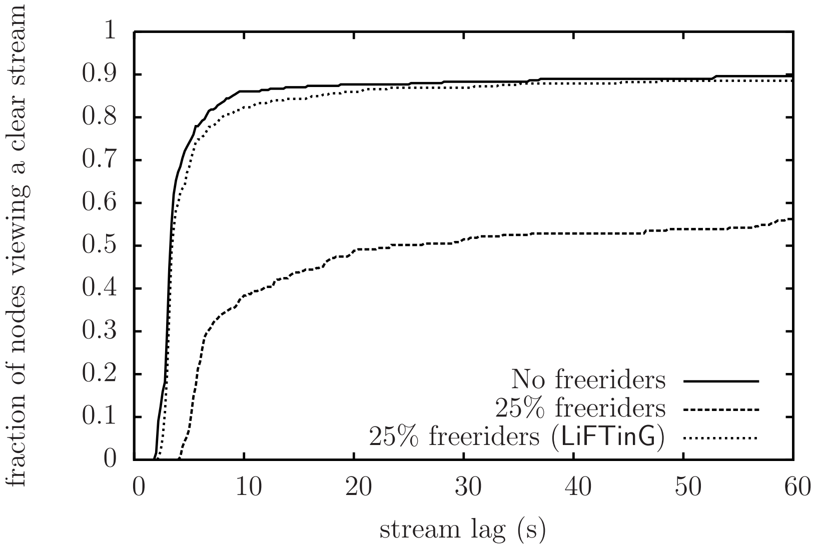


Figure 7.1: System efficiency in the presence of freeriders.

Gossip protocols are almost not impacted by crashes [KMG03, EGH$^+$03]. However, high-bandwidth content dissemination with gossip clearly suffers more from freeriders than from crashes (Chapter 5). When content is pushed in a single phase, a freerider is equivalent to a crashed node. In three-phase protocols, crashed nodes do not provide upload bandwidth anymore but they do not consume any bandwidth either, as they do not request content from proposers after they crash. On the contrary, freeriders decrease their contribution, yet keep requesting content.

7.1 Freeriding

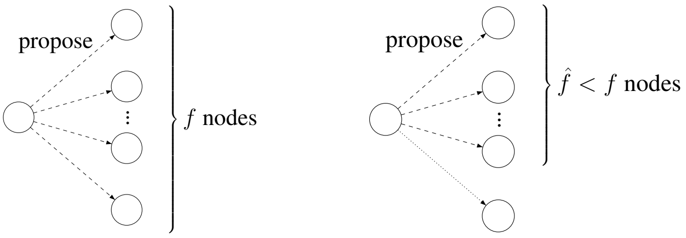
Nodes are either honest or freeriders. Honest nodes strictly follow the protocol, including the verifications of LiFT. Freeriders allow themselves to deviate from the protocol in order to minimize their contribution while maximizing their benefit. In addition, freeriders may adopt any behavior not to be expelled, including lying to verifications, or cover up colluding freeriders' bad actions. Note that under this model, freeriders do not wrongfully accuse honest nodes. Effectively, making honest nodes expelled *(i)* does not increase the benefit of freeriders, *(ii)* does not prevent them from being detected, i.e., detection is based solely on the suspected node's behavior regardless of other nodes' behaviors (details in Section 7.3.1), and finally *(iii)* leads to an increased proportion of freeriders, degrading the benefit of all nodes. This phenomenon is known as the *tragedy of the commons* [Har68]. We denote by m the number of freeriders.

Freeriders may deviate from the gossip protocol in three ways: *(i)* bias the partner selection, *(ii)* drop messages they are supposed to send, or *(iii)* modify the content of the messages they send. In the sequel, we exhaustively list all possible attacks in each phase of the protocol, discuss their motivations and impacts, and then extract and classify those that may increase the individual interest of a freerider or the common interest of colluding freeriders. Attacks that require or profit to colluding nodes are denoted with a '$\star$'. The analysis on the possible freeriding attacks to the three-phase protocol is at the core of LiFT.

7.1.1 Propose phase

During the first phase, a freerider may *(i)* communicate with less than f nodes, *(ii)* propose less chunks than it should, *(iii)* select as communication partners only a specific subset of nodes, or *(iv)* reduce its proposing rate.

(i) **Decreasing fanout** By proposing chunks to $\hat{f} < f$ nodes per gossip period, as illustrated in Figure 7.2, the freerider trivially reduces the potential number of requests, and thus the probability of serving chunks. Therefore, its contribution in terms of the amount of data uploaded is decreased. Setting $\hat{f}$ to 0 corresponds to the passive freeriders of Chapter 5.

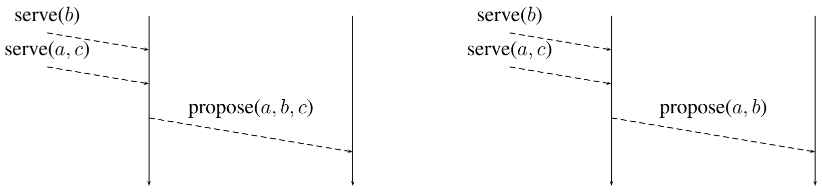


(a) Honest node (b) Freerider

Figure 7.2: A freerider communicates with $\hat{f} < f$ partners.

(ii) **Invalid proposal** A proposal is valid if it contains every chunk received in

the last gossip period. Proposing only a subset of the chunks received in the last period, as illustrated in Figure 7.3, obviously decreases the number of requested chunks. However, a freerider has no interest in proposing chunks it does not have since, contrarily to symmetric TfT-based protocols, uploading chunks to a node does not imply that the latter sends chunks in return. In other words, proposing more (and possibly fake) chunks does not increase the benefit of a node and does thus not need to be considered.

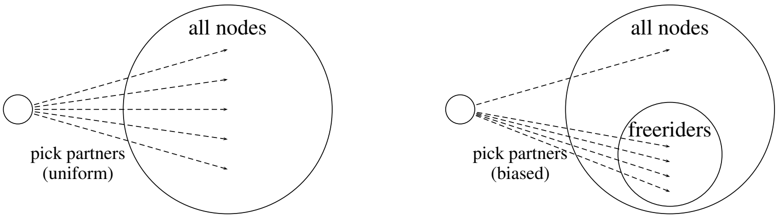


(a) Honest node

(b) Freerider

Figure 7.3: A freerider deliberately removes some chunks (c here) from its proposal.

(iii) **Biasing the partners selection (⋆)** Considering a group of colluding nodes, a freerider may want to bias the random selection of nodes to privilege its colluding partners, so that the group's benefit increases, as illustrated in Figure 7.4.

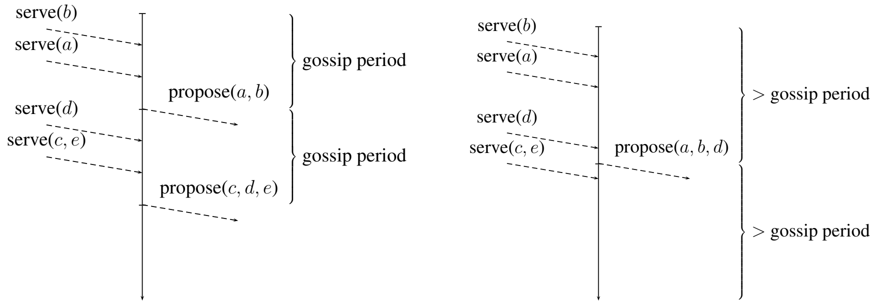


(a) Honest node

(b) Colluding freeriders

Figure 7.4: An honest node picks communication partners uniformly at random from the set of all nodes whereas a freerider biases the partner selection to pick mainly colluding nodes.

(iv) **Increasing the gossip period** A freerider may increase its gossip period T_g, leading to less frequent proposals advertising more, but "older", chunks per proposal as illustrated in Figure 7.5. This implies a decreased interest of the requesting nodes and thus a decreased contribution for the sender. This is due to the fact that an old chunk has a lower probability of being of interest as it becomes more replicated over time.



(a) Honest node (b) Freerider

Figure 7.5: With a larger gossip period, some proposed chunks are unlikely to be requested (e.g., a and b here).

7.1.2 Pull Request Phase

Nodes are expected to request only chunks that they have been proposed. A freerider would increase its benefit by opportunistically requesting extra chunks (even from nodes that did not propose these chunks). The dissemination protocol itself prevents this misbehaving by automatically dropping such requests, i.e., a node only pushes chunks that it effectively proposed.

7.1.3 Serving Phase

In the serving phase, freeriders may *(i)* send only a subset of what was requested or *(ii)* send junk. The first attack obviously decreases the freeriders' contribution as they serve fewer chunks than they are supposed to. However, as we mentioned above, in the considered asymmetric protocol, a freerider has no interest in sending junk data, since it does not receive anything in return of what it sends. The freeriders that do not respond to requests at all correspond to the active freeriders described in Chapter 5.

7.1.4 Summary

Analyzing the basic gossip protocol in detail allowed to identify the possible attacks. Interestingly enough, these attacks share similar aspects and can thus be gathered into three classes that dictate the rationale along which our verification procedures are designed.

The first is *quantitative correctness* that characterizes the fact that a node effectively proposes to the correct number of nodes (f) at the correct rate ($1/T_g$). Assuming this first aspect is verified, two more aspects must be further considered: *causality* that reflects the correctness of the deterministic part of the protocol, i.e., received chunks must be proposed in the next gossip period, and *statistical validity* that evaluates the fairness (with respect to the distribution specified by the protocol) in the random selection of communication partners.

7.2 LiFT

LiFT is a Lightweight protocol for Freerider-Tracking that encourages nodes, in a coercive way, to contribute their fair share to the system, by means of distributed verifications. LiFT consists of *(i) direct* verifications and *(ii) a posteriori* verifications. Verifications, that require more information than what is available at the verifying node and the inspected node, are referred to as *cross-checking.* In order to control the overhead of LiFT, the frequency at which such verifications are triggered is controlled by a parameter p_{cc}, as described in Section 7.2.2. Verifications can either lead to the emission of *blames* or to *expulsion*, depending on the gravity of the misbehavior.

Direct verifications are performed regularly while the protocol is running: the nodes' actions are directly checked. They aim at checking that all chunks requested are served and that all chunks served are further proposed to a correct number of nodes, i.e, they check the *quantitative correctness* and *causality.* Direct verifications are composed of *(i)* direct checking and *(ii)* direct cross-checking.

A posteriori verifications are run sporadically. They require each node to maintain a log of its past interactions, namely a *history.* In practice, a node stores a trace of the events that occurred in the last h seconds, i.e., corresponding to the last $n_h = h/T_g$ gossip periods. The history is audited to check the statistical validity of the random choices made when selecting communication partners, namely *entropic check.* The veracity of the history is verified by cross-checking the involved nodes, namely a posteriori cross-checking.

We present the blaming architecture in Section 7.2.1 and present direct verifications in Section 7.2.2. Since freeriders can collude not to be detected, we expose how they can cover up each other's misbehaviors in Section 7.2.3 and address this in Section 7.2.4. We analyze the message complexity of LiFT in Section 7.2.5. The different attacks and corresponding verifications are summarized in Table 7.1.

Attack	**Type**	**Detection**
fanout decrease ($\hat{f} < f$)	quantitative	direct cross-check
partial propose ($\mathcal{P}$)	causality	direct cross-check
partial serve ($\lvert\mathcal{S}\rvert < \lvert\mathcal{R}\rvert$)	quantitative	direct check
bias partners selection ($\star$)	entropy	entropic check, *a posteriori* cross-check

Table 7.1: Summary of attacks and associated verifications.

7.2.1 Blaming Architecture

In LiFT, the detection of freeriders is achieved by means of a score assigned to each node. When a node detects that some other node freerides, it emits a blame message containing a *blame value* against the suspected node. Summing up the blame values of a node results in a score. For scores to be meaningful, blames emitted by different verifications should be comparable and homogeneous. In order to collect blames targeting a given node and maintain its score, each node is monitored by a set of other nodes named *managers*, distributed among the participants. Blame messages towards a node are sent to its managers. When the score of a node p drops beyond a *fixed* threshold (the design choice of using a fixed threshold is explained in Section 7.3.1), the managers spread – through gossip – a revocation message against p making the nodes of the system progressively remove p from their view. A representation of blame messages sent to p's managers and revocation messages gossiped from those managers to other participants in case p's score goes beyond a threshold is synthesized in Figure 7.6.

The blaming architecture of LiFT is built on top of the AVMON [MG09] monitoring overlay. In AVMON, nodes are assigned a *fixed-size* set of M *random* managers *consistent* over time which make it very appealing in our setting, namely a dynamic peer-to-peer environment subject to churn with possibly colluding nodes. The fact that the number M of managers is constant makes the protocol scalable as the monitoring load at each node is independent of the system size. Randomness prevents colluding freeriders from covering each other up and consistency enables long-term blame history at the managers. The monitoring relationship is based on a hash function and can be advertised in a gossip-fashion by piggybacking node's monitors in the view maintenance messages (e.g., exchanges of local views in the distributed peer-sampling service). Doing so, nodes quickly discover other nodes' managers – and are therefore able to blame them if necessary – even in the presence of churn. In addition, nodes can locally verify (i.e., without the need for extra communication) whether the mapping, node to managers, is correct by hashing the nodes' IP addresses, preventing freeriders from forging fake or colluding managers. In case a manager does not map correctly to a node, a revocation against the concerned node is sent.

7.2.2 Direct Verifications

In LiFT, two direct verifications are used. The first aims at ensuring that every requested chunk is served, namely a *direct check*. Detection can be done locally and it is therefore always performed. If some requested chunks are missing, the requesting node blames the proposing node by $f/|\mathcal{R}|$ (where $\mathcal{R}$ is the set of requested chunks) for each chunk that has not been delivered.

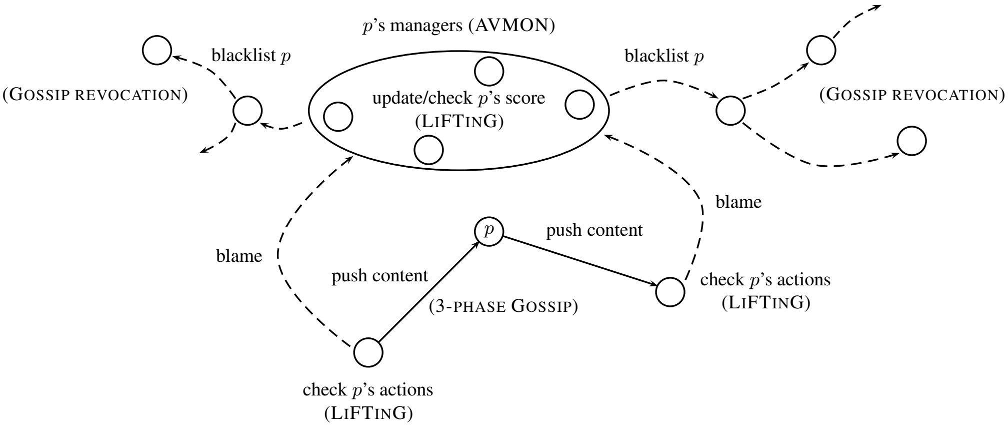


Figure 7.6: Overview of LiFT.

The second verification checks that received chunks are further proposed to f nodes within the next gossip period. This is achieved by a *cross-checking* procedure that works as follows: a node p_1 that received a chunk c_i from p_0 acknowledges to p_0 that it proposed c_i to a set of f nodes. Then, p_0 sends confirm requests (with probability p_{cc}) to the set of f nodes to check whether they effectively received a propose message from p_1 containing c_i. The f witnesses reply to p_0 with answer messages confirming or infirming p_1's acknowledgment sent to p_0.

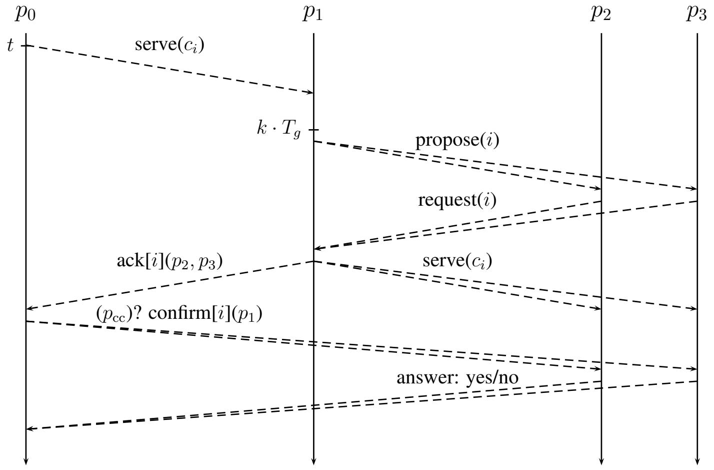


Figure 7.7: Cross-checking protocol.

Figure 7.7 depicts the message sequence composing a direct cross-checking verification (with a fanout of 2 for the sake of readability). The blaming mechanism works as follows: *(i)* if the ack message is not received, the verifier p_0

blames the verified node p_1 by f, and *(ii)* for each missing or negative answer message, p_0 blames p_1 by 1.

Since the verification messages (i.e., ack, confirm and confirm responses) for the direct cross-checking are small and in order to limit the subsequent overhead of LiFT, direct cross-checking is done exclusively with UDP. The blames corresponding to the different attacks are summarized in Table 7.2.

Attacks	**Blame values**
fanout decrease ($\hat{f} < f$)	$f - \hat{f}$ from each verifier
partial propose	1 (per invalid proposal) from each verifier
partial serve ($\|\mathcal{S}\| < \|\mathcal{R}\|$)	$f \cdot (\|\mathcal{R}\| - \|\mathcal{S}\|)/\,\|\mathcal{R}\|$ from each requester

Table 7.2: Summary of attacks and associated blame values.

Blames emitted by the direct verification procedures of LiFT are summed into a score reflecting the nodes' behaviors. For this reason, blame values must be comparable and homogeneous. This means that two misbehaviors that reduce a freerider's contribution by the same amount should lead to the same value of blame, regardless of the misbehaviors and the verification.

We consider a freerider p_f that received c chunks and wants to reduce its contribution by a factor δ ($0 \leq \delta \leq 1$). To achieve this goal, p_f can: *(i)* propose the c received chunks to only $\hat{f} = (1-\delta) \cdot f$ nodes, *(ii)* propose only a proportion $(1-\delta)$ of the chunks it received, or *(iii)* serve only $(1-\delta) \cdot |\mathcal{R}|$ of the $|\mathcal{R}|$ chunks it was requested. For the sake of simplicity, we assume that $\hat{f}, c \cdot \delta$, c/f and $\delta \cdot |R|$ are all integers. The number of verifiers, that is, the number of nodes that served the c chunks to p_f is called the *fanin* (f_{in}). On average, we have $f_{in} \simeq f$ and each node serves c/f chunks [GKM03].

We now derive, for each of the three aforementioned misbehaviors, the blame value emitted by the direct verifications.

(i) *Fanout decrease (direct cross-check):* If p_f proposes all the c chunks to only $\hat{f}$ nodes, it is blamed by 1 by each of the f_{in} verifiers, for each of the $f - \hat{f}$ missing "propose target". This results in a blame value of $f_{in} \cdot (f - \hat{f}) = f_{in} \cdot \delta \cdot f \simeq \delta f^2$.

(ii) *Partial propose (direct cross-check):* If p_f proposes only $(1-\delta) \cdot c$ chunks to f nodes, it is blamed by f by each of the nodes that provided at least one of the missing chunks. A freerider has therefore interest in removing from its proposal chunks originating from the smallest subset of nodes. In this case, its proposal is invalid from the standpoint of $\delta \cdot f_{in}$ verifiers. This results in a blame value of $\delta \cdot f_{in} \cdot f \simeq \delta \cdot f^2$.

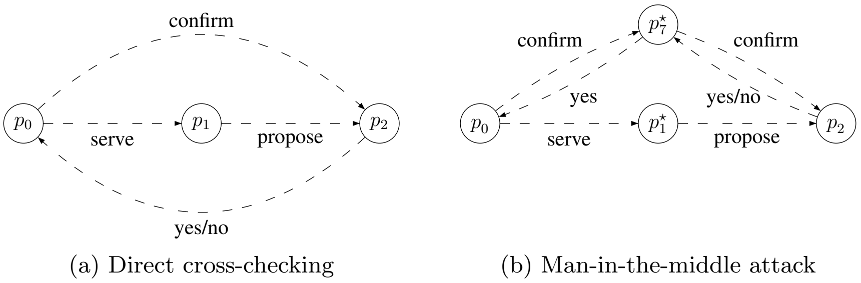
(iii) *Partial serve (direct check):* If p_f serves only $(1-\delta) \cdot |\mathcal{R}|$ chunks, it is blamed by $f/\,|\mathcal{R}|$ for each of the $\delta \cdot |\mathcal{R}|$ missing chunks by each of the f

requesting nodes. This again results in a blame value of $f \cdot (f / |\mathcal{R}|) \cdot \delta \cdot |\mathcal{R}| = \delta \cdot f^2$.

The blame values emitted by the different direct verifications are therefore homogeneous and comparable on average since all misbehaviors lead to the same amount of blame for a given degree of freeriding δ. Thus, they result in a consistent and meaningful score when summed up.

7.2.3 Fooling the Direct Cross-check ($\star$)

Considering a set of colluding nodes, nodes may lie to verifications to cover each other up. Consider the situation depicted in Figure 7.8a, where p_1 is a freerider. If p_0 colludes with p_1, then it will not blame p_1, regardless of p_2's answer. Similarly, if p_2 colludes with p_1, then it will answer to p_0 that p_1 sent a valid proposal, regardless of what p_1 sent. Even when neither p_0 nor p_2 collude with p_1, p_1 can still fool the direct cross-checking thanks to a colluding third party by implementing a *man-in-the-middle attack* as depicted in Figure 7.8b. Indeed, if a node p_7 colludes with p_1, then p_1 can tell p_0 it sent a proposal to p_7 and tell p_2 that the chunk originated from p_7. Doing this, both p_0 and p_2 will not detect that p_1 sent an invalid proposal. The *a posteriori verifications* presented in the next section address this issue.



(a) Direct cross-checking (b) Man-in-the-middle attack

Figure 7.8: Direct cross-checking and attack. Colluding nodes are denoted with a '$\star$'.

7.2.4 A Posteriori Verifications

As stated in the analysis of the gossip protocol, the random choices made in the partners selection must be verified. In addition, the example described in the previous section, where freeriders collude to circumvent direct cross-checking, highlights the need for statistical verification of a node's past communication partners.

The history of a node that biased its partner selection contains a relatively large proportion of colluding nodes. If only a small fraction of colluding nodes is present in the system, they will appear more frequently than honest nodes in each other's histories and can therefore be detected. Based on this remark, we propose an *entropic check* to detect the bias induced by freeriders on the history of nodes, illustrated in Figure 7.9.

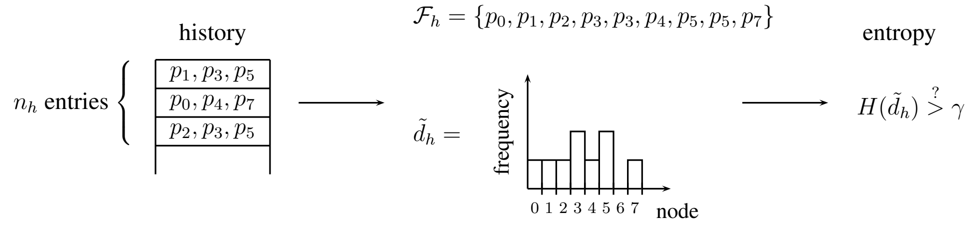


Figure 7.9: Entropic check on proposals ($f = 3$).

Every h seconds, each node picks a random node and verifies its local history. When inspecting the history of p, the verifier computes the number of occurrences of each node in the set of proposals sent by p during the last h seconds. Defining $\mathcal{F}_h$ as the multiset of nodes to whom p_1 sent a proposal during this period (a node may indeed appear more than once in $\mathcal{F}_h$), the distribution $\tilde{d}_h$ of nodes in $\mathcal{F}_h$ characterizes the randomness of the partners selection. We denote by $\tilde{d}_{h,i}$ the number of occurrences of node i ($i \in \{1, \ldots, n\}$) in $\mathcal{F}_h$ normalized by the size of $\mathcal{F}_h$. Assessing the uniformity of the distribution $\tilde{d}$ of p_1's history is achieved by comparing its Shannon entropy to a threshold γ ($0 \leq \gamma \leq \log_2(n_h f)$).

$$H(\tilde{d}_h) = -\sum_i \tilde{d}_{h,i} \log_2(\tilde{d}_{h,i}) \tag{7.1}$$

The entropy is maximum when every node of the system appears at most once in $\mathcal{F}_h$ (assuming $n > |\mathcal{F}_h| = n_h f$). In that case, it is equal to $\log_2(n_h f)$. Since the peer selection service may not be perfect, the threshold γ must be tolerant to small deviation with respect to the uniform distribution to avoid *false positives* (i.e., honest nodes being blamed). Details on how to dimension γ are given in Section 7.3.2.

An entropic check must be coupled with an *a posteriori* cross-checking verification procedure to guarantee the validity of the inspected node's history. Cross-checking is achieved by polling all or a subset of the nodes mentioned in the history for an acknowledgment. The inspected node is blamed by 1 for each proposal in its history that is not acknowledged by the alleged receiver. Therefore, an inspected freerider replacing colluding nodes by honest nodes in its history in order to pass the entropic check will not be covered by the honest nodes and will thus be blamed accordingly.

Because of the man-in-the middle attack presented in Section 7.2.2, a complementary entropic check is performed on the multi-set of nodes $\mathcal{F}'_h$ that asked the nodes in $\mathcal{F}_h$ for a confirmation, i.e., direct cross-checking. On the one hand, for an honest node p_0, $\mathcal{F}'_h$ is composed of the nodes that sent chunks to p_0 – namely its *fanin*. On the other hand, for a freerider $p_0^\star$ that implemented the man-in-the-middle attack, the set $\mathcal{F}'_h$ of $p_0^\star$ contains a large proportion of colluding nodes – the nodes that covered it up for the direct cross-checking – and thus fail the entropic check. If the history of the inspected node does not pass the entropic checks (i.e, fanin and fanout), the node is expelled from the system.

Local history auditing verifications are sporadically performed by the nodes using TCP connections. The reasons for using TCP are that *(i)* the overhead of establishing a connection is amortized since local history auditing happens sporadically and carries out a large amount of data, i.e., proportional to h, and *(ii)* local auditing is very sensitive to message losses as it can lead to expulsion from the system.

7.2.5 Communication Costs

In this section, we evaluate the overhead incurred by LiFT. To this end, we compute the maximum number of verification and blame messages sent by a node during one gossip period. The overheads of the verifications are summarized in Table 7.3. Note that we do not consider statistical verifications in this section as it does not imply a regular overhead but only sporadic message exchanges, the *a posteriori* verifications are evaluated in practice in Section 7.4.

Direct Check Verifying that what was requested is actually served does not require any exchange of verification messages as direct check consists only in comparing the number of chunks requested by the verifier to the number of chunks it really received. However, direct check may lead to the emission of f blames, i.e., a blame for each sender (to M managers). The communication overhead caused by direct checking is therefore $\mathcal{O}(M \cdot f)$ messages.

Cross-checking In order to check that the chunks it sent during the previous gossip period are proposed further on, the verifier polls the f partners of its f partners with probability p_{cc}, i.e., sending confirm messages. Similarly, a node is polled by $p_{\mathrm{cc}} \cdot f^2$ nodes per gossip period on average and therefore sends $p_{\mathrm{cc}} \cdot f^2$ answers to confirm messages. Finally, a node sends the list of its current partners (i.e., ack messages) to the f nodes (on average) that served chunks to it in the last gossip period, i.e., sending of the ack messages. In addition, since a node inspects its f partners, direct cross-checking may lead to the emission of a maximum of f blames (to M managers). The communication overhead caused by direct cross-checking is therefore $\mathcal{O}(p_{\mathrm{cc}} \cdot f^2 + p_{\mathrm{cc}} \cdot M \cdot f + f)$ messages. The number of messages sent by LiFT is $\mathcal{O}(M \cdot f + f^2)$. This has to be compared to the number of messages sent by the three-phase gossip protocol itself, namely $f(2 + |\mathcal{S}|)$, where $\mathcal{S}$ is the set of served chunks, the two additional messages being the proposal and the request. The overhead of LiFT is negligible when taking into account the size of the chunks sent by a node, which is several orders of magnitude larger than the verification and blame messages. Note that M is a system parameter defining the number of managers of a node and does not depend on the size of the system. Finally, since $f \sim \ln(n)$, both the three-phase protocol and LiFT scale with the number of nodes.

direct verifications (messages)	0
direct verifications (blames)	$\mathcal{O}(M \cdot f)$ for the verifier
direct cross-check (messages)	$\mathcal{O}(p_{\mathrm{cc}} f^2)$ for the verifier (confirm messages) $\mathcal{O}(p_{\mathrm{cc}} f)$ for the inspected node (ack messages) $\mathcal{O}(p_{\mathrm{cc}} f^2)$ for each witness (answer messages)
direct cross-check (blames)	$\mathcal{O}(p_{\mathrm{cc}} \cdot M \cdot f)$ for the verifier

Table 7.3: Overhead of verifications.

7.3 Parameterizing LiFT

This section provides a methodology to set LiFT's parameters. With this aim, the performance of LiFT with respect to detection is analyzed theoretically. Closed form expressions of the detection and false positive probabilities function of the system parameters are given. Theoretical results allow the system designer to set the system parameters, e.g., detection thresholds.

This section is split in two. First, the design of the score-based detection mechanism is presented and analyzed taking into account message losses. Second, the entropy-based detection mechanism is analyzed taking into account the underlying peer-sampling service. Both depend on the degree of freeriding and on the favoring factor, i.e., how freeriders favor colluding partners.

The principal notations used in this section are summarized in Table 7.4 (page 113).

7.3.1 Score-based Detection

Due to message losses, a node may be wrongfully blamed, i.e., blamed even though it follows the protocol. Freeriders are additionally blamed for their misbehaviors. Therefore, the score distribution among the nodes is expected to be a mixture of two components corresponding respectively to those of honest nodes and freeriders. In this setting, likelihood maximization algorithms are traditionally used to decide whether a node is a freerider or not. Such algorithms are based on the relative score of the nodes and are thus not sensitive to wrongful blames. Effectively, wrongful blames have the same impact on honest nodes and freeriders.

However, in the presence of freeriders, two problems arise when using relative score-based detection: *(i)* freeriders are able to decrease the probability of being detected by wrongfully blaming honest nodes, and *(ii)* the score of a node joining the system is not comparable to those of the nodes already in the system. For these reasons, in LiFT, the impact of wrongful blames, due to message losses, is automatically compensated and detection thus consists in comparing the nodes' compensated scores to a fixed threshold η. In short, when the compensated score of a node drops below η, the managers of that node broadcast a revocation message expelling the node from the system using gossip.

Considering message losses independently drawn from a Bernoulli distribution of parameter p_l (we denote by $p_r = 1-p_l$ the probability of reception), we derive a closed-form expression for the expected value of the blames applied to honest nodes by direct verifications during a given timespan. Periodically increasing all scores accordingly leads to an average score of 0 for honest nodes. This way, the fixed threshold η can be used to distinguish between honest nodes and freeriders. To this end, we analyze, for each verification, the situations where message losses can cause wrongful blames and evaluate their average impact. For the sake of the analysis, we assume that *(i)* a node receives at least one chunk during every gossip period (and therefore it will send proposals during the next gossip period), and *(ii)* each node requests a constant number $|\mathcal{R}|$ of chunks for each proposal it receives. We consider the case where cross-checking is always performed, i.e., $p_{\text{cc}} = 1$.

Direct Check (dc) For each requested chunk that has not been served, the node is blamed by $f/|\mathcal{R}|$. If the proposal is received but the request is lost (i.e., $p_r(1-p_r)$), the node is blamed by f ((a) in Equation 7.2). Otherwise, when both the proposal and the request message are received (i.e., p_r^2), the node is blamed by $f/|R|$ for each of the chunks lost (i.e., $(1-p_r)|R|$) ((b) in Equation 7.2). The expected blame applied to an honest node (by its f partners), during one gossip period, due to message losses is therefore:

$$\tilde{b}_{\text{dc}} = f \cdot \left[\overbrace{p_r(1-p_r) \cdot f}^{\text{(a)}} + \overbrace{p_r^2 \cdot (1-p_r)\,|\mathcal{R}| \cdot \frac{f}{|\mathcal{R}|}}^{\text{(b)}} \right] = p_r(1-p_r^2) \cdot f^2 \qquad (7.2)$$

Cross-checking (cc) On average, a node receives f proposals during each gossip period. Therefore a node is subject to f direct cross-checking verifications and each verifier asks for a confirmation to the f partners of the inspected node. Let p_1 be the inspected node and p_0 a verifier. First, note that p_0 verifies p_1

only if it served chunks to p_1, which requires that its proposal and the associated request have been received (i.e., p_r^2). If at least one chunk served by p_0 or the ack has been lost (i.e., $1 - p_r^{|\mathcal{R}|+1}$), p_0 will blame p_1 by f regardless of what happens next, since all of the f proposals sent by p_1 are invalid from the standpoint of p_0 ((a) in Equation 7.3). Otherwise, that is, if all the chunks served and the ack have been received (i.e., $p_r^{|\mathcal{R}|+1}$), p_0 blames p_1 by 1 for each negative or missing answer from the f partners of p_1. This situation occurs when the proposal sent by p_1 to a partner, the confirm message or the answer is lost (i.e., $1 - p_r^3$) ((b) in Equation 7.3).

$$\tilde{b}_{\text{cc}} = f \cdot p_r^2 \left[\overbrace{(1 - p_r^{|R|+1}) \cdot f}^{\text{(a)}} + \overbrace{f \cdot p_r^{|\mathcal{R}|+1}(1 - p_r^3)}^{\text{(b)}} \right] = p_r^2(1 - p_r^{|\mathcal{R}|+4}) \cdot f^2 \tag{7.3}$$

A Posteriori Cross-checking (apcc) This procedure asks the nodes that appear in the inspected node's history for confirmation which can cause wrongful blames. Effectively, if a proposal sent by the inspected node has not been received by the destination node, due to message losses, the latter will not acknowledge reception when asked. This leads again to wrongful blames. However, since the nodes are polled using TCP, the polling message and the answer are not subject to message losses. On average, only $p_r \cdot n_h f$ proposals in the inspected node history are confirmed by the destination leading to an average blame of:

$$\tilde{b}_{\text{apcc}} = (1 - p_r) \cdot n_h f \tag{7.4}$$

Similarly to direct verifications, the wrongful blames applied by the local auditing must be compensated. However, this should be done only sporadically, i.e., only when a node is effectively audited, since these verifications are not triggered at each gossip period.

From the previous analysis, we obtain a closed-form expression for the expected value of the blame b applied to an honest node by direct verifications due to message losses:

$$\tilde{b} = \tilde{b}_{\text{dc}} + \tilde{b}_{\text{cc}} = p_r(1 + p_r - p_r^2 - p_r^{|\mathcal{R}|+5}) \cdot f^2 \ . \tag{7.5}$$

Following the same line of reasoning, a closed-form expression for the standard deviation $\sigma(b)$ of b can be derived.

Figure 7.10 depicts the distribution of scores after one gossip period in a simulated network of 10,000 honest nodes in steady state (where both direct verifications and direct cross-checking are performed with $p_{\text{cc}} = 1$). The message loss rate p_l has been set to 7%, the fanout f to 12 and $|\mathcal{R}| = 4$. The scores of the nodes have been increased by $-\tilde{b} = 72.95$, according to Formula (7.5). We observe that, as expected, the average score (dotted line) is close to zero (< 0.01)

which means that the wrongful blames have been successfully compensated. The experimental standard deviation is $\sigma(b) = 25.6$.

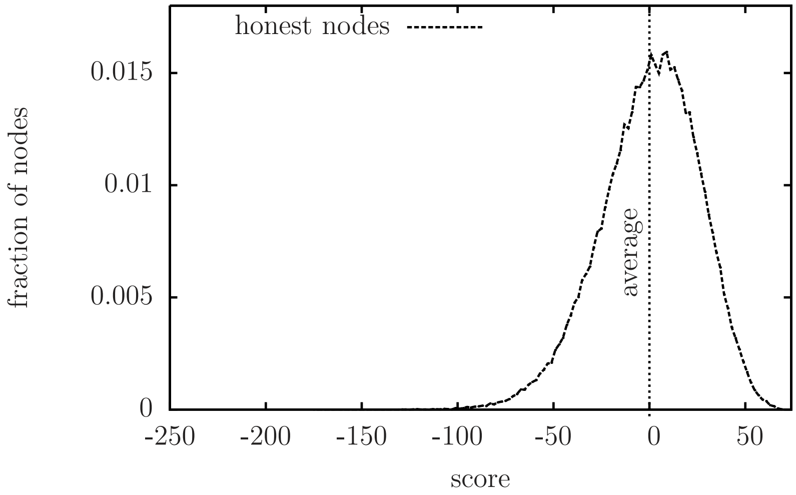


Figure 7.10: Impact of message losses.

A node can be expelled from the system either when its score drops beyond a fixed threshold (η) or upon a local auditing procedure. We now evaluate the ability of LiFT to detect freeriders (probability of detection α) and the proportion of honest nodes wrongfully expelled from the system (probability of false positives β) in both situations.

As mentioned above, the score-based detection mechanism uses a fixed threshold η to which the scores of the nodes are compared. With this aim, the score of each node is *adjusted* (to compensate wrongful blames) and *normalized* by the number of gossip periods r the node spent in the system. At the t-th gossip period, the normalized score of a node writes:

$$s = -\frac{1}{r}\sum_{i=0}^{r}\left(b_{t-i} - \tilde{b}\right) \ , \tag{7.6}$$

where b_i is the value of the blames applied to the node during the i-th gossip period. From the previous analysis, we get the expectation and the standard deviation of the blames applied to honest nodes at each round due to message losses, therefore, assuming that the b_i are i.i.d. (independent and identically distributed) we get $\mathbb{E}[s] = 0$ and $\sigma(s) = \sigma(b)/\sqrt{n_T}$. Using Bienaymé-Chebyshev's inequality, we derive an upper bound for the probability of false positive:

$$\beta = P\left(s < \eta\right) \leq P\left(|s| > -\eta\right) \leq \frac{\sigma(b)^2}{r \cdot \eta^2} \ .$$

The probability α to catch a freerider depends on its *degree of freeriding* that characterizes its deviation to the protocol. Formally, we define the degree of freeriding as a 3-uple $\Delta = (\delta_1, \delta_2, \delta_3)$, $0 \leq \delta_1, \delta_2, \delta_3 \leq 1$, so that a freerider contacts only $(1 - \delta_1) \cdot f$ nodes per gossip period, proposes the chunks received from a proportion $(1 - \delta_2)$ of the nodes that served it in the previous gossip period, and serves $(1 - \delta_3) \cdot |\mathcal{R}|$ chunks to each requesting node. The gain in terms of the upload bandwidth saved is therefore $1 - (1 - \delta_1)(1 - \delta_2)(1 - \delta_3)$.

Following the same line of reasoning as in the previous section, we can derive a closed-form expression for the expected blame applied to a freerider as a function of Δ:

$$\begin{aligned}\tilde{b}'(\Delta) = {} & (1 - \delta_1) \cdot p_r \left(1 - p_r^2(1 - \delta_3)\right) \cdot f^2 + \delta_2 \cdot f^2 \\ & + (1 - \delta_2) \cdot p_r^2 \cdot \left[p_r^{|\mathcal{R}|+1}(1 - p_r^3(1 - \delta_1)) + (1 - p_r^{|\mathcal{R}|+1})\right] \cdot f^2 \enspace .\end{aligned}$$

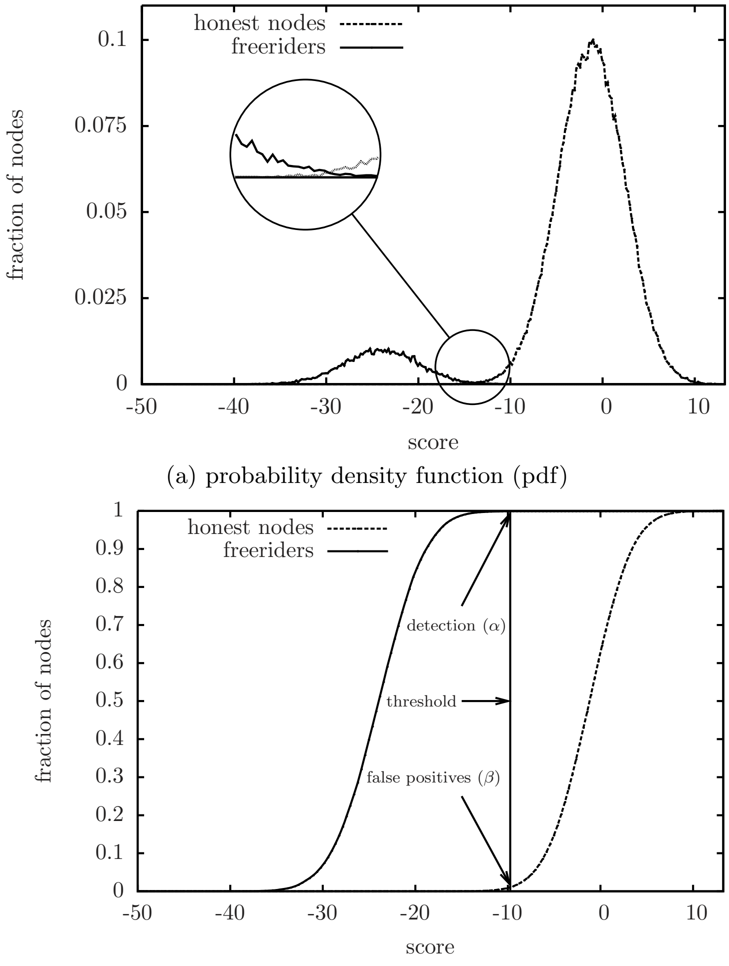
Similarly to the expression $\sigma(b)$, a closed-form expression for the standard deviation $\sigma(b'(\Delta))$ of $b'(\Delta)$ can be obtained. The probability of detection α, similarly to the probability of false positives β, can be lower bounded:

$$\alpha \geq 1 - \frac{\sigma(b'(\Delta))^2}{r \cdot (\tilde{b}'(\Delta) - \eta)^2} \enspace .$$

Note that under the assumption that losses are independently drawn from a Bernoulli distribution the performance of LiFT increases over time. Effectively, as the detection threshold is fixed and the standard deviations of the score distributions tend to zero when the time spent in the system increases, the probability of detection α increases to one and the probability of false positive β decreases to zero.

Figure 7.11 depicts the distribution of normalized scores in the presence of $1,000$ freeriders of degree $\Delta = (0.1, 0.1, 0.1)$ in a $10,000$-node system after $r = 50$ gossip periods. We plot separately the distribution of scores among honest nodes and freeriders. As expected, the probability density function (Figure 7.11a) is split into two disjoint modes separated by a gap: the lowest (i.e., left most) mode corresponds to freeriders and the highest one to honest nodes. Figure 7.11b depicts the cumulative distribution function of scores and illustrates the notion of detection and false positives for a given value of the detection threshold (i.e., $\eta = -9.75$).

We set the detection threshold η to -9.75 so that the probability of false positive is lower than 1%, we assume that freeriders perform all possible attacks with the same probability ($\delta_1 = \delta_2 = \delta_3 = \delta$) and we observe the proportion of freeriders detected by LiFT for several values of δ. Figure 7.12 plots α and β as functions of δ. For instance, we observe that for a node freeriding by 5%, the probability of being detected by LiFT is 65%. Beyond 10% of freeriding, a node



(a) probability density function (pdf)

(b) cumulative distribution function (cdf)

Figure 7.11: Distribution of normalized scores in the presence of freeriders ($\Delta = (0.1, 0.1, 0.1)$).

is detected over 99% of the time. It is commonly assumed that users are willing to use a modified version of the client application only if it increases significantly their benefit (resp. decreases their contribution). In FlightPath [LCM$^+$08], this threshold is assumed to be around 10%. With LiFT, a freerider achieves a gain of 10% for $\delta = 0.035$ which corresponds to a probability of being detected of 50% (Figure 7.12).

7.3.2 Entropy-based Detection

For the sake of fairness and in order to prevent colluding nodes from covering each other up, LiFT includes an entropic check assessing the statistical validity

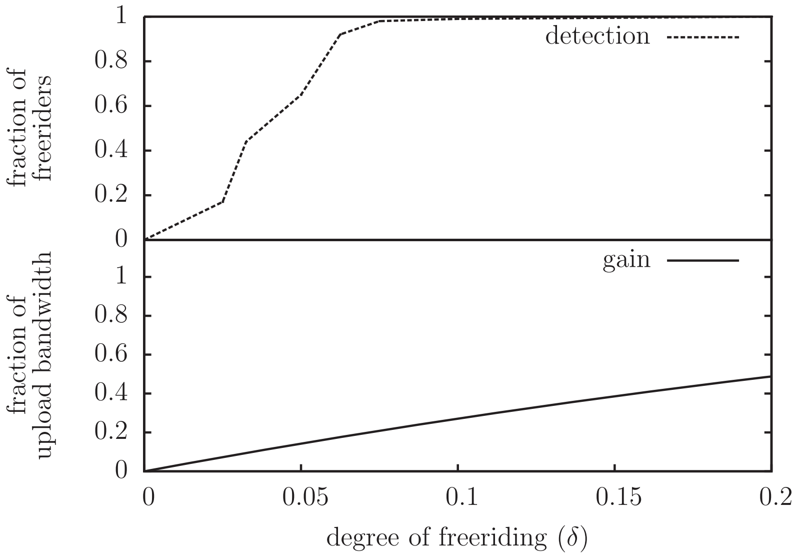
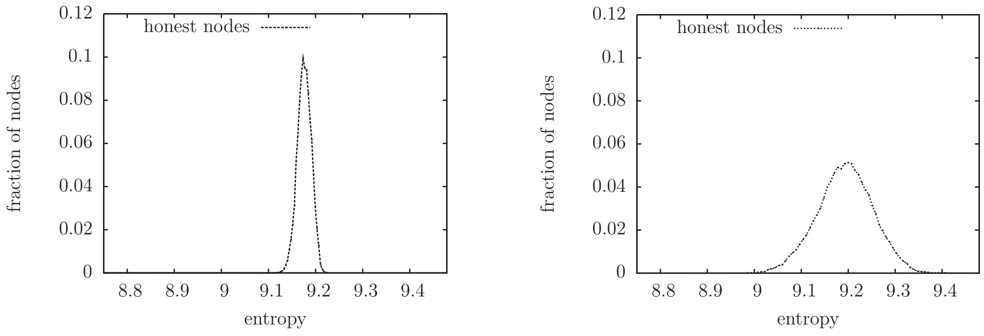


Figure 7.12: Proportion of freeriders detected by LiFT.

of the partner selection. To this end, the entropy H of the distribution of the inspected node's former partners is compared to a threshold γ, which is a parameter of the system. The distribution of the entropy of honest nodes' histories depends on the peer sampling algorithm used and can be estimated by simulations. Figure 7.13a depicts the distribution of entropy for a history of $n_h f = 600$ partners ($n_h = 50$ and $f = 12$) of a 10,000-node system using a full membership-based partner selection. The observed entropy ranges from 9.11 to 9.21 for a maximum reachable value of $\log_2(n_h f) = 9.23$. Similarly, the entropy of the fanin multi-set $\mathcal{F}'_h$, e.g., nodes that selected the inspected node as partner, is depicted in Figure 7.13b. The observed entropy ranges from 8.98 to 9.34. Note that the size of $\mathcal{F}'_h$ can be greater than $n_h f$ (but is $n_h f$ on average) and therefore the bound $\log_2(n_h f)$ does not apply to the entropy of fanin.

The presented results show that the probability of wrongfully expelling an inspected node during local auditing is negligible when the threshold γ is set to 8.95. This threshold is used for both fanout and fanin entropic check.

We now analytically determine to what extent a freerider can bias its partner selection without being detected by local auditing, given a threshold γ and a number of colluding nodes m'. A first requirement to be able to detect colluding nodes is that the number of proposals in a node's history must be greater than the number of colluding freeriders. Otherwise, by proposing chunks only to other freeriders in a round-robin manner, a node may still be able to achieve a maximized entropy. We therefore set h so that $n_h f \gg m'$. Consider a freerider that biases partner selection in order to favor colluding freeriders by picking a freerider as partner with probability p_m and an honest node with probability



(a) entropy of fanout

(b) entropy of fanin

Figure 7.13: Distribution of the entropy H of the nodes' histories using a full membership-based partner selection.

$1 - p_m$. We seek the maximum value $p_m^\star$ for p_m, function of γ and m'. Defining the probability law of the partner selection among colluding nodes (resp. honest nodes) by X (resp. by Y), the entropy of its fanout writes:

$$
\begin{aligned}
H(\mathcal{F}_h) &= -\sum p(n)\log_2(p(n)) \\
&= -\sum_{n\in X} p(n)\log_2(p(n)) - \sum_{n\in Y} p(n)\log_2(p(n)) \\
&= -\sum_{n\in X} p_m p_X(n)\log_2(p_m p_X(n)) \\
&\quad -\sum_{n\in Y}(1-p_m)p_Y(n)\log_2((1-p_m)p_Y(n)) \\
&= -p_m \sum_{n\in X}\left(p_X(n)\log_2(p_X(n)) + p_X(n)\log_2(p_m)\right) \\
&\quad -(1-p_m)\sum_{n\in Y}\left(p_Y(n)\log_2(p_Y(n)) + p_Y(n)\log_2(1-p_m)\right) \\
&= -p_m \left(\log_2(p_m)\underbrace{\sum_{n\in X} p_X(n)}_{=1} + \underbrace{\sum_{n\in X}\left(\log_2(p_X(n))\cdot p_X(n)\right)}_{=-H(X)} \right) \\
&\quad -(1-p_m)\left(\log_2(1-p_m)\underbrace{\sum_{n\in Y} p_y(n)}_{=1} + \underbrace{\sum_{n\in Y}\left(\log_2(p_Y(n))\cdot p_Y(n)\right)}_{-H(Y)} \right)
\end{aligned}
$$

$$= -p_m \log_2 p_m - (1-p_m)\log_2(1-p_m) + p_m H(X) + (1-p_m)H(Y) \ .$$

Since X and Y are independent, this quantity is maximized when X and Y are the uniform distribution. Therefore, to maximize the entropy of its history, a freerider must choose uniformly at random its partners in the chosen class, i.e., honest or colluding. Therefore, given a threshold γ and a maximum number of colluding nodes m', we calculate $H(X)$ knowing the number of occurrences of each m' colluding freeriders in the freerider fraction $(p_m^\star \cdot n_h \cdot f)$ of history is $\frac{p_m \cdot n_h \cdot f}{m'}$, the probability to have a freerider is $\frac{p_m^\star \cdot n_h \cdot f}{m'}/(p_m^\star \cdot n_h \cdot f) = 1/m'$. We have $H(X)$ and $H(Y)$ defined as:

$$\begin{aligned}
H(X) &= -\sum_{m'} \frac{1}{m'} \log_2\left(\frac{1}{m'}\right) \\
H(Y) &= -\sum_{n_h \cdot f(1-p_m^\star)} \frac{1}{n_h \cdot f(1-p_m^\star)} \log_2\left(\frac{1}{n_h \cdot f(1-p_m^\star)}\right) \\
&= -\log_2\left(\frac{1}{n_h \cdot f(1-p_m^\star)}\right) \ .
\end{aligned}$$

We can therefore calculate γ as:

$$\begin{aligned}
\gamma &= -p_m^\star \log_2 p_m^\star - (1-p_m^\star)\log_2(1-p_m^\star) - p_m^\star \log_2\left(\frac{1}{m'}\right) \\
&\quad + (1-p_m^\star)(-\log_2\left(\frac{1}{n_h \cdot f(1-p_m^\star)}\right) \\
&= -p_m^\star \log_2\left(\frac{p_m^\star}{m'}\right) - (1-p_m^\star)\log_2\left(\frac{1}{n_h \cdot f}\right) \ , \qquad (7.7)
\end{aligned}$$

where $p_m^\star$ is the maximum value for p_m that a freerider can use without being detected. Inverting numerically Formula (7.7), we deduce that for $\gamma = 8.95$ a freerider colluding with 25 other nodes can serve its colluding partners 15% of the time, without being detected. In this setting, a freerider can therefore decrease its contribution by 15%.

7.4 Evaluation

We now evaluate LiFT on top of the gossip-based streaming protocol described in Algorithm 3.3, over the PlanetLab testbed.

Notations	**Descriptions**
n, m	number of nodes / freeriders
$\lvert\mathcal{R}\rvert$, $\lvert\mathcal{S}\rvert$	number of chunks requested, resp. served
f	fanout
n_h	size of history
$\mathcal{F}_h, \mathcal{F}'_h$	multi-set of fanout / fanin in history
p_{cc}	probability to trigger cross-checking
p_l	probability of message loss ($p_r = 1 - p_l$)
$\tilde{b}$	average value of wrongful blames
$\sigma(b)$	standard deviation of wrongful blames
r	number of gossip periods spent in the system
s	normalized score
$\Delta = (\delta_1, \delta_2, \delta_3)$	degree of freeriding (3-uple)
$\delta = \delta_1 = \delta_2 = \delta_3$	degree of freeriding
$\tilde{b}(\Delta)$	average value of blames (freeriders)
$\sigma(b'(\delta))$	standard deviation of blames (freeriders)
η	detection threshold (blame-based detection)
α	probability of detection (blame-based detection)
β	probability of false positive (blame-based detection)
γ	detection threshold (entropy-based detection)

Table 7.4: Summary of principal notations.

7.4.1 Experimental Setup

We have deployed and executed LiFT on a 300 PlanetLab node testbed, broadcasting a stream of 674 kbps in the presence of 10% of freeriders. The freeriders *(i)* contact only $\hat{f} = 6$ random partners ($\delta_1 = 1/7$), *(ii)* propose only 90% of what they receive ($\delta_2 = 0.1$) and finally *(iii)* serve only 90% of what they are requested ($\delta_3 = 0.1$). The fanout of all nodes is set to 7 and the gossip period is set to 500 ms. The blaming architecture uses $M = 25$ managers for each node.

7.4.2 Practical Cost

We report on the overhead measurements of direct and *a posteriori* verifications (including blame messages sent to the managers) for different stream rates.

Direct Verifications Table 7.5 gives the bandwidth overhead of the direct verifications of LiFT for three values of p_{cc}. Note that the overhead is not null when $p_{\text{cc}} = 0$ since ack messages are always sent. Yet, we observe that the overhead is negligible when $p_{\text{cc}} = 0$ (i.e., when the system is healthy) and remains reasonable when $p_{\text{cc}} = 1$ (i.e., when the system needs to be purged from freeriders).

	direct verifications			***a posteriori* verif.**
	$p_{cc} = 0$	$p_{cc} = 0.5$	$p_{cc} = 1$	
674 kbps stream	1.07%	4.53%	8.01%	3.60%
1082 kbps stream	0.69%	3.51%	5.04%	2.89%
2036 kbps stream	0.38%	2.80%	2.76%	1.74%

Table 7.5: Practical overhead

A Posteriori Verifications A history message contains n_h entries. Each entry consists of f nodes identifiers and the chunk ids that were proposed. Both the fanout and fanin histories are sent upon a posteriori verification.

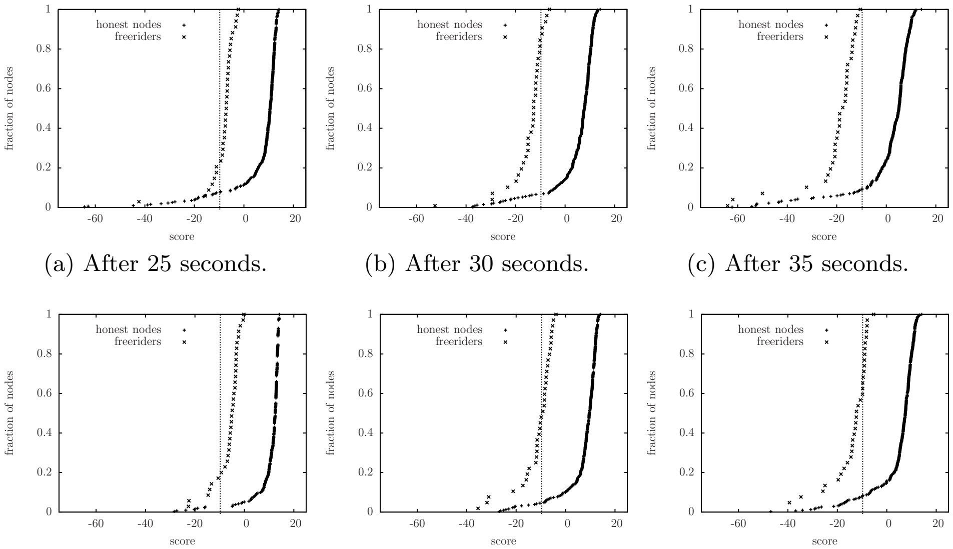
Besides the entropic checks, *a posteriori* cross-checking is performed on a subset of the fanout or fanin entries. We measured the maximum overhead, that is when the whole fanout and fanin histories are cross-checked. The overhead incurred by *a posteriori* verifications in our experimental setup (i.e., a history size $n_h = 50$, a gossip period of 500 milliseconds, a fanout of $f = 7$ and *a posteriori* verification period of $h = 25$ seconds) is given in Table 7.5.

7.4.3 Experimental Results

We have executed LiFT with $p_{\text{cc}} = 1$ and $p_{\text{cc}} = 0.5$. Figure 7.14 depicts the scores obtained after 25, 30 and 35 seconds when running direct verifications and cross-checking. The scores have been compensated as explained in Section 7.3.1, assuming a loss rate of 4% (average value observed on PlanetLab).

The two cumulative distribution functions for honest nodes and freeriders are clearly separated. The threshold for expelling freeriders is set to -9.75 (as specified in the analysis). In Figure 7.14b ($p_{\text{cc}} = 1$, after 30 s) the detection mechanism expels 86% of the freeriders and 12% of the honest nodes. In other words, after 30 seconds, 14% of freeriders are not yet detected and 12% represent false positives, mainly corresponding to honest nodes that suffer from very poor connection (e.g., limited connectivity, message losses and bandwidth limitation). These nodes do not deliberately freeride, but their connection does not allow them to contribute their fair share. This is acceptable as such nodes should not have been allowed to join the system in the first place. As expected, with p_{cc} set to 0.5 the detection is slower but not twice as slow. Effectively, with nodes freeriding with $\delta_3 > 0$ (i.e., partial serves) the direct checking blames freeriders without the need for any cross-check. This explains why the detection after only 35 seconds with $p_{\text{cc}} = 0.5$ (Figure 7.14f) is comparable with the detection after 30 seconds with $p_{\text{cc}} = 1$ (Figure 7.14b).

As stated in the analysis, we observe that the gap between the two cumulative distribution functions widens over time. However, the variance of the score does not decrease over time (for both honest nodes and freeriders). This is due to



(a) After 25 seconds. (b) After 30 seconds. (c) After 35 seconds.

(d) After 25 seconds. (e) After 30 seconds. (f) After 35 seconds.

Figure 7.14: Cumulative distribution functions of scores with $p_{\text{cc}} = 1$ (above) and $p_{\text{cc}} = 0.5$ (below).

the fact that we considered in the analysis that the blames applied to a given node during distinct gossip periods were independent and identically distributed (i.i.d.). In practice however, successive gossip periods are correlated. Effectively, a node with a poor connection is usually blamed more than nodes with high capabilities, and this remains true over the whole experiment.

7.5 Related Approaches

Since its successful application in BitTorrent the Tit-for-Tat (TfT) incentives have become a *de facto* standard for dealing with freeriders in large scale content distribution systems. BAR Gossip – and by extension FlightPath – is the most successful and complete application including TfT to ensure fair collaboration of participants in the context of gossip protocols. However, it suffers from several issues that drastically limit both its efficiency and its practicality in a large-scale content dissemination system.

First, BAR Gossip makes intensive use of both symmetric and asymmetric cryptography. Beyond the fact that this adds a non negligible overhead to the protocol, both in terms of message exchange and computation, it requires a trusted third party to issue identification certificates, namely a public key infrastructure. In addition to requiring prior registration, asymmetric cryptography techniques generate a high load, proportional to the size of the network, on the centralized server. Further, it is shown in [HJVR08] that BAR Gossip collapses when the system grows beyond a given size.

Second, similarly to TfT-based protocols, BAR Gossip relies on altruistic nodes and opportunistic pushes where nodes upload pieces of data without receiving anything in exchange. This component is essential for TfT to work in a real life setting mainly to ensure that nodes joining the system can gain bargaining power (i.e., pieces to exchange) to initiate symmetric exchanges but also to ensure that nodes with low upload capacities are unchoked (which is unlikely in bidirectional exchanges, as nodes with high upload bandwidth are preferred). This component is used without any protection against freeriders despite the fact that most attacks to TfT actually exploit it [LMSW06, SPCY07]. This shows that protocols using TfT are still sensitive to freeriders and that the problem of designing incentives for the wide class of epidemic dissemination protocols (to which the opportunistic pushes belong) is of the utmost importance in fighting against freeriders. In BAR Gossip this issue is addressed by forcing the nodes that are opportunistically unchoked to send the exact same amount of data they received, should they send junk data if they have nothing of interest. This results in a waste of bandwidth and seems inappropriate in a context where the upload bandwidth of nodes is precisely the resource that needs to be used optimally.

Third, protocols such as BAR Gossip are extremely vulnerable to the presence of colluding nodes. The first reason for this is that the game theory at the core of BAR Gossip does not handle teams of players, but an even larger problem is that the opportunistic component is alone sufficient for achieving very good performance when exploited by a coalition. Assuming a set of colluding freeriders (potentially hosted on a high speed network or even on the same machine [Dou02]) that take advantage of the opportunistic component to obtain pieces for free (or against a small amount of junk data) and then share these pieces among the coalition, the stream can be downloaded with a small contribution in return. This situation where nodes collude is not captured by the BAR model which considers rational nodes to be nodes willing to contribute as long as this generates profit in return without attention to the nodes with which they collaborate.

Trying to solve these issues, the BitTorrent file sharing community has tried to promote *private trackers* that account for the ratio upload/download of registered users and force them to have a ratio above a certain limit (typically above 0.5). Doing so, the private trackers constrain users to *seed* files, i.e., to altruistically serve content to other registered users. This solution is in fact naive, since the reporting of upload/ratio to the trackers is not secured anyway and thus, a modified client can report arbitrarily large upload to its tracker, promoting its ratio and thus benefit from regular seeders. Interestingly, regular seeders blindly contribute in order to boost their ratio. In the end, freeriders benefit even more from regular seeders as regular seeders are forced to seed and even compete to contribute the most possible so that their ratio increases.

PeerReview [HKD07] deals with malicious nodes following an accountability approach. Nodes maintain signed logs of their actions that can be checked using a reference implementation running in addition to the application. When combined with CSAR [BDHU09], PeerReview can be applied to non-deterministic protocols. However, the intensive use of cryptography and the sizes of the logs maintained and exchanged drastically reduce the scalability of this solution. In addition, the validity of PeerReview relies on the fact that messages are always received which is not the case over the Internet.

The case of malicious participants was considered in the context of generic gossip protocols, i.e., consisting of state exchanges and updates [JMB04]. This system relies on cryptography for signing messages between nodes and does not consider malicious behaviors that stem from the partner selection, i.e., biasing the random choices. In addition, they do not tackle the problem of collusion.

The approaches that relate the most to LiFT are the distributed auditing protocol proposed in [HJVR08] and the passive monitoring protocol proposed in [KKU08]. In the first protocol, freeriders are detected by cross-checking their neighbors' reports. The latter focuses on gossip-based search in the Gnutella network. The nodes monitor the way their neighbors forward/answer queries in order to detect freeriders and query droppers. Yet, contrarily to LiFT – which is based on random peer selection – in both protocols the nodes's views are static, forming a fixed mesh overlay. In fact, the ideas to enforce fairness proposed in [PvS10] are indeed to promote long-term collaboration between nodes so that they can decide if they should give their fixed neighbors access to their resources or not based on past collaboration. These techniques thus cannot be applied to gossip protocols. In addition, the situation where colluding nodes cover up each other's misbehavior (not addressed in these work) makes such monitoring protocols vain.

7.6 Summary

We presented LiFT, a protocol for tracking freeriders in gossip-based asymmetric data dissemination systems. Beyond the fact that LiFT deals with the inherent randomness of the protocol, LiFT precisely relies on this randomness to make robust its verification mechanisms against colluding freeriders with a very low overhead. We provided a theoretical analysis of LiFT that allows system designers to set its parameters to their optimal values and characterizes its performance backed up by extensive simulations. We reported on our experimentations on PlanetLab which prove the practicability and efficiency of LiFT.

We can only see a short distance ahead, but we can see plenty there that needs to be done.

Alan Turing

8 Conclusion

In this thesis, we have investigated live streaming with gossip. In particular, we have investigated *if* and *how* a simple gossip-based algorithm can efficiently be used in scenarios *(i)* where participants can only contribute limited resources, *(ii)* when these limited resources are heterogeneously distributed among participants, and *(iii)* where an ideal state in the P2P environment is not reached, meaning that participants are not contributing their fair share of work.

The gossip paradigm provides many features required by high-bandwidth content dissemination such as fast propagation of rumors, probabilistic guarantee that each rumor reaches all participants, and high resilience to churn and high scalability. Its inherent redundancy, however, limits its usage, as it stands, to the dissemination of small updates.

The gossip-based protocol presented in Chapter 3 takes full potential of gossip's advantages by limiting its usage to the gossiping of content location. The content is subsequently pulled without duplicates, avoiding gossip's limitation in the context of high-bandwidth content dissemination. We therefore conclude that:

- Gossip can indeed be used for high-bandwidth content dissemination, and that it furthermore appears as an appealing approach for live streaming.

The remaining chapters of the body of work give specific answers to *how* gossip can be used for live streaming. Chapter 4 and Chapter 5 address scenario *(i)* by showing how such a gossip must be used in a scenario where nodes have limited resources. In Chapter 6, we present HEAP, a fanout adaptation mechanism to let gossip adapt to nodes' capabilities when they are heterogeneously distributed among nodes as answer to scenario *(ii)*. Finally, Chapter 7 proposes to complement gossip with LiFT for detecting and expelling freerider nodes that are not contributing their fair share of work, scenario *(iii)*.

Below, we first summarize our contributions in the thesis. Thereafter, we then outline a few open issues and directions for future investigations.

8.1 Summary of Results

The summary of the results of this thesis centers around three axes:

Live Streaming with Gossip in Constrained Environment In bandwidth-constrained environments, the upload bandwidth of nodes is considered the bottleneck for collaborative tasks. Its usage must thus be made with parsimony (Chapter 2). The gossip paradigm, presented in Chapter 3, is therefore used to disseminate the location of content which is then pulled by interested nodes, thus avoiding to waste bandwidth in receiving duplicates of payload.

We have shown in Chapter 4 that in such a constrained environment, increasing the fanout does not help nodes to receive a better quality stream and that the proactiveness of the protocol, namely how frequently a node picks different fanout partners to communicate with, should be the highest possible. In other words, a node should pick a different subset of fanout partners every gossip period.

- Increasing the fanout does not incur an intolerable load because of the duplicate propose messages, but it gives nodes the possibility to be requested by a larger number of nodes than usual. This means that nodes have possibly to serve a large number of nodes (possibly as large as fanout of them) at the same time and thus exceed their bandwidth with bursts.
- Regarding proactiveness, the reasons are different, but the problem remains the same in that nodes have a tendency to exceed their bandwidth. The less proactive the protocol is, the more a node close to the source remains longer close to the source and has content that is interesting for its chosen fanout partners. This node therefore needs to serve a relatively large number of them for a long period of time, exceeding its bandwidth constraint, whereas nodes far from the source keep under-utilizing their bandwidth.

In Chapter 5, we have proposed two mechanisms to complement the dissemination protocol, namely *Codec* and *Claim*, resulting in the gossip++ protocol. A node can receive many proposals for a given chunk but can also receive no proposal at all, be it because of message loss or simply because of the probabilistic nature of gossip. The fact that messages are lost, because of nodes exceeding their bandwidth and dropping them or because of lossy links, represents a problem that also, and principally, needs to be tackled in the second and third phases of the protocol, namely the request and the serve. *Codec* aims at being able to recover missing chunks because no proposal was received for them, while *Claim* leverages the possibly many duplicates of gossip in content location to re-request different nodes in case a request or a serve was dropped or lost.

Heterogeneous Bandwidth-constrained Environment Chapter 6 showed that the inherent load-balancing properties of gossip is not effective in the case where the upload capability distribution of nodes is heterogeneous. Considering that the average upload capability of the distribution is sufficient to sustain the stream rate, we proposed a novel way to account for heterogeneity in gossip protocols. With the computation of an average bandwidth capability, which aggregation is efficiently done with gossip, we proposed a way of locally adapting the fanout of nodes in such a way that their contribution is proportional to their upload capabilities and moreover, we ensured that the reliability of gossip in disseminating the content location was unaltered. Effectively, the contribution of a node highly depends on its fanout and not only should a node increase its fanout if it is considered rich and decrease it if it is considered poor but the amount of this local increase or decrease must result in a system that still guarantees efficiency, globally.

Presence of Freeriders In Chapter 7, we proposed LiFT, a protocol to secure high-bandwidth gossip-based dissemination protocols against freeriders. When sharing large content, participants can be tempted not to contribute their fair share of work if they have the feeling that their benefit is unaltered whatever their contribution. This behavior is common in asymmetric systems where the nodes' benefit is not directly correlated with their contribution. A node serving another for a given amount is not guaranteed that this very same node will eventually serve it back with the same amount. LiFT ensures that nodes closely follow the dissemination protocol and detects nodes that freeride by means of cross-checking previous interactions and statistical verifications. It does not use any cryptography and relies only on the fact that given its randomness, gossip will always reach a proportion of honest nodes willing to help detecting freeriders.

8.2 Future Work

The work in this thesis opens directions for both new research and enhancements to overcome limitations in gossip in general and in gossip++, HEAP or LiFT. We start by research in general in the field of gossip in Section 8.2.1, discuss networking problems in Section 8.2.2 and then discuss about performance in live streaming in Section 8.2.3. Section 8.2.4 discusses how adaptation to upload capabilities and knowledge about the distribution of capabilities could be used in order to further enhance the HEAP dissemination protocol. Finally, Section 8.2.5 reviews future work in the field of LiFT.

8.2.1 On Democratizing Gossip

Gossip has produced quite a number of very interesting scientific research papers but little is known about its usage in the industry. The only proof we have that gossip is used in commercial products is its presence as a membership service (i.e., a failure detector) in Amazon's Simple Storage Service (Amazon S3) [DHJ+07]. Following discussions with many researchers met during these past years, although it appears that there is a clear classification between structured and unstructured overlays, maybe ironically, there is no well-defined classification in the field of unstructured overlays. Gossip is sometimes viewed as just another mesh overlay, sometimes as flooding and sometimes as some kind of *random* process. These different perceptions, even though they are a bit reductive, not to say uncomplimentary, are correct in some sense:

- Gossip is indeed very close to mesh overlays since they are unstructured and random, but in contrary to mesh overlays, the overlay in gossip is highly dynamic.
- Gossip is a way of implementing the broadcast primitive with probabilistic guarantee, and the way it disseminates information in practice is close to flooding, except that the flood is indeed controlled.
- Gossip is inherently random, either in the underlying overlay construction or in the way it uses it to convey information.

The rare use of gossip in practice can be explained by two facts.

(i) Gossip has mainly been used for disseminating small updates and has only rarely been considered as an alternative for high-bandwidth content dissemination scenarios.

Because it creates many duplicates, gossip, in its simplest form, is not a good candidate for high-bandwidth content dissemination where the bandwidth usage of nodes is of utmost importance. Gossip is therefore restrained to propagating small updates, which it does very well, and therefore reduced to act as a side protocol, e.g., detecting failures, but not at the core of the collaborative task, except when the task itself deals with small updates, e.g., aggregation in sensor networks. A clear example of gossip acting as a side protocol in practice is its usage as a membership service in the Dynamo storage system of Amazon [DHJ+07].

(ii) Gossip represents a large signaling overhead as it creates many duplicates because of its random and proactive nature and this signaling overhead has not yet shown to provide added value to the protocol.

The trend moving from client-server to P2P architectures first moved to structured overlays such as trees. The main goal of trees is that once constructed and assuming there is no churn, the signaling overhead is reduced to a minimum as content can simply be pushed from the root to the leaves: the tree structure itself ensures that nodes will not receive duplicate information and that all nodes will eventually receive the data. Moving from trees to multiple-trees convinced most researchers that it was of the utmost importance to give a chance to all nodes to contribute to the system for improved overall performance. The next step, that is, moving from a structured overlay to an unstructured overlay, has already lost some researchers that wanted to stick to tree-like overlays for their reduced signaling overhead. Some research has indeed even turned back from mesh overlays to multiple-trees (on top of meshes), e.g, GridMedia [ZZSY07] and the last version of Coolstreaming [LXQ$^+$08]. Nonetheless and maybe also thanks to BitTorrent, mesh overlays are very appealing alternatives to trees, especially in case of catastrophic failures and churn.

What we consider the next step is indeed moving from mesh overlays to dynamic unstructured overlays, namely gossip. This next step has also lost some researchers on the way, e.g., the ones convinced by trees, the ones convinced by mesh overlays then coming back to multiple trees and the ones convinced by mesh overlays as such. To some extent, gossip might also have suffered from the rather bad publicity made about flooding, when it was used for propagating requests in the early Napster P2P system.

It is easy to show that the gossip approach is very simple, and by extension, much simpler than any other structured or unstructured approach. But if the practical issues of complicated approaches have already been solved, simplicity is not an argument of choice anymore. The signaling overhead from trees to meshes and from meshes to gossip has constantly increased and this could be seen as a waste of bandwidth exactly as memory is wasted by newer computer programs, namely software bloats.

Showing that the signaling overhead of gossip is not a waste but a feature is a research topic on its own and needs future attention. A clear example is the use of *Claim* showing that the possibility to request the same content from different nodes is a clear added value in the presence of freeriders or failures, known to happen in large-scale systems. There is thus much to bet that in an online Internet-based TV broadcasting system, not only will users freeride or fail, but they will also probably switch channels frequently leading to dynamic join and leave in the different streams corresponding to these channels [CRC+08]. A dynamic approach such as gossip seems rather adequate in this situation.

8.2.2 On Networking Issues

TCP-Friendliness Using gossip, the set of communication partners of a node changes every gossip period and since messages sent are small, it is quite natural to transfer them via UDP. Nevertheless, doing so can have a negative impact on other applications competing for bandwidth. In other words, none of the protocols proposed in this thesis are *TCP-friendly*. Our gossip-based protocol might simply take priority over other applications, similarly to most commercial voice-over-IP protocols. Making protocols using multiple incoming streams TCP-friendly is quite difficult assuming the serving nodes are static [WH01, MO07]. Doing the same for ever changing partners, such as with gossip, is therefore a problem on its own and needs further research.

Guarded Nodes Even though the Internet was imagined as an end-to-end connectivity model where each node could contact each other node in the network, the IPv4 soon reaches its limit in the address space and one way of limiting the use of public addresses is to hide many computers behind a single public address, namely IP masquerading. This is usually done with Network Address Translation (NAT) in such a way that a whole private network is *hidden* behind a router acting as a bridge between the private network and the Internet. This, with the emergence of software firewalls makes nodes harder to reach as they are *guarded*. The problem of limited reachability is solved in three very different ways: an explicit way and two transparent ways.

Explicit Port Forwarding In the router, the ports used by the application have to be manually opened and correctly forwarded to the corresponding computer. This is the most common way of benefiting fully from application and is especially the norm in applications such as BitTorrent clients (e.g., [Bit]), Wuala [Wua], or eMule [eMu]. These programs and the following procedures for opening and forwarding the correct ports to the correct computers demands certain skills that the common users do not have or, for good reasons, simply do not want to hear about.

NAT-traversal There exist techniques for punching holes in the router transparently [FSK05, RMMW08, MMR10] or creating a mesh in the presence of guarded peers [WJJ05]. Although these techniques work very well for *single* end-to-end communication such as VoIP (e.g., Skype [Sky]), it is not clear how these techniques behave in case each node eventually communicates with each other node in the network, such as in a gossip protocol.

NAT-avoidance At the application level, recent work [DOvNB07, KPQS09, LvRR10] shows how to avoid traversing NAT by relying on *gateway* nodes (themselves possibly forming a route of hole-punched gateways [KPQS09]) to direct the traffic to guarded nodes, providing that there exists gateways able to reach those nodes. Considering small updates, the load on gateways is negligible and the increase in their load stays reasonable. For high-bandwidth content dissemination, however, these proposals might not scale since gateways are loaded proportionally to the number of hosts for which they need to act as gateways. A solution could be to consider these gateways as being themselves source of content dissemination and thus create a sub-gossip in the private network for which they act as gateway, possibly reducing their load by a non negligible factor. However, their role resembles the role of supernodes and their impact in case of failure might be the disconnection of the whole private network.

In a live streaming application targeting the mass, it is not possible to ask every user to manually open its host on a router. There is therefore a need *(i)* at the network level, to traverse NAT in an efficient way even for a very large number of possibly communicating hosts, and if this proves to be impossible *(ii)* at the application level, research for achieving high-bandwidth content dissemination, possibly via gateway nodes.

Head-to-head Comparison with Trees and Mesh-based Systems In order to convince gossip-skeptical researchers that gossip is a clear alternative to trees and meshes for high-bandwidth content dissemination, there is a need to extend the work proposed in this thesis to do a head-to-head comparison with tree-based and mesh-based systems. Such a comparison is very challenging for many reasons.

Testbed We need a testbed that can show protocols scale, that is realistic, where experiments are reproducible and controlled. PlanetLab is the most realistic testbed since hosts are distributed over the Internet and thus its topology is inherently the one of the Internet. Although one can argue that hosts in Universities, institutions and research labs are connected to Internet backbones, the delays are realistic and it is the closest to reality that we can have as long as the bandwidth utilization of nodes is manually capped.

Unfortunately, PlanetLab nodes suffer from a heterogeneous and possibly very high load, mainly induced by ongoing experiments. The scale of the testbed is thus very much reduced when trying to deploy bandwidth-demanding applications. In addition, its load varies from time to time not providing reproducible experiments in the long run. The only viable option would be to run the different head-to-head applications at the same time with reduced bandwidth demand, which is of course not an optimal solution.

Simulations give reproducible results but when done with a realistic communication layer, the scale of the experiments is reduced to hundreds of nodes, not unleashing performance and possible issues of protocols with respect to scalability.

The best solution might be to run experiments on clusters, e.g., Grid'5000 [Gri], with a realistic Internet-like topology such as ModelNet [BCG$^+$] and thoroughly measure the bandwidth usage of nodes and make it comparable, be the protocols based on UDP with explicit retransmission such as *Claim*, or relying on TCP and its implicit retransmission, that indeed also uses measurable bandwidth.

Protocols Picking the correct protocols to compare with, reimplementing them or introducing in them a common evaluation metrics and deploying them is a very challenging task on its own. Thanks to Harry Li and Meng Zhang, we have the code of BAR Gossip/FlightPath [LCW$^+$06, LCM$^+$08] written in Python and of GridMedia [ZZSY07] written in C++ whereas all our code is written in Java. Comparing these approaches is not equitable since some are targeting performance while the others are targeting resilience to Byzantine nodes, showing a good example in the difficulty of picking competitors. In addition, some competitors are closed-source, e.g., Zattoo [Zat], PPLive [PPL], and reimplementing them ourselves is not only time-consuming but may be far from reality, knowing the many tricks programmers have sometimes to use to boost performance, possibly making the code cryptic. Beyond the fact that the programming language could have some impact on the execution of the experiments, it is very hard to dive into thousands of lines of code in different languages, add hooks for measuring common metrics and finally be able to deploy and run experiments on a testbed, hoping that the hooks themselves do not change the behavior nor performance of the implementation. The various parameters of the protocols might not have optimal values or optimal values depending on the experiment, e.g., the view size of mesh-based protocols in the presence of churn [LGL08].

Evaluation Metrics We defined our notion of stream quality and stream lag and used them as the two metrics throughout this thesis. Our notion of stream quality differs from the *average delivery ratio* used in [ZZSY07] or the *continuity index* used in [LXQ$^+$08]. There is yet no common evaluation metric in streaming, making it difficult to compare approaches quantitatively without adding hooks in the code and redo experiments. Whereas all researchers agree that receiving the maximum percentage of the original stream is optimal, opinions on how scattered the losses should be differ. The solution would be to force each system to provide a clear stream to all nodes and measure its minimum stream lag, similarly to file sharing where the common metrics is the time needed to receive the complete file. Unfortunately, this option seems unrealistic in a constrained environment in the presence of catastrophic failures or churn where nodes might periodically not receive a clear stream, be it on time or at all.

Experimental Setups It is difficult to give optimal values to protocols' parameters in the absolute, but it might be even harder to find optimal values for different experimental setups, e.g., different stream rates, upload bandwidths, number of nodes, characteristics of churn or percentage of failures and when they happen. It is clear that for relative low-bandwidth content dissemination, the cost of signaling in mesh-based or gossip systems will be very high compared to trees relative to the stream rate, whereas for high-bandwidth content dissemination, it will be considered in both cases negligible, but what is today or in the future a reasonable stream rate for watching TV over the Internet? For a free online system, low quality might be enough (and forced by the business model anyway), but for replacing regular TV broadcasting, broadcasters should aim at HDTV or even higher quality already.

How would one or the other solution compare with different upload capability distributions? What is indeed a realistic upload capability distribution and for how long?

If streaming is a free online system or replacement for regular TV broadcasting, the infrastructures and guarantees provided to clients are very different. But clients behaviors might also be very different in one or the other case, resulting in possibly very different churn or catastrophic failure scenarios. There exists traces of clients' behaviors in P2P systems resulting in synthetic but realistic traces, mainly for file sharing [SR06, GSS06]. Recent studies [ZZSY07, CRC+08] in the context of television might also eventually result in synthetic traces. But would such traces make sense for a free online TV systems with one channel? Are these traces realistic in free online systems with multiple channels? And finally in substitution of regular TV broadcasting? According to what has happened in the context of file sharing, it seems there needs to be a working system first, e.g., Gnutella, with many studies and with lessons learned, new systems can emerge, e.g., BitTorrent. The same behavior might also happen in the context of television, and we might need to wait for traces coming from commercial products such as PPLive [PPL] or Zattoo [Zat] to evaluate new systems such as gossip-based ones.

There is thus a lot to do in the context of comparison between existing approaches. Before even thinking about realistic conditions, we still lack a common platform and metrics for evaluating the different approaches. The first step towards this might be the creation of a simple benchmark, fixing once and for all common experimental conditions such as the stream (defining its length and stream rate), the upload bandwidth of the source, the number of nodes and their upload capabilities (defining the upload capabilities distribution) and imposing that all nodes must receive a clear stream (since there are no catastrophic failures nor churn). Starting from this common configuration, then define catastrophic failures, churn and observe from real demand if configurations depend on the end usage, e.g., free online service or replacement for regular TV broadcasting. Such a common benchmark is what unexpectedly happened in the field of image processing with the *Lenna* standard test image [Mun96, Ros] or more generally and commonly with the usage of SPEC benchmarks [SPE].

Network, Social and Byzantine Awareness Gossip is relying on an ever changing, dynamic random graph for establishing communications between nodes. With random communications, a node in Lausanne might infect a node in China which in turn might infect another node back in Lausanne. This unawareness of geographic location can be a real issue in systems where routing is not possible or very expensive, e.g., sensor networks. Even though the communications between nodes are usually short, as opposed to mesh-based systems, they can still represent a waste of resources in terms of network usage, basically used for routing purposes. We see at least two ways to resolve this issue:

Smart routing with smart routers We could imagine that with the emergence of new kinds of software-enabled routers [DEA$^+$09], it could be possible that a subset of what multicast was supposed to offer at the network layer could be implemented. We can imagine that some kinds of well-defined packets could be stored for a short amount of time in different routers level and that requests might not need to cross different continents before reaching a node, or a router that has the needed data. In short, when the first node in Lausanne serves the node in China, the data could be temporarily stored in a router close to Lausanne, and as soon as the node in China would need to serve the second node in Lausanne, the request of this second node would be intercepted by the router and directly served. Besides obvious scalability issues that would need to be solved, there exists in this case a clear incentive for ISPs to implement such a service since the data stays in their domain.

Network-aware Peer Sampling Service To favor collaboration between well-interconnected nodes, e.g., large bandwidth and low delay, random peer sampling services could be biased to favor networking neighbors in the partial views of nodes. By doing so, one has to keep in mind that not only should the gossip random graph stay connected but also that randomness is a key point in distributing the load (equally or heterogeneously) among the whole set of nodes. Disrupting this randomness should be done with caution so that different well-connected neighborhoods can still easily infect other neighborhoods, for instance, with the use of a hierarchical gossip [KMG03], or at least that a dedicated amount of their fanout is always reserved for unbiased partners.

The peer sampling service could also be aware of social links between nodes. In fact, nodes might be tempted to be more altruistic (or maybe less selfish) when collaborating with their *friends*. Friend-to-Friend (F2F) computing [PCT04, LD06] is already popular in file-sharing as modifications of BitTorrent [GADK10, IPKA] and a F2F-aware peer sampling service might also be tempting in order to possibly increase performance and reduce the impact of freeriders.

The peer sampling service might also need to be tolerant to Byzantine attacks, such as the Brahms peer sampling service, proposed in [BGK$^+$09] but no practical implementation of such systems exists yet. It might be that a F2F-aware peer sampling service is enough not to suffer from Byzantine attacks, but since some part of the peer sampling service needs to provide nodes with other completely random other nodes anyway, designing an enhanced peer sampling service (either in terms of network-awareness or F2F-awareness or both), resisting to Byzantine attacks and being practically implementable constitutes a real research challenge.

8.2.3 On Increasing Performance

Chunk Priorities Video content is composed of an audio track synchronized with a series of pictures, called frames that are compressed using different techniques. There exists typically three types of frames, I-frames, P-frames and B-frames. I-frames are the least compressible and do not require other video frames to decode, whereas P-frames can use data from previous frames to decode and B-frames can use data from both previous and forward frames to decode. B-frames are therefore subject to the highest amount of compression. From this simple classification, one can easily deduce that I-frames are more important than others, because they contain more information than I-frames and B-frames.

The fact that there exists an importance in the classification of frame types could be given to the protocol in order to favor certain chunks over the others. In other words, priorities could be given to chunks depending on their content importance in such a way that important packets could be advertised more, or retransmitted more stubbornly and with higher priority. This idea is partially present in NEEM [PRM$^+$03], where nodes keep long-lived TCP connections with the nodes in their partial views, forming a mesh. When nodes are forced to drop packets because of congestion, they choose them according to their semantics. Unfortunately, there exists currently no application bundling this priority information at the chunk level, e.g., in the MPEG transport stream packets [LCC07].

Dissemination Tree Prediction As pointed out in [ZLLY05, ZZSY07], the epidemic dissemination process results in a dissemination tree when nodes do not explicitly request content they already have. Gossip can therefore easily be used to build trees [LPR07, CPOR07] to achieve multicast. Even though such approaches rely on a gossip protocol keeping the random graph connected in case of churn, the use of a tree for dissemination suffers from reconstruction in case of failures or churn, and it is not clear the dissemination would behave better than with multiple-trees during reconstruction [CDK$^+$03]. In addition, creating a single tree trivially suffers from not asking leaf nodes to contribute.

While gossip makes the implicit dissemination trees ever changing, we could let communication partners request content from other nodes *before* they actually received the content. Since the dissemination protocol works in three phases, nodes could propose chunk ids of chunks they were proposed and have requested before they actually have delivered them. The goal is basically to reduce the time between reception of a chunk and the actual sending of it to another node.

By proposing the chunks they have requested before having received them, nodes have no guarantee that they will actually receive the requested chunk and thus eventually serve requesting nodes. While this proposal might improve dissemination in a rather static overlay, assuming it is possible to tune the retransmission mechanism to increase the retransmission timeouts in a smart way, it is not clear how it would affect performance in case of churn.

Additionally, in order to save a roundtrip communication time before the serve (i.e., the time to send a propose and receive the corresponding request), the source could directly push content to fanout nodes instead of proposing it first. While this can intuitively only improve performance, a failure detection feature is lost. In fact, since the source trivially knows that each of its proposals is of interest, the fact that a propose does not result in a request clearly means that the non-requesting node is overloaded, has crashed or left and therefore not a good candidate for being a forwarder at this stage of the dissemination. This simple feature is making sure that the first hop nodes chosen by the source are indeed capable of forwarding the produced content and could be used as a feedback mechanism in the peer sampling service.

Push then Pull It is well known that spreading a rumor first follows an exponential growth where a very large proportion of nodes rapidly learns about it, and then a shrinking phase where a little proportion of nodes has to wait longer until they can learn about it, with very few nodes possibly never learning about it.

It is therefore appealing to take advantage of the first exponential growth phase by disseminating content rapidly to a very large proportion of nodes and avoid the shrinking phase by *(i)* letting the missed content be reconstructed with error coding [CKS09] or *(ii)* letting the nodes pull the content that they did not yet receive, e.g., [KRAV03, BHO+99].

It would be interesting to see if both exponential growth and shrinking phases could be implemented by a push-based gossip for the first phase [CKS09, BHO+99] and a pull-based gossip for the second phase [KRAV03]. Because it is intuitively hard to know when a node should stop pushing content and when it should start pulling content and from whom, it is similarly hard to know if a node has the right to pull or if it was still supposed to push content. Such a fuzziness in the protocol could potentially encourage nodes to freeride: a node could simply try to pull as much data as it can by acting as if it is always in the shrinking phase.

A possible solution would be to gossip content in two *waves*. The first regular wave consists in proposing to fanout other nodes the content that nodes have just received (or requested, following the above proposition) and a second wave reproposing the same content a bit later (possibly with a lower fanout to decrease signaling overhead) so that nodes in the shrinking phase have the possibility to pull content from nodes having proposed it, that is, without pulling totally randomly and blindly. It is clear that such waves allow nodes not to serve content that was not proposed and that a verification protocol such as LiFT can be adapted to verify that nodes effectively execute the second wave.

8.2.4 On Adaptation to Heterogeneity

Source Decisions according to the Average Capability Thanks to a gossip-based aggregation, HEAP is able to adapt nodes' contribution according to their capabilities. The average capability is in fact not used by the source except for possibly biasing its partner selection towards rich nodes (as described in Section 6.4, page 88). In fact, we have seen that the maximum stream rate that can be disseminated is upper bounded by the average capability. The fact that the source has a very good approximation of this average capability can be used in different ways, such as:

Adapt Source Coding The source could gracefully increase or decrease the quality of the stream dynamically, by adapting its coding [WHZ+00], according to the average capability measured.

Restrict Joins Assuming a fixed stream rate, the source could select which nodes can join and which ones cannot, depending on their capabilities. A very rich node joining would help dissemination and a very poor one might possibly break it.

Different Adaptation Mechanisms In HEAP, we decided that all nodes should contribute in a way that is directly proportional to their capabilities. We can imagine that system designers would like that *(i)* the load distribution follows a different pattern or even that *(ii)* this pattern depends on the distribution of capabilities itself. To illustrate both solutions, we can imagine *(i)* that all nodes having an upload bandwidth smaller than the stream rate have to contribute as much as they can while richer nodes simply have to compensate as little as they need or oppositely that rich nodes contribute as much as possible in order to increase the dissemination speed and poor ones as little as possible, and *(ii)* the expected contribution of poor nodes can depend on their proportion, e.g., the more they are, the closer to their maximum capabilities they need to contribute.

We concentrated on the upload bandwidth of nodes in order to adapt their fanout and therefore their contribution. The gossiping period is another obvious knob to adapt contribution of nodes according to heterogeneity in capabilities. Other capabilities might be interesting to consider for other kinds of applications, such as storage quantity or memory in a distributed file system, for instance.

In fact, measuring capabilities is a problem on its own. Measuring free storage space seems rather easy but measuring available upload bandwidth is not, since it depends on many factors, such as when the node communicates, with whom, for how long or even how much bandwidth is used by other applications on the same machine or other devices on the same network. This is the reason why most P2P applications do not depend on the available upload bandwidth but simply give the users the possibility to limit the maximum bandwidth they want to let the application use, e.g., [Bit, eMu]. Accurate measurement of certain capabilities is therefore a very challenging issue which could help system designers take resources and their heterogeneity among nodes better into account.

Assuming we could accurately measure capabilities, gossip, in contrary to meshes, gives nodes the possibility of eventually communicating with every other node in the system. For scalability reasons, it is not possible to keep track of statistics with each other node in the system. Another challenge would therefore be to group nodes into classes according to their capabilities in such a way that *(i)* efficiency could be improved by biasing the dissemination from the source to richer nodes first and poorer nodes last by bringing some kind of structure into the random communication graph, and *(ii)* it could help the protocol be more network-aware, so that nodes that are close route-wise can make use of these properties (possibly using partial native multicast routes [ZWJ+06, Fra]).

Finally, we have seen that there is usually a clear correlation between large bandwidth and low average communication delay, the reason why traditional gossip can adapt to a certain extent (Section 4.2.1 page 43, Section 6.2 page 76, and pointed out in [DXL+06]). In order to take this phenomenon into account we can define the probability of acceptance p_{acc} as the ratio of the number of chunks requested to the number of chunks proposed and modulate the fanout adaptation of HEAP with this information. The probability of acceptance will be larger for a node with low average communication delay than for a node with high average communication delay. Taking p_{acc} into account requires aggregating the average probability of acceptance instead of the average bandwidth capability or both together so that the fanout is computed as $f = \overline{f}(b/\overline{b})(p_{acc}/\overline{p_{acc}})$.

8.2.5 On a More Robust Protocol

Punishing Freeriders Instead of simply expelling freeriders, we can imagine a system where freeriders benefit only if all honest notes have already benefited *correctly* from the system. The idea is rather intuitive: all honest nodes are served in priority and if there is some contribution left that can be distributed to freeriders, then they compete to have some benefit. Assuming gossip dissemination can be structured in such a way that some nodes can be placed closer to the source than others, e.g., by biasing the peer sampling service, putting nodes that cannot contribute a lot of resources or that deliberately do not contribute at the end of the dissemination acts as an incentive to encourage freeriders to indeed start to contribute. Effectively, since streaming has time constraints, a node that is always far from the source will suffer from an increased stream lag.

LiFT in a Heterogeneous Environment LiFT, in its current version, assumes that all nodes have the same fanout. However, put in the context of HEAP, nodes need to verify that other nodes use a fanout value that corresponds to their advertised capability. Admitting that this can be only an implementation detail in the case of static capabilities, the problem is a real challenge in the case where capabilities of nodes vary over time.

LiFT with Privileged Verifiers Nodes in LiFT have all the same roles of both managers and verifiers for each other. We can imagine injecting special nodes in the dissemination or picking some trusted nodes for a special role, namely *spying*. Spies in the dissemination are trusted by the source but act as normal nodes for every other node. The detection of freeriders could be facilitated as all obvious misbehaviors detected by spies could be taken for granted. Following the same line of reasoning, nodes could have more trust in their social friends than in other random nodes.

Oracle Measuring the Health of the System We have described in Chapter 7 that the p_{cc} parameter adapts the performance of LiFT and modulates its overhead accordingly. Assuming an oracle could measure the health of the system, it could dynamically increase (resp. decrease) this parameter in order to detect freeriders faster, increasing (resp. decreasing) the overhead of LiFT accordingly. Imagine that the source uses the average capability to have a measure of this health, it is clear that the source could instruct nodes to increase or decrease p_{cc} *publicly*, but the announcement of a decrease should be done only when the average capability is far above the stream rate, so that nodes starting to freeride after this announcement have only a very limited impact on the dissemination. Unfortunately, the presence of freeriders when the average capability is close to the stream rate is exactly where p_{cc} should be set to its maximum resulting in turn in a maximum overhead of LiFT.

There is thus possibly a need for a time-varying detection scheme, such as purging freeriders very fast at some stage, and then possibly release pressure on the nodes without letting nodes start to freeride once they know detection is relaxed.

LiFT without AVMON Managers in LiFT are assigned and maintained using the AVMON protocol [MG09]. Whereas AVMON provides every property needed by LiFT, we believe an architecture providing weaker properties would perfectly fit LiFT's needs. A simple example is AVMON's *verifiability* property. The fact that nodes can verify that a manager is indeed a regular and *authorized* manager of its managed nodes is not needed by LiFT. The weaker property we need is that there exists at least a single manager of a node (in its m managers) that is not colluding with the managed node. This property ensures that there exists at least one manager regularly maintaining the score of the node and ready to expel it if needed. In essence, we believe that there must exist a protocol matching LiFT's requirements, which compared to AVMON, would be simpler, provide weaker guarantees, and possibly achieve better performance.

LiFT & TfT Beyond gossip protocols, LiFT could be used to secure the asymmetric component of TfT-based protocols, namely *opportunistic unchoking*, which is considered to constitute their Achile's heel [LMSW06, SPCY07]. We can consider for instance a gossip protocol, secured by LiFT, to disseminate fresh chunks in the system, coupled with a protocol based on symmetric exchanges to complete the dissemination using traditional swarming with TfT incentives.

◊◊◊

Abbreviations

A	ADSL	Asymmetric Digital Subscriber Line
	Amazon S3	Amazon Simple Storage Service (see [DHJ$^+$07])
	apcc	*A posteriori* Cross-checking (LiFT, Chapter 7)
	ARQ	Automatic Repeat reQuest (see RFC3366 [FW02])
C	CPU	Central Processing Unit
	CSR	Capacity Supply Ratio (see [ZZSY07])
D	dc	Direct Check (LiFT, Chapter 7)
	DHT	Distributed Hash Table (see [RD01,RFS$^+$01,SMK$^+$01,ZHS$^+$04])
F	F2F	Friend-to-Friend (see [PCT04, LD06, GADK10, IPKA])
	FEC	Forward Error Correction (see [Riz97])
H	HDTV	High Definition TV, typically 1080p format
	HEAP	HEterogeneity-Aware gossip Protocol
I	IP	Internet Protocol
	IPv4	Fourth revision in the development of the Internet Protocol
	ISP	Internet Service Provider
L	LiFT	Lightweight Freerider-Tracking Protocol
M	MDC	Multiple Description Coding (see [GKAV98, PR99, Goy01])
	MP3	MPEG-1 Audio Layer 3
	MPEG	Moving Picture Experts Group
	MPEG TS	MPEG Transport Stream
	MTU	Maximum Transmission Unit
N	NAT	Network Address Translation
P	P2P	Peer-to-Peer
R	RFC	Request For Comment
T	TCP	Transmission Control Protocol
	TfT	Tit-for-Tat (see [Coh03])
	TTL	Time to live
U	UDP	User Datagram Protocol
V	VLC	VideoLAN Client (see [Vid])
	VoD	Video on Demand
	VoIP	Voice over Internet Protocol

Bibliography

[ADH05] André Allavena, Alan Demers, and John E. Hopcroft. Correctness of a Gossip Based Membership Protocol. In *PODC*, 2005.

[AGBH03] Luc Onana Alima, Ali Ghodsi, Per Brand, and Seif Haridi. Multicast in DKS(N, k, f) Overlay Networks. In *OPODIS*, 2003.

[AGR06] Siddhartha Annapureddy, Christos Gkantsidis, and Pablo Rodriguez. Providing Video-on-Demand using Peer-to-Peer Networks. In *IPTV*, 2006.

[AH00] Eytan Adar and Bernardo Huberman. Free riding on Gnutella. *First Monday*, 5(10), 2000.

[ATW02] John G. Apostolopoulos, Wai-tian Tan, and Susie J. Wee. Video Streaming: Concepts, Algorithms, and Systems. Technical report, HP Laboratories Palo Alto, 2002.

[BB05] Stefan Birrer and Fabian E. Bustamante. The Feasibility of DHT-based Streaming Multicast. In *MASCOTS*, 2005.

[BCG$^+$] David Becker, Jeff Chase, Diwaker Gupta, Dejan Kostić, Priya Mahadevan, Amin Vahdat, Kashi Vishwanath, Kevin Walsh, and Ken Yocum. Modelnet. `http://modelnet.sysnet.ucsd.edu`.

[BDHU09] Michael Backes, Peter Druschel, Andreas Haeberlen, and Dominique Unruh. CSAR: A Practical and Provable Technique to Make Randomized Systems Accountable. In *NDSS*, 2009.

[BGK$^+$09] Edward Bortnikov, Maxim Gurevich, Idit Keidar, Gabriel Kliot, and Alexander Shraer. Brahms: Byzantine Resilient Random Membership Sampling. *Computer Networks*, 53:2340–2359, 2009.

[BGKM07] Sébastien Baehni, Rachid Guerraoui, Boris Koldehofe, and Maxime Monod. Towards Fair Event Dissemination. In *ICDCSW*, 2007.

[BHO$^+$99] Kenneth Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal Multicast. *TOCS*, 17(2):41–88, 1999.

[Bit] BitTorrent, Inc. μTorrent. `http://www.utorrent.com/`.

[BMM+08] Thomas Bonald, Laurent Massoulié, Fabien Mathieu, Diego Perino, and Andrew Twigg. Epidemic Live Streaming: Optimal Performance Trade-offs. In *SIGMETRICS*, 2008.

[Bol01] Béla Bollobás. *Random Graphs*. Cambridge University Press, 2001.

[BPLCH09] Fadi Boulos, Benoît Parrein, Patrick Le Callet, and David Hands. Perceptual Effects of Packet Loss on H.264/AVC Encoded Videos. In *VPQM*, 2009.

[BRP+05] Ashwin R. Bharambe, Sanjay G. Rao, Venkata N. Padmanabhan, Srinivasan Seshan, and Hui Zhang. The Impact of Heterogeneous Bandwidth Constraints on DHT-Based Multicast Protocols. In *IPTPS*, 2005.

[BRS06] Michael Bishop, Sanjay Rao, and Kunwadee Sripanidulchai. Considering Priority in Overlay Multicast Protocols under Heterogeneous Environments. In *INFOCOM*, 2006.

[CDK+03] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony I. T. Rowstron, and Atul Singh. SplitStream: High-bandwidth Multicast in Cooperative Environments. In *SOSP*, 2003.

[CDKR02] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony I. T. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *JSAC*, 20(8):100–110, 2002.

[CE07] Niklas Carlsson and Derek L. Eager. Peer-Assisted On-Demand Streaming of Stored Media Using BitTorrent-Like Protocols. In *Networking*, 2007.

[CGN+04] Yang-hua Chu, Aditya Ganjam, T. S. Eugene Ng, Sanjay G. Rao, Kunwadee Sripanidkulchai, Jibin Zhan, and Hui Zhang. Early Eexperience with an Internet Broadcast System Based on Overlay Multicast. In *ATEC*, 2004.

[CJW09] Hyunseok Chang, Sugih Jamin, and Wenjie Wang. Live Streaming Performance of the Zattoo Network. In *IMC*, 2009.

[CKS09] Mary-Luc Champel, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. FoG: Fighting the Achilles' Heel of Gossip Protocols with Fountain Codes. In *SSS*, 2009.

[Coh03] Bram Cohen. Incentives Build Robustness in BitTorrent. In *P2P Econ*, 2003.

[CPOR07] Nuno Carvalho, José Pereira, Rui Carlos Oliveira, and Luís Rodrigues. Emergent Structure in Unstructured Epidemic Multicast. In *DSN*, 2007.

[CRC⁺08] Meeyoung Cha, Pablo Rodriguez, Jon Crowcroft, Sue Moon, and Xavier Amatriain. Watching Television Over an IP Network. In *IMC*, 2008.

[CRSZ02] Yang-hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. A Case for End System Multicast. *JSAC*, 20(8):1456–1471, 2002.

[DC90] Stephen E. Deering and David R. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *TOCS*, 8(2):85–110, 1990.

[DEA⁺09] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *SOSP*, 2009.

[Dee88] Stephen E. Deering. Multicast Routing in Internetworks and Extended LANs. In *SIGCOMM*, 1988.

[DGH⁺87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic Algorithms for Replicated Database Maintenance. In *PODC*, 1987.

[DHJ⁺07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, 2007.

[DHRS07] Prithula Dhungel, Xiaojun Hei, Keith W. Ross, and Nitesh Saxena. The Pollution Attack in P2P Live Video Streaming: Measurement Results and Defenses. In *P2P-TV*, 2007.

[DLHC05] Chris Dana, Danjue Li, David Harrison, and Chen-Nee Chuah. BASS: Bittorrent Assisted Streaming System for Video-on-Demand. In *MMSP*, 2005.

[Dou02] John R. Douceur. The Sybil Attack. In *IPTPS*, 2002.

[DOvNB07] Niels Drost, Elth Ogston, Rob van Nieuwpoort, and Henri Bal. ARRG: Real-World Gossiping. In *HPDC*, 2007.

[DSW06] Alexandros G. Dimakis, Arnand D. Sarwate, and Marting J. Wainwright. Geographic Gossip: Efficient Aggregation for Sensor Networks. In *IPSN*, 2006.

[DXL⁺06] Mayur Deshpande, Bo Xing, Iosif Lazardis, Bijit Hore, Nalini Venkatasubramanian, and Sharad Mehrotra. CREW: A Gossip-based Flash-Dissemination System. In *ICDCS*, 2006.

[EGH⁺03] Patrick T. Eugster, Rachid Guerraoui, Sidath B. Handurukande, Petr Kouznetsov, and Anne-Marie Kermarrec. Lightweight Probabilistic Broadcast. *TOCS*, 21(4):341–374, 2003.

[EGKM04] Patrick T. Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulié. Epidemic Information Dissemination in Distributed Systems. *Computer*, 37(5):60–67, 2004.

[eMu] eMule. The eMule Project. `http://www.emule-project.net`.

[FFM04] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with Coral. In *NSDI*, 2004.

[FFM07] Yaacov Fernandess, Antonio Fernández, and Maxime Monod. A Generic Theoretical Framework for Modeling Gossip-Based Algorithms. *OSR*, 41(5):19–27, 2007.

[FGK$^+$09a] Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Boris Koldehofe, Martin Mogensen, Maxime Monod, and Vivien Quéma. Heterogeneous Gossip. In *Middleware*, 2009.

[FGK$^+$09b] Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Maxime Monod, and Vivien Quéma. Stretching Gossip with Live Streaming. In *DSN*, 2009.

[FGKM10] Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, and Maxime Monod. Boosting Gossip for Live Streaming. In *P2P*, 2010.

[Fra] Paul Francis. Yoid: Extending the Internet Multicast Architecture. `http://www.icir.org/yoid/`.

[FSK05] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-Peer Communication Across Network Address Translators. In *ATEC*, 2005.

[FW02] Gorry Fairhurst and Lloyd Wood. Advice to link designers on link Automatic Repeat reQuest (ARQ). RFC 3366, Network Working Group, 2002.

[GADK10] Wojciech Galuba, Karl Aberer, Zoran Despotovic, and Wolfgang Kellerer. Leveraging social networks for increased BitTorrent robustness, 2010.

[GGH$^+$06] Benoît Garbinato, Rachid Guerraoui, Jarle Hulaas, Maxime Monod, and Jesper H. Spring. Frugal Mobile Objects. In *EAWMSN (DCOSS Workshops)*, 2006.

[GGH$^+$07a] Benoît Garbinato, Rachid Guerraoui, Jarle Hulaas, Alexei Kounine, Maxime Monod, and Jesper Honig Spring. The Weight-Watcher Service and its Lightweight Implementation. In *IC-SAMOS*, 2007.

[GGH$^+$07b] Benoît Garbinato, Rachid Guerraoui, Jarle Hulaas, Maxime Monod, and Jesper H. Spring. Pervasive Computing with Frugal Objects. In *PCAC*, 2007.

[GHH$^+$06] Rachid Guerraoui, Sidath B. Handurukande, Kévin Huguenin, Anne-Marie Kermarrec, Fabrice Le Fessant, and Etienne Rivière. GosSkip, an Efficient, Fault-Tolerant and Self Organizing Overlay Using Gossip-based Construction and Skip-Lists Principles. In *P2P*, 2006.

[GHK$^+$] Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec, Maxime Monod, and Ymir Vigfusson. Decentralized Polling with Respectable Participants. *Distributed Computing*. Under submission.

[GHK$^+$10] Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec, Maxime Monod, and Swagatika Prusty. LiFTinG: Lightweight Freerider-Tracking Protocol in Gossip. In *Middleware*, 2010.

[GHKM09a] Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec, and Maxime Monod. Brief Announcement: Towards Secured Distributed Polling in Social Networks. In *DISC*, 2009.

[GHKM09b] Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec, and Maxime Monod. Decentralized Polling with Respectable Participants. In *OPODIS*, 2009.

[GHKM09c] Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec, and Maxime Monod. On Tracking Freeriders in Gossip Protocols. In *P2P*, 2009.

[GKAV98] Vivek K. Goyal, Jelena Kovacevic, Ramon Arean, and Martin Vetterli. Multiple Description Transform Coding of Images. *ICIP*, 1998.

[GKM03] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Peer-to-Peer Membership Management for Gossip-Based Protocols. *TC*, 52(2):139–149, 2003.

[GLL08] Yang Guo, Chao Liang, and Yong Liu. Adaptive Queue-based Chunk Scheduling for P2P Live Streaming. In *Networking*, 2008.

[GMS04] Christos Gkantsidis, Milena Mihail, and Amin Saberi. Random Walks in Peer-to-peer Networks. In *INFOCOM*, 2004.

[Goy01] Vivek K. Goyal. Multiple Description Coding: Compression Meets the Network. *SPM*, 18(5):74–94, 2001.

[Gri] Grid'5000. The ADT ALADDIN-G5K Initiative. `http://www.grid5000.fr`.

[GSS06] P. Brighten Godfrey, Scott Shenker, and Ion Stoica. Minimizing churn in distributed systems. In *SIGCOMM*, 2006.

[Ham07] James Hamilton. On Designing and Deploying Internet-Scale Services. In *LISA*, 2007.

[Har68] Garrett Hardin. The Tragedy of the Commons. *Science*, 162:1243–1248, 1968.

[HHL88] Sandra M. Hedetniemi, Stephen T. Hedetniemi, and Arthur L. Liestman. A survey of gossiping and broadcasting in communication networks. *Networks*, 18(1):319–349, 1988.

[HHL06] Zygmunt J. Haas, Joseph Y. Halpern, and Li Li. Gossip-Based Ad Hoc Routing. *TON*, 14(3):479–491, 2006.

[HJVR08] Maya Haridasan, Ingrid Jansch-Porto, and Robbert Van Renesse. Enforcing Fairness in a Live-Streaming System. In *MMCN*, 2008.

[HKD07] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical Accountability for Distributed Systems. In *SOSP*, 2007.

[HLL$^+$07] Xiaojun Hei, Chao Liang, Jian Liang, Yong Liu, and Keith Ross. A Measurement Study of a Large-Scale P2P IPTV System. *TMM*, 9(8), 2007.

[IPKA] Tomas Isdal, Michael Piatek, Arvind Krishnamurthy, and Thomas Anderson. OneSwarm. `http://oneswarm.cs.washington.edu`.

[JMB04] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Detection and Removal of Malicious Peers in Gossip-Based Protocols. In *FuDiCo*, 2004.

[JMB05] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-Based Aggregation in Large Dynamic Networks. *TOCS*, 23(3):219–252, 2005.

[JMB09] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-Man: Gossip-based fast overlay topology construction. *ComNet*, 53(13):2321–2339, 2009.

[JVG$^+$07] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based Peer Sampling. *TOCS*, 25(3):1–36, 2007.

[KKD04] David Kempe, Jon Kleinberg, and Alan Demers. Spatial Gossip and Resource Location Protocols. *JACM*, 51(6):943–967, 2004.

[KKU08] Murat Karakaya, İbrahim Körpeoğlu, and Özgür Ulusoy. Counteracting Free-riding in Peer-to-Peer Networks. *ComNet*, 52(3):675–694, 2008.

[KLR07] Rakesh Kumar, Yong Lin, and Keith Ross. Stochastic Fluid Theory for P2P Streaming Systems, 2007.

[KMG03] Anne-Marie Kermarrec, Laurent Massoulié, and Ayalvadi Ganesh. Probabilistic Reliable Dissemination in Large-Scale Systems. *TPDS*, 14(3):248–258, 2003.

[Kol03] Boris Koldehofe. Buffer management in probabilistic peer-to-peer communication protocols. In *SRDS*, 2003.

[KPQS09] Anne-Marie Kermarrec, Alessio Pace, Vivien Quéma, and Valerio Schiavoni. NAT-resilient Gossip Peer Sampling. In *ICDCS*, 2009.

[KRAV03] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. In *SOSP*, 2003.

[KS04] Valerie King and Jared Saia. Choosing a Random Peer. In *PODC*, 2004.

[KSSV00] Richard Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vöcking. Randomized Rumor Spreading. In *FOCS*, 2000.

[KSTT04] Ramayya Krishnan, Michael Smith, Zhulei Tang, and Rahul Telang. The Impact of Free-Riding on Peer-to-Peer Networks. In *HICSS*, 2004.

[LCC07] Jin Li, Yi Cui, and Bin Chang. PeerStreaming: design and implementation of an on-demand distributed streaming system with digital rights management capabilities. *MultiSys*, 13(3):173–190, 2007.

[LCM$^+$08] Harry Li, Allen Clement, Mirco Marchetti, Manos Kapritsos, Luke Robinson, Lorenzo Alvisi, and Mike Dahlin. FlightPath: Obedience vs. Choice in Cooperative Services. In *OSDI*, 2008.

[LCW$^+$06] Harry Li, Allen Clement, Edmund Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, and Michael Dahlin. BAR Gossip. In *OSDI*, 2006.

[LD06] Jinyan Li and Frank Dabek. F2F: reliable storage in open networks. In *IPTPS*, 2006.

[LGL08] Chao Liang, Yang Guo, and Yong Liu. Is Random Scheduling Sufficient in P2P Video Streaming? In *ICDCS*, 2008.

[LM99] Meng-Jang Lin and Keith Marzullo. Directional gossip: Gossip in a wide area network. In *EDCC*, 1999.

[LMSW06] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free Riding in BitTorrent is Cheap. In *HotNets*, 2006.

[LMSW07] Thomas Locher, Remo Meier, Stefan Schmid, and Roger Wattenhofer. Push-to-Pull Peer-to-Peer Live Streaming. In *DISC*, 2007.

[LOM94] Kurt Lidl, Josh Osborne, and Joseph Malcolm. Drinking from the Firehose: Multicast USENET News. In *UWC*, 1994.

[LPR07] João Leitão, José Pereira, and Luís Rodrigues. Epidemic Broadcast Trees. In *SRDS*, 2007.

[LRLZ08] Jiangchuan Liu, Sanjay G. Rao, Bo Li, and Hui Zhang. Opportunities and Challenges of Peer-to-Peer Internet Video Broadcast. *Proc. of the IEEE*, 96(1), 2008.

[LvRR10] João Leitão, Robbert van Renesse, and Luís Rodrigues. Balancing Gossip Exchanges in Networks with Firewalls. In *IPTPS*, 2010.

[LXQ$^+$08] Bo Li, Susu Xie, Yang Qu, Gabriel Y. Keung, Chuang Lin, Jiangchuan Liu, and Xinyan Zhang. Inside the New Coolstreaming: Principles, Measurements and Performance Implications. In *INFOCOM*, 2008.

[MG09] Ramsés Morales and Indranil Gupta. AVMON: Optimal and Scalable Discovery of Consistent Availability Monitoring Overlays for Distributed Systems. *TPDS*, 20(4):446–459, 2009.

[MKR06] Maxime Monod, Jörg Kienzle, and Alexander Romanovsky. Looking Ahead in Open Multithreaded Transactions. In *ISORC*, 2006.

[MMR10] Rohan Mahy, Philip Matthews, and Jonathan Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766, Internet Engineering Task Force, 2010.

[MO07] Lin Ma and Wei Ooi. Congestion Control in Distributed Media Streaming. In *INFOCOM*, 2007.

[MR07] Nazanin Magharei and Reza Rejaie. Mesh or Multiple-Tree: A Comparative Study of Live P2P Streaming Approaches. In *INFOCOM*, 2007.

[MR09a] Nazanin Magharei and Reza Rejaie. Overlay Monitoring and Repair in Swarm-based Peer-to-Peer Streaming. In *NOSSDAV*, 2009.

[MR09b] Nazanin Magharei and Reza Rejaie. PRIME: Peer-to-Peer Receiver-Driven Mesh-Based Streaming. *TON*, 17(4):1052–1065, 2009.

[MTGR07] Laurent Massoulié, Andrew Twigg, Christos Gkantsidis, and Pablo Rodriguez. Randomized Decentralized Broadcasting Algorithms. In *INFOCOM*, 2007.

[Mun96] David C. Munson. A Note on Lena. *TIP*, 5(1):3, 1996.

[OKJ09] Anis Ouali, Brigitte Kerherve, and Brigitte Jaumard. Toward Improving Scheduling Strategies in Pull-based Live P2P Streaming Systems. In *CCNC*, 2009.

[PCT04] Bogdan C. Popescu, Bruno Crispo, and Andrew S. Tanenbaum. Safe and Private Data Sharing with Turtle: Friends Team-Up and Beat the System. In *SPW*, 2004.

[PKT$^+$05] Vinay Pai, Kapil Kumar, Karthik Tamilmani, Vinay Sambamurthy, and Alexander E. Mohr. Chainsaw: Eliminating Trees from Overlay Multicast. In *IPTPS*, 2005.

[Pla] PlanetLab. The Trustees of Princeton University. `http://www.planet-lab.org`.

[PM08] Fabio Picconi and Laurent Massoulié. Is There a Future for Mesh-based Live Video Streaming? In *P2P*, 2008.

[PPKB07] Fabio Pianese, Diego Perino, Joaquín Keller, and Ernst W. Biersack. PULSE: An Adaptive, Incentive-Based, Unstructured P2P Live Streaming System. *TOM*, 9(8):1645–1660, 2007.

[PPL] PPLive. Pplive. `http://www.pplive.com`.

[PR99] Rohit Puri and Kannan Ramchandran. Multiple Description Source Coding through Forward Error Correction Codes. In *ACSSC*, 1999.

[PRM$^+$03] José Orlando Pereira, Luís Rodrigues, M. João Monteiro, Rui Carlos Oliveira, and Anne-Marie Kermarrec. NEEM: Network-Friendly Epidemic Multicast. In *SRDS*, 2003.

[PvS10] Guillaume Pierre and Maarten van Steen. *Handbook of Research on P2P and Grid Systems for Service-Oriented Computing: Models, Methodologies and Applications*, chapter Enforcing Fairness in Asynchronous Collaborative Environments. IGI Global, 2010.

[RD01] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware*, 2001.

[RFS$^+$01] Sylvia Ratnasamy, Paul Francis, Scott Shenker, Richard Karp, and Mark Handley. A Scalable Content-Addressable Network. In *SIGCOMM*, 2001.

[RHP$^+$03] Luis Rodrigues, Sidath Handurukande, Jose Pereira, Rachid Guerraoui, and Anne-Marie Kermarrec. Adaptive gossip-based broadcast. In *DSN*, 2003.

[Riz97] Luigi Rizzo. Effective Erasure Codes for Reliable Computer Communication Protocols. *CCR*, 27(2):24–36, 1997.

[RMMW08] Jonathan Rosenberg, Rohan Mahy, Philip Matthews, and Dan Wing. Session Traversal Utilities for NAT (STUN). RFC 5389, Network Working Group, 2008.

[Ros] Chuck Rosenberg. The Lenna Story. `http://www.lenna.org`.

[SBR06] Yu-Wei Sung, Michael Bishop, and Sanjay Rao. Enabling Contribution Awareness in an Overlay Broadcasting System. *CCR*, 36(4):411–422, 2006.

[SDK$^+$07] Kyoungwon Suh, Christophe Diot, Jim Kurose, Laurent Massoulié, Christoph Neumann, Donald F. Towsley, and Matteo Varvello. Push-to-peer video-on-demand system: Design and evaluation. *JSAC*, 25(9):1706–1716, 2007.

[SGMZ04] Kunwadee Sripanidkulchai, Aditya Ganjam, Bruce Maggs, and Hui Zhang. The Feasibility of Supporting Large-Scale Live Streaming Applications with Dynamic Application End-Points. In *SIGCOMM*, 2004.

[Sky] Skype Limited. Skype. `http://www.skype.com`.

[SMK$^+$01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *SIGCOMM*, 2001.

[SPBP06] Neil Spring, Larry Peterson, Andy Bavier, and Vivek Pai. Using Planetlab for Network Research: Myths, Realities, and Best Practices. *OSR*, 40(1):17–24, 2006.

[SPCY07] Michael Sirivianos, Jong Park, Rex Chen, and Xiaowei Yang. Freeriding in BitTorrent with the Large View Exploit. In *IPTPS*, 2007.

[SPE] SPEC. Standard Performance Evaluation Corporation. `http://www.spec.org/`.

[SR06] Daniel Stutzbach and Reza Rejaie. Understanding Churn in Peer-to-Peer Networks. In *IMC*, 2006.

[VF06] Vivek Vishnumurthy and Paul Francis. On Heterogeneous Overlay Construction and Random Node Selection in Unstructured P2P Networks. In *INFOCOM*, 2006.

[VF07] Vivek Vishnumurthy and Paul Francis. A Comparison of Structured and Unstructured P2P Approaches to Heterogeneous Random Peer Selection. In *ATEC*, 2007.

[VGK$^+$07] Nevena Vratonjić, Priya Gupta, Nikola Knežević, Dejan Kostić, and Antony I. T. Rowstron. Enabling DVD-like Features in P2P Video-on-demand Systems. In *P2P-TV*, 2007.

[VGvS05] Spyros Voulgaris, Daniela Gavidial, and Maarten van Steen. CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. *JNSM*, 13(2):197–217, 2005.

[Vid] VideoLAN - VLC media player. The VideoLAN Team. `http://www.videolan.org`.

[vR00] Robbert van Renesse. Scalable and secure resource location. In *HICSS*, 2000.

[vRBV03] Robbert van Renesse, Kenneth Birman, and Werner Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System

Monitoring, Management, and Data Mining. *TOCS*, 21(2):164–206, 2003.

[vRMH98] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Middleware*, 1998.

[VYF06] Vidhyashankar Venkataraman, Kaouru Yoshida, and Paul Francis. Chunkyspread: Heterogeneous Unstructured Tree-Based Peer-to-Peer Multicast. In *ICNP*, 2006.

[WH01] Jörg Widmer and Mark Handley. Extending Equation-based Congestion Control to Multicast Applications. In *SIGCOMM*, 2001.

[WHZ$^+$00] Dapeng Wu, Yiwei Thomas Hou, Wenwu Zhu, Hung-Ju Lee, Ti-Hao Chiang, Ya-Qin Zhang, and H. Jonathan Chao. On End-to-End Architecture for Transporting MPEG-4 Video Oover the Internet. *TCSVT*, 10(6):923–941, 2000.

[WHZ$^+$01] Dapeng Wu, Yiwei Thomas Hou, Wenwu Zhu, Ya-Qin Zhang, and Jon M. Peha. Streaming Video over the Internet: Approaches and Directions. *TCSVT*, 11:282–300, 2001.

[WJJ05] Wenjie Wang, Cheng Jin, and Sugih Jamin. Network Overlay Construction Under Limited End-to-End Reachability. In *INFOCOM*, 2005.

[Wua] Wuala. Wuala by LaCie. `http://www.wuala.com`.

[You] Youtube. Youtube, LLC. `http://www.youtube.com`.

[Zat] Zattoo. Zattoo. `http://www.zattoo.com`.

[Zho09] Lidong Zhou. Building reliable large-scale distributed systems: when theory meets practice. *SIGACT News*, 40(3):78–85, 2009.

[ZHS$^+$04] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *JSAC*, 22:41–53, 2004.

[ZLLY05] Xinyan Zhang, Jiangchuan Liu, Bo Li, and Tak-shing Peter Yum. Coolstreaming/DONet: A Data-driven Overlay Network for Peer-to-Peer Live Media Streaming. In *INFOCOM*, 2005.

[ZWJ$^+$06] Beichuan Zhang, Wenjie Wang, Sugih Jamin, Daniel Massey, and Lixia Zhang. Universal IP multicast delivery. *ComNet*, 50(6):781–806, 2006.

[ZZSY07] Meng Zhang, Qian Zhang, Lifeng Sun, and Shiqiang Yang. Understanding the Power of Pull-Based Streaming Protocol: Can We Do Better? *JSAC*, 25(9):1678–1694, 2007.

List of Figures, Algorithms and Tables

About the Author

Maxime Monod (full name Ducimetière Alias Monod) was born on May 6th 1981 in Lausanne of mother Francine Bonfils (born Perret) and father Jean-Daniel Ducimetière Alias Monod. He joins a family composed of his brother Cédric Monod (1974) and his sister Leslie Monod-Mounoud (1977). His family originates from Corsier-sur-Vevey and Saint-Saphorin-sur-Morges. After the divorce of his parents, Maxime is also raised by Guy Bonfils, fondly considered as his second father.

Besides drama classes, piano, guitar, karate and tennis, it is after the buying of a Commodore 64 that Maxime is infected for good by the virus of computer science.

After having graduated from the *Gymnase Auguste Piccard* (CESSrive high school), where he obtained a *baccalauréat ès sciences* and a *maturité fédérale* (Science bias), he studies computer science at the Swiss Federal Institute of Technology in Lausanne (EPFL). He graduates in 2004 with an Engineering Diploma, after completing his diploma project (Master's thesis) at McGill University, Montréal under the supervision of both Prof. Alfred Strohmeier (Software Engineering Laboratory, EPFL) and Prof. Jörg Kienzle (Software Engineering Laboratory, McGill University). After completing the Officier Candidate School in the armored forces in Thun, he joins the Distributed Programming Laboratory of EPFL, led by Prof. Rachid Guerraoui in the beginning of 2005.

He first works two years towards the elaboration of an event-based programming model in the context of mobile devices for the European project PALCOM, and then begins his PhD in the context of large-scale distributed systems, partially funded by the SNF. He supervised several semester projects, master projects, and internships. He also acted four years as a teaching assistant for the *Introduction to Object-oriented Programming* course, given to undergraduate students in computer and communication sciences.

After several years playing basketball in Pully, Maxime enjoys playing tennis, running, hiking, alpine skiing, ski touring, biking, bike touring and participates in various races and competitions, in particular: 20 km de Lausanne, Lausanne Marathon, Morat-Fribourg, Sierre-Zinal, Swiss Raid Commando, Lombardia Raid, Monaco Raid, Patrouille des Glaciers or Cyclotour du Léman.

A Propos de l'Auteur

Maxime Monod (nom de famille complet Ducimetière Alias Monod) est né le 6 mai 1981 à Lausanne de mère Francine Bonfils (née Perret) et de père Jean-Daniel Ducimetière Alias Monod. Il rejoint une famille composée de son frère Cédric Monod (1974) et de sa soeur Leslie Monod-Mounoud (1977). Il est originaire de Corsier-sur-Vevey et Saint-Saphorin-sur-Morges. Après le divorce de ses parents, Maxime est aussi élevé par Guy Bonfils, qu'il considère affectueusement comme son deuxième père.

A côté de cours de théâtre, de piano, de guitare, de karaté et de tennis, c'est probablement après l'achat d'un Commodore 64 que Maxime a définitivement attrapé le virus de l'informatique.

Après avoir obtenu son baccalauréat ès sciences et sa maturité fédérale au Gymnase Auguste Piccard (anciennement CESSrive), en section scientifique, il étudie l'informatique à l'Ecole Polytechnique Fédérale de Lausanne. Il obtient en 2004 le diplôme d'Ingénieur en Informatique en effectuant son projet de diplôme à *McGill University*, Montréal sous la supervision du Prof. Alfred Strohmeier (Laboratoire de Génie Logiciel, EPFL) et du Prof. Jörg Kienzle (*Software Engineering Laboratory*, *McGill University*). Après l'école d'officiers dans les troupes blindées à Thoune, il rejoint début 2005 le Laboratoire de Programmation Distribuée de l'EPFL, dirigé par le Prof. Rachid Guerraoui.

Il travaille tout d'abord deux ans à l'élaboration d'un modèle de programmation événementielle dans le contexte des environnements mobiles au sein du projet européen PALCOM, puis commence sa thèse sur les systèmes distribués à large échelle pour l'obtention de son doctorat, partiellement financé par le FNS. Il a supervisé plusieurs projets de semestre, projets de master ou encore encadré des stagiaires. Pendant quatre ans, il a aussi été assistant pour le cours *Introduction à la Programmation Objet*, donné aux étudiants de première année des sections informatique et systèmes de communications.

Après plusieurs années de basket à Pully, Maxime pratique le tennis, la course à pied, la randonnée, le ski alpin, la peau de phoque, le vélo, le cyclotourisme et participe à différentes courses et compétitions, notamment les 20 km de Lausanne, le marathon de Lausanne, Morat-Fribourg, Sierre-Zinal, le Swiss Raid Commando, le Lombardia Raid, le Monaco Raid, la Patrouille des Glaciers ou encore le Cyclotour du Léman.

"That's all folks!"