Boosting Gossip for Live Streaming

Davide Frey‡, Rachid Guerraoui*, Anne-Marie Kermarrec‡, Maxime Monod*
*Ecole Polytechnique Fédérale de Lausanne
‡INRIA Rennes-Bretagne Atlantique

***Abstract*—Gossip protocols are considered very effective to disseminate information in a large scale dynamic distributed system. Their inherent simplicity makes them easy to implement and deploy. However, whereas their probabilistic guarantees are often enough to disseminate data in the context of low-bandwidth applications, they typically do not suffice for high-bandwidth content dissemination: missing 1% is unacceptable for live streaming.**

In this paper, we show how the combination of two simple mechanisms copes with this seemingly inherent deficiency of gossip: *(i) codec*, an erasure coding scheme, and *(ii) claim*②, a content-request scheme that leverages gossip duplication to diversify the retransmission sources of missing information. We show how these mechanisms can effectively complement each other in a new gossip protocol, *gossip*++, which retains the simplicity of deployment of plain gossip. In a realistic setting with an average bandwidth capability (800 kbps) close to the stream rate (680 kbps) and 1% message loss, plain gossip can provide at most 99% of the stream. Using *gossip*++, on the other hand, all nodes can view a perfectly clear stream.

I. Introduction

Since it was first adopted as a technique for maintaining replicated databases [1], the gossip paradigm has been recognized as efficient and practical in a variety of contexts. These include publish-subscribe, application-level multicast, and overlay construction and maintenance [1]–[7]. In all these cases, gossip has shown to be simple, lightweight and extremely resilient to churn.

These great properties of gossip are mainly the result of the redundancy it natively offers in data dissemination. Specifically, in the infect-and-die model [8], each node forwards each gossip message exactly once to fanout other nodes chosen at random from the set of all n nodes. Clearly, the larger the fanout value, the greater the redundancy offered by the protocol as nodes can receive each message from multiple sources. Theory suggests that the fanout value should be chosen as $f = \ln(n) + c$, where c is a constant, defining the probability of the gossip protocol to result in a connected random graph as $\exp(-\exp(-c))$. In other words, f can be chosen such that all messages are gossiped to all nodes *with high probability* (*whp*). Specifically, larger fanout values allow for a trade-off between the probability of obtaining atomic dissemination, and the cost of the protocol in terms of consumed bandwidth.

However, these results have to be revised when gossip is applied in the context of high-bandwidth content dissemination, such as file sharing [4] or video streaming [9], [10]. In such a setting, gossip protocols normally adopt a three-phase approach to maintain communication costs within reasonable limits. Let us consider the case of video streaming. The source splits the content to be disseminated into *chunks*. Then, each node, including the source, advertises the new content it has by gossiping the corresponding chunk identifiers to fanout other nodes (first phase). A node receiving such an advertisement message replies by *requesting* the chunks it needs among those proposed (second phase). Finally, nodes receiving a request for a given chunk reply by sending the actual payload, i.e., *serving* the node (third phase).

This mode of operation brings significant changes in the behavior of gossip protocols in the presence of message loss or churn. First, the redundancy inherent in gossip is only present in the first phase of the protocol. As a result, the protocol is only resilient to the loss of messages advertising available chunks, but not to the loss of the chunks themselves. Second, the quality of the disseminated stream is constrained by the average upload bandwidth of nodes, which is commonly considered the bottleneck in such collaborative systems. As a result, content-dissemination protocols should produce the smallest possible overhead and be able to operate near the limits of available bandwidth.

In the context of gossip, this means that the protocol cannot arbitrarily increase its fanout even to guarantee atomic dissemination during the first phase. Indeed, as shown in [11], too large fanout values easily saturate the scarce bandwidth resources of participating nodes and result in significant performance losses.

On the other hand, the best performance is, in this case, achieved with fanout values that are only slightly larger than $\ln(n)$. Such small values allow the gossip 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 a naive three-phase solution to deliver the entirety of the available content to all participating nodes.

To exemplify this effect, we ran a set of experiments disseminating a video stream of 680 kpbs to 200 nodes and evaluated the behavior of the three phases of the described gossip protocol with several fanout values. All nodes except the source have a bandwidth cap of 800 kbps and the source's fanout is set to 5 and its bandwidth usage is therefore roughly 5 times the stream rate. 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 percentage 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

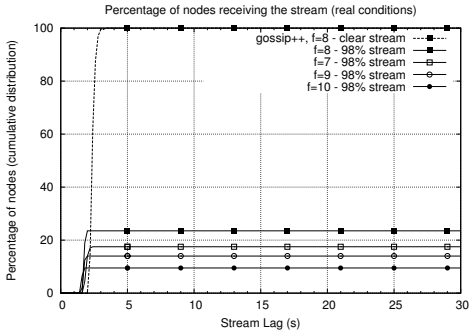

Fig. 1. In a realistic scenario with constrained bandwidth (800 kbps bandwidth cap) and additional message losses (1%), increasing the fanout of standard gossip does not help. On the other hand, gossip++ can deliver a clear stream to all nodes.

theoretical results about the reliability of gossip, 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, as depicted in Figure 1, 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. The stream lag represents *how live* the stream is played, i.e., the time elapsed between the sending of the stream from the source and the actual playback on the nodes. 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 [12].

In this paper, we show how these issues can be addressed through the proper combination of two simple mechanisms: (i) *Codec*, an erasure coding scheme, and (ii) *Claim*②, a content-request scheme that leverages gossip duplication to diversify the retransmission sources of missing information. *Codec* operates by adding redundant coded chunks to the stream so that it can be reconstructed after the loss of a random subset of its chunks. *Claim*②, 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.

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*② 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*.

II. Background & Related Work

In its original form, the gossip paradigm is an implementation of a broadcast primitive that provides a message from the source to all nodes in the system with high probability. A source node randomly chooses a set of communication partners, i.e., *fanout* nodes and sends them a message. Upon receipt of a message, nodes forward it once to *fanout* other nodes, also chosen at random. Since each node forwards the message only once, we say the algorithm follows an *infect-and-die* model.

A. Gossip for high-bandwidth content dissemination

Among its various applications, gossip has also been used 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 messages to be transmitted. Then it begins dissemination using a gossip-based protocol. The problem is that nodes generally cannot afford gossiping these chunks directly due to their large sizes. 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 [13] 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 TTL). It additionally and continuously codes the file and sends new coded chunks with the same TTL. Once a node has received enough chunks to decode the file (i.e., k random chunks) it codes it and starts to gossip newly coded 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 makes it possible 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 the nodes that have completed the first phase to contribute much more than the others, possibly fostering freeriding vocations.

B. Three-phase gossip

To address the problem of bandwidth utilization more effectively, an appealing solution is to use a three-phase gossip protocol [4], [9], [11] as depicted in Figure 2. Its characteristic is the ability to exploit gossip's redundancy on small *propose* messages, while receiving the actual payload only once. This makes the model very appealing in high-bandwidth content-dissemination applications such as streaming.

The three phases are as follows.

- *Propose phase.* Periodically, i.e., every gossip period, each node picks a *new* set of f (fanout) other nodes uniformly at random. This is usually achieved using a random peer sampling protocol [5], [14], [15]. It then

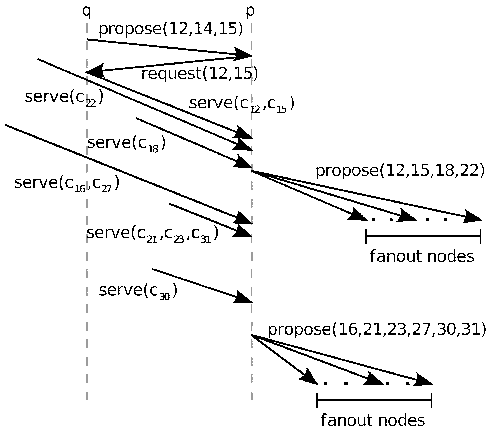

Fig. 2. Three-phase gossip protocol with an infect-and-die behavior.

proposes, to each of them, the identifiers of the chunks it received since its last propose phase. This is illustrated in Figure 2, 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, 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 2, 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 2, node q serves p with chunks c_{12} and c_{15}.

Such a scheme has been successfully used in several applications. The work in [4] was the first, to the best of our knowledge, to propose the use of such a protocol for file sharing. More recently, the same scheme has been used in systems [9], [10] designed to tolerate Byzantine nodes based on symmetric exchanges *à la* Tit-for-Tat [16]. In particular, the work in [9] also recognizes the need for forward error correction (FEC). However, different from the approach we present, it uses a very low gossip fanout with chunks being proposed for multiple rounds and requires as much as 100% of coding, i.e., 50% of the data sent represents the original stream and 50% the added coded data. Its use of large gossip periods enables the use of TCP connections between peers, thereby preventing the need for explicit retransmission. However, the approach focuses on being tolerant to byzantine attacks and not on performance. Specifically, the authors do not provide any results in constrained environments, and more recent work has recognized scalability issues [17].

C. Alternative approaches to dissemination

Clearly, gossip is not the only viable mechanism for high-bandwidth content dissemination and video streaming. Initial work on streaming adopted tree-based approaches [18]–[21]. However, most work has focused on more redundant mesh structures [22]–[25]. Some mesh-based systems split the initial stream into multiple substreams and disseminate them by creating multiple trees over the mesh [26]. Others use the mesh structure directly without superimposing specific dissemination paths [27]. In mesh-based systems, an important challenge is to devise the optimal *scheduling algorithm* [28], i.e., picking which chunks to send to whom in a node's neighborhood, in order to both ensure a high quality stream for individual reception and fast and efficient spreading of the stream in the whole system. In gossip, the scheduling problem is clearly reduced by design. A node has no incentive to wait until it receives multiple proposals for a chunk in order to decide which node to request from, because it is impossible for a node to maintain statistics of how nodes behaved in the past in order to request from the *optimal* node, since the set of partners change dynamically. In addition, a node has no guarantee that it will receive multiple proposals for the same chunk (if any) and in what order it will receive the chunks themselves. On the other hand, recent work [11] has showed that gossip, in its most dynamic form, provides an effective, simple and implicit means to split the stream into multiple random dissemination paths without having to explicitly split the stream among multiple paths. This makes gossip particularly appealing in the presence of message loss and failures.

III. GOSSIP++

We consider a system in which a source broadcasts a stream with the aid of a three-phase gossip protocol. The source has enough bandwidth to serve at most s nodes ($s = 5$ in our experiments), while all other nodes have their upload bandwidth limited to b, with b only slightly larger than the stream rate. The rationale is that dissemination protocols must impose the smallest overhead on top of the original stream so that for a given bandwidth capacity, the system provides the nodes with the best stream possible [23]. Nodes communicate over lossy links (e.g., UDP). Every node can receive incoming data from any other node in the system (i.e., the nodes are not guarded/firewalled, or there exists means to circumvent such protections [29], [30]). Nodes can fail by crashing, or exhibit freeriding behavior, that is, decrease their contributions while still benefiting from the system. Other types of attacks such as those in which nodes inject junk content (e.g., pollution attacks [10], [31]) are outside the scope of this paper.

In order to provide reliable dissemination of streaming content, we augment the three-phase gossip protocol with two components: *Codec* and *Claim*② as described in the following.

A. 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 a shorter 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).

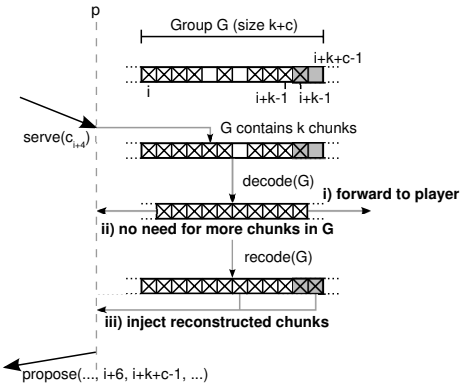

Fig. 3. Codec: a node p receiving k chunks in G decodes the group to reconstruct the k source chunks and sends them to the player (step (i)). Node p then signals the protocol not to request any more chunks in G (step (ii)). Optionally (step (iii)), p reencodes the k source events and injects reconstructed chunks into the protocol.

Erasure coding (FEC): The source of the stream uses a *block-based* FEC implementation [32] to create, for each group of k source chunks, c additional coded 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 3). 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).

The cost of FEC: The cost of using FEC mainly consists of network cost. 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 3). The decoding process then provides the node with the k source chunks needed to play the videostream. 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 to serve useful chunks to nodes in need. Optionally (step (iii) in Figure 3), in order not to stop abruptly the dissemination of the reconstructed chunks (source or coded chunks: chunks $i+6$ and $i+k+c-1$ in that case), nodes can reinject decoded chunks into the protocol.[1] The performance improvement of this step is evaluated and discussed in Section IV-G.

B. Claim②

While Codec can reconstruct missing chunks, it still needs at least k chunks from each group. Claim② 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 (a-la TCP), the requesting node rerequests nodes in the set of proposing nodes in a round-robin manner, as presented in Figure 4.

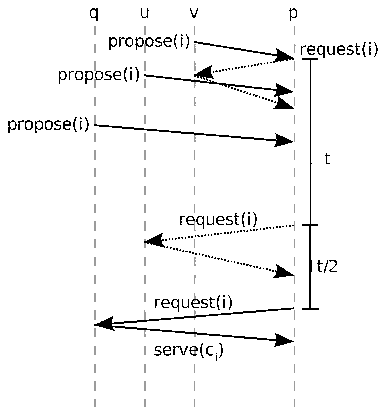

Fig. 4. Claim②: Node v has proposed chunk i to node p which requested it. Either the request or the serve was lost and p, instead of reasking v now requests u, that also proposed chunk i. If u fails to serve c_i, p requests chunk i again from another node that proposed i. Node q finally serves p with 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 $\mu + 3.29\sigma$ where μ and σ are respectively the average and standard deviation of the roundtrip times experienced by the node (representing the 99.9^{th} percentile in a normal distribution).[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.

IV. Evaluation

We evaluated gossip++ on 200 nodes, deployed over 40 Grid'5000 machines. The use of a cluster platform like

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

[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.

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. 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 200 ms and all messages are sent over UDP.

A. Metrics

We evaluated 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 [12]. 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 [33].

When using plain gossip without *Codec*, playing 100% of the chunks requires receiving every single chunk disseminated by the source. When using *Codec*, on the other hand, a node can play all chunks if it receives at least k random chunks per group of $k+c$ chunks. When such a 100% quality is impossible to reach, we define a jitter percentage of the stream as 100% minus the percentage of correctly received packets [23].

B. Overview

In the following, we present the results of our evaluation. We first show in Section IV-C 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*②. We then evaluate *Claim*② alone as well as in combination with *Codec*. Again, *Claim*② alone proves to be insufficient, but its combination with *Codec* provides all nodes with a clear stream even in highly constrained settings. In Section IV-E we evaluate the impact of different FEC percentages in combination with *Claim*②. This allows us to identify 5% FEC as the best trade-off. Finally, we push the analysis further and examine performance *(i)* when the bandwidth gets more and more constrained (Section IV-F), *(ii)* in the presence of nodes that cannot or do not provide their fair share of work, i.e., freeriders (Section IV-G), and *(iii)* in the presence of catastrophic crash failures (Section IV-H).

C. 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 [11], 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 discussed in Section I, 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.

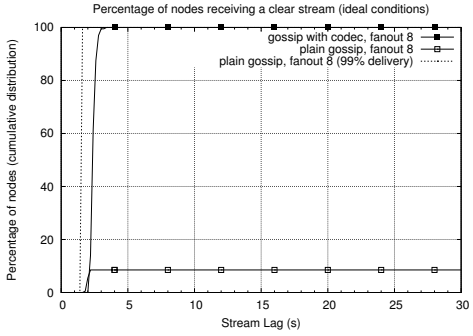

Fig. 5. In an ideal scenario where bandwidth is unconstrained and without message loss, plain gossip delivers a clear stream to only 8.5% of nodes since all nodes do not receive a proposal for each chunk. Adding FEC on the other hand, 100% of the nodes can view the original stream.

To get a better understanding of this poor performance, we ran an experiment with a fanout of 8 in an unconstrained bandwidth scenario without message loss. Results are depicted in Figure 5: 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.

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), confirming the theoretical results that each chunk proposal (followed by its actual payload) reaches all nodes with high probability.

D. Realistic conditions: Need for Claim②

When moving to realistic conditions, however, it becomes clear that *Codec* 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.

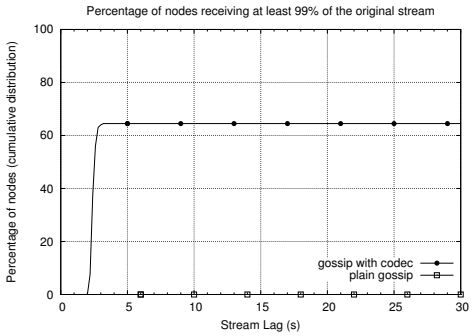

Fig. 6. Without FEC, no node can even receive 99% of the original stream with constrained bandwidth and message losses. Using *Codec*, there is a large improvement since 64.5% of the nodes receive at least 99% of the original stream with a stream lag shorter than 3.3 s.

These results show that, while *Codec* is able to address the inability of gossip to reach all nodes with a fanout of 8, it is not sufficient to recover from message losses resulting from the communication layer and the associated bandwidth constraints.The situation, instead, improves dramatically if we add *Claim*② to the picture. Figure 7 shows that *Claim*② combined with *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 recover from more missing chunks, be them not proposed or lost. We thus also show, in Figure 7, results for 50% coding (resp. 100% coding), i.e., where 2/3 (resp. 1/2) 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*② 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*② guarantees that a node that received proposals for a chunk will eventually receive the chunk when reclaiming it from one of the proposing nodes.

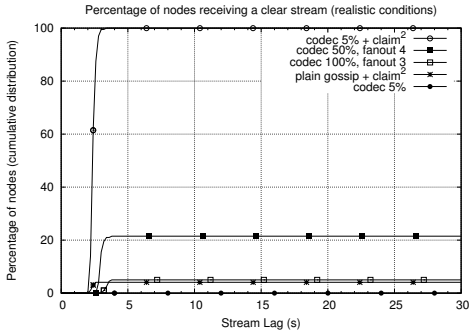

Fig. 7. While *Codec* (5% coding) can provide a 1% jittered stream to 64.5% of nodes (Figure 6) it cannot, alone, provide a clear stream to any node. *Claim*② improves plain gossip only a very little but significantly boosts the performance of *Codec*. When applied together the two techniques provide a clear stream to all nodes. Finally, increasing the percentage of FEC without retransmission does not help in recovering missing chunks.

However, with *Claim*② a node will never be able to retrieve chunks for which it never got a proposal.

E. 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*② in several scenarios with a bandwidth cap of 800 kbps, and message loss rates ranging from no message loss to 5% 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 8, 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% message loss. With 5% 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.

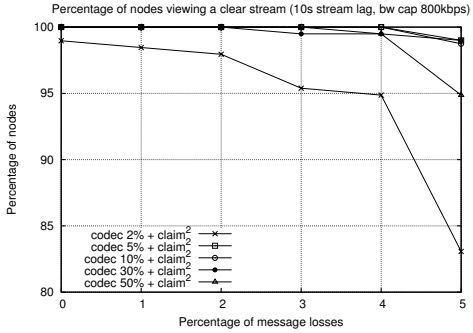

Fig. 8. *Codec* with 2% and 50% coding provide a clear stream to a lower number of nodes than other FEC percentages as message loss percentage increases.

These figures can be better understood by analyzing the data in Figure 9. The plot shows the cumulative distribution of stream lag for the various FEC 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: nodes try to send more data than they are allowed to by their bandwidth limiters. This leads to dropped messages and retransmissions ultimately explaining the poor results with respect to stream lag of *Codec* 30% and *Codec* 50%.

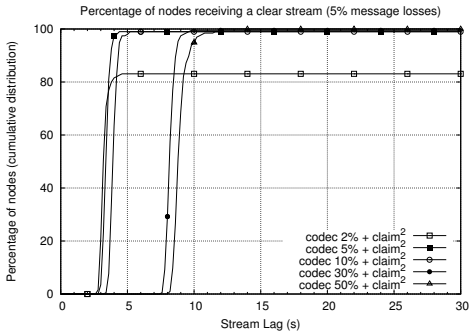

Fig. 9. With 5% message loss, *Codec* with 2% coding provides a clear stream to a maximum of 83% of nodes. Other percentages reach at least 98.7% but with different stream lags. The bandwidth used by *Codec* with 30% and 50% coding is larger than with lower coding percentages (Figure 10). The traffic excess is limited by the token bucket, which results in needing *Claim*② to recover missing chunks, which in turns, explains the larger stream lag.

These observations are confirmed by Figure 10, showing 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 9.

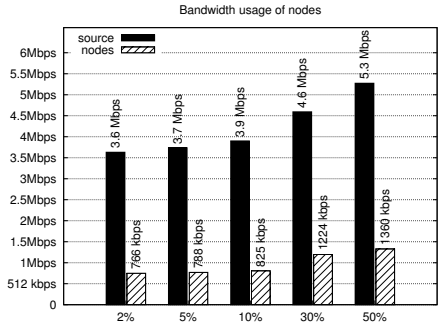

Fig. 10. Bandwidth usage of both source and tentative bandwidth usage of nodes for different percentages of coding. The source bandwidth usage increases according to the FEC overhead and the tentative bandwidth usage of nodes quickly exceed the bandwidth cap of 800 kbps, explaining the bad performances in Figure 8 for 30% and 50% coding.

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*② to combine 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 11. The plot shows the percentage of chunks that *Codec* was able to reconstruct with each of the five coding percentages. The depicted percentage includes both the chunks for which no proposal was received and the requests/re-requests that were saved, i.e., the missing chunks that 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.

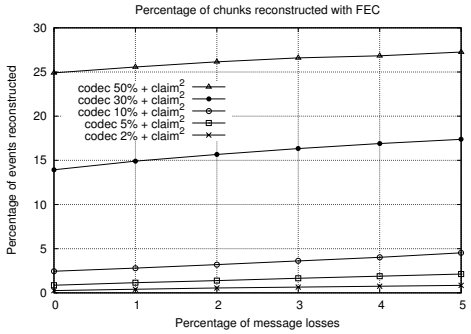

Fig. 11. The percentage of chunks recovered with erasure coding increases with the message-loss percentage at a rate that is largely independent of the specific FEC value.

F. On Bandwidth Constraints

Next, 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 been successfully decoded). The plot in Figure 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

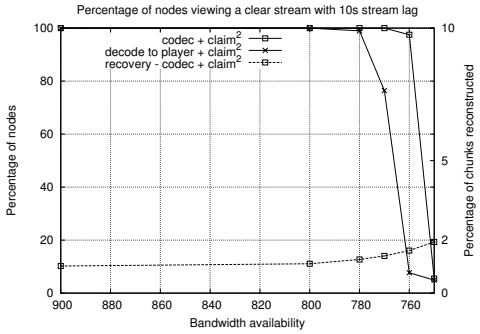

Fig. 12. By decreasing the available bandwidth of nodes, nodes tend to lose more messages due to bursts and therefore need both *Codec* and *Claim*② to deliver a clear stream. Once the available bandwidth gets below 770 kbps not all nodes can view a clear stream with gossip++ anymore. Interestingly, *Codec* is more and more effective in avoiding requests and re-requests until the system collapses.

dropped by the bandwidth limiter, which in turn causes a larger proportion of chunks to be recovered by the *Codec* mechanism.

G. 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 requested any 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 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 contributing. For this reason we ran the following experiments with a larger upload bandwidth of 1000 kbps.

1) Passive Freeriders: Figure 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 reinjects the reconstructed source chunks it did not receive *regularly*, and *reencodes* the k source chunks into c coded chunks that it also reinjects in the dissemination process (step (iii) in Figure 3).

The plot in Figure 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.

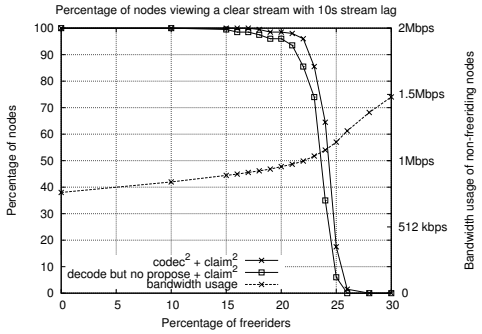

Fig. 13. Increasing the percentage of freeriders results in decreasing the average fanout. *Codec*② performs slightly better than *Codec*.

The third line in Figure 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*② to issue more and more rerequests, which in turn get more and more likely to be dropped. The data that nodes attempt to send 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% freeriding nodes can be tolerated with approximately 20% of slack in the upload bandwidth, i.e., 1000 kbps instead of 800 kbps. This highlights the scalability in terms of bandwidth consumption of the combination *Codec*② + *Claim*②.

2) Active Freeriders: Figure 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*② and a standard retransmission mechanism such as ARQ [34], labeled as "retransmission" in the plot. Specifically, while *Claim*②'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*②.

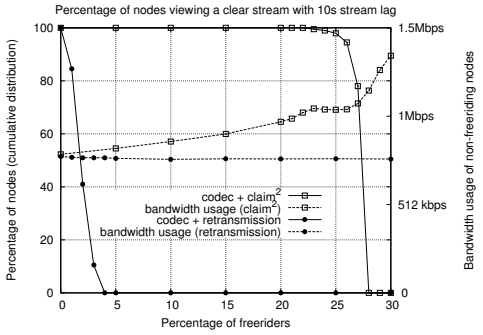

Fig. 14. When the percentage of freeriders increases, *Claim*② is able to shift the contribution that freeriders should provide to non-freeriding nodes, providing good performance as long as their bandwidth usage remains under the bandwidth cap. A standard retransmission mechanism, on the other hand, is not able to shift the load to non-freeriding nodes. Its bandwidth usage therefore remains constant and significantly lower than the allowed bandwidth.

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 low as 4% of active freeriding nodes.

*Claim*② on the other hand is very effective in having non-freeriding nodes compensate for up to 25% 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.

H. Crashes

Finally, we consider the behavior of gossip++ in a catastrophic-failure scenario. Figure 15 reports a zoomed-in view of the percentage of nodes delivering each chunk around the moment at which 20% or 50% of the nodes crash.

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 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.

V. Limitations and Future Work

In this paper we presented gossip++, an integration to gossip consisting of *Codec*, a FEC encoding mechanism,

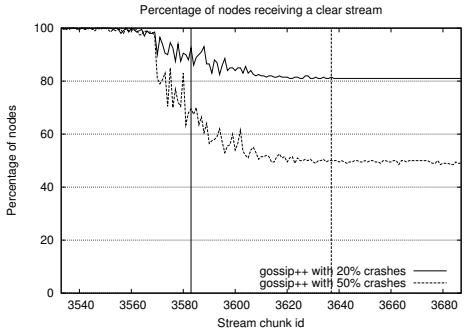

Fig. 15. While 20% (resp. 50%) of the nodes crash between chunks 3583 and 3637 (700 ms duration in a video of about 15 minutes), the percentage of nodes that receive each chunk drops from chunk 3570 on and lasts less than 1 s.

and *Claim*②, 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. In spite of these good performance figures, there are still a number of directions along which gossip++ can be improved.

A. Coding

Codec is based on FEC with systematic coding so that even if there are missing chunks in a group, some content can be delivered to the player resulting only in a jittered group instead of a whole undecoded group. Nevertheless, it would be interesting to see how network coding could be used in a gossip context, since recent work [25] has shown its efficiency for live streaming in meshes.

B. Hostile Nodes

The first major direction to consider is that of hostile nodes: such as byzantine and freeriding ones.

1) Byzantine Attacks: Byzantine nodes can attack gossip in a number of different ways. For example, they can attack the underlying random peer sampling [35] or they can pollute the protocol thereby decreasing the performance of dissemination and increasing the load on all nodes [10]. For example, this can translate to injecting junk data, i.e., pollution attacks [31] or imposing an arbitrary load on nodes until they fail, possibly leading to denial of service.

2) Freeriders: Freeriding nodes can essentially be of two types. First, there are nodes that cannot contribute to the protocol because of firewalls or because they do not have enough available bandwidth. Then, there are nodes that actively choose not to collaborate while attempting to benefit from the system.

In both cases, we have shown that gossip++ can distribute the load of non participating nodes to other nodes as long

as they have enough bandwidth to do so. However, gossip++ cannot do anything in the case of scarce bandwidth, i.e., when the stream rate is very close to the outgoing bandwidth capability. We therefore recognize the need for detecting freeriders as in [17], [36], [37] and plan to complement gossip++ with a lightweight mechanism to detect and expel freeriders.

C. Bandwidth Heterogeneity

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 ADSL connections. It would therefore be interesting to see how techniques such as *Codec* and *Claim*② could improve protocols such as [38].

VI. Conclusion

The work we presented in this paper 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*②, which is explicitly designed to leverage the redundancy that is inherent in gossip dissemination. We showed that, when combined with *Codec*, *Claim*② 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.

Acknowledgements

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see http://www.grid5000.fr).

Maxime Monod has been partially funded by the Swiss National Science Foundation with grant 200021-113825.

References

[1] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic Algorithms for Replicated Database Maintenance," in *PODC*, 1987.

[2] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky, "Bimodal Multicast," *TOCS*, vol. 17, no. 2, pp. 41–88, 1999.

[3] R. van Renesse, Y. Minsky, and M. Hayden, "A gossip-style failure detection service," in *Middleware*, 1998.

[4] M. Deshpande, B. Xing, I. Lazardis, B. Hore, N. Venkatasubramanian, and S. Mehrotra, "CREW: A Gossip-based Flash-Dissemination System," in *ICDCS*, 2006.

[5] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "Gossip-based Peer Sampling," *TOCS*, vol. 25, no. 3, pp. 1–36, 2007.

[6] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec, "Lightweight Probabilistic Broadcast," *TOCS*, vol. 21, no. 4, pp. 341–374, 2003.

[7] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, "Randomized gossip algorithms," *TIT*, vol. 52, no. 6, pp. 2508–2530, June 2006.

[8] A.-M. Kermarrec, L. Massoulié, and A. J. Ganesh, "Probabilistic Reliable Dissemination in Large-Scale Systems," *TPDS*, vol. 14, no. 3, pp. 248–258, 2003.

[9] H. Li, A. Clement, M. Marchetti, M. Kapritsos, L. Robinson, L. Alvisi, and M. Dahlin, "FlightPath: Obedience v.s. Choice in Cooperative Services," in *OSDI*, 2008.

[10] H. Li, A. Clement, E. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin, "BAR Gossip," in *OSDI*, 2006.

[11] D. Frey, R. Guerraoui, A.-M. Kermarrec, M. Monod, and V. Quéma, "Stretching Gossip with Live Streaming," in *DSN*, 2009.

[12] F. Boulos, B. Parrein, P. Le Callet, and D. Hands, "Perceptual Effects of Packet Loss on H.264/AVC Encoded Videos," in *VPQM*, 2009.

[13] M.-L. Champel, A.-M. Kermarrec, and N. Le Scouarnec, "FoG: Fighting the Achilles' Heel of Gossip Protocols with Fountain Codes," in *SSS*, 2009.

[14] V. King and J. Saia, "Choosing a Random Peer," in *PODC*, 2004.

[15] C. Gkantsidis, M. Mihail, and A. Saberi, "Random Walks in Peer-to-peer Networks," in *INFOCOM*, 2004.

[16] B. Cohen, "Incentives Build Robustness in BitTorrent," in *P2P Econ*, 2003.

[17] M. Haridasan, I. Jansch-Porto, and R. van Renesse, "Enforcing Fairness in a Live-Streaming System," in *MMCN*, 2008.

[18] Y.-H. Chu, S. Rao, and H. Zhang, "A Case for End System Multicast," *JSAC*, vol. 20, no. 8, pp. 1456–1471, 2000.

[19] V. Padmanabhan, H. Wang, P. Chou, and K. Sripanidkulchai, "Distributing streaming media content using cooperative networking," in *NOSSDAV*, 2002.

[20] M. Bawa, H. Deshpande, and H. Garcia-Molina, "Transience of peers and streaming media," in *HotNets-I*, 2002.

[21] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable application layer multicast." in *SIGCOMM*, 2002.

[22] B. Li, Y. Qu, Y. Keung, S. Xie, C. Lin, J. Liu, and X. Zhang, "Inside the New Coolstreaming: Principles, Measurements and Performance Implications," in *Proc. of INFOCOM*, 2008.

[23] M. Zhang, Q. Zhang, L. Sun, and S. Yang, "Understanding the Power of Pull-Based Streaming Protocol: Can We Do Better?" *JSAC*, vol. 25, no. 9, pp. 1678–1694, 2007.

[24] N. Magharei and R. Rejaie, "PRIME: Peer-to-Peer Receiver-Driven Mesh-Based Streaming," *TON*, vol. 17, no. 4, pp. 1052–1065, 2009.

[25] M. Wang and B. Li, "R^2: Random Push with Random Network Coding in Live Peer-to-Peer Streaming," *JSAC*, vol. 25, no. 9, pp. 1655–1666, 2007.

[26] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "SplitStream: High-bandwidth Multicast in Cooperative Environments," in *SOSP*, 2003.

[27] F. Picconi and L. Massoulié, "Is There a Future for Mesh-based Live Video Streaming?" in *P2P*, 2008.

[28] C. Liang, Y. Guo, and Y. Liu, "Is random scheduling sufficient in p2p video streaming?" in *ICDCS*, 2008.

[29] A.-M. Kermarrec, A. Pace, V. Quéma, and V. Schiavoni, "NAT-resilient Gossip Peer Sampling," in *ICDCS*, 2009.

[30] W. Wang, C. Jin, and S. Jamin, "Network Overlay Construction Under Limited End-to-End Reachability," in *INFOCOM*, 2005, pp. 2124–2134.

[31] P. Dhungel, X. Hei, K. W. Ross, and N. Saxena, "The pollution attack in p2p live video streaming: measurement results and defenses," in *P2P-TV*, 2007.

[32] L. Rizzo, "Effective Erasure Codes for Reliable Computer Communication Protocols," *CCR*, vol. 27, no. 2, pp. 24–36, 1997.

[33] J. Li, Y. Cui, and B. Chang, "Peerstreaming: design and implementation of an on-demand distributed streaming system with digital rights management capabilities," *MultiSys*, vol. 13, no. 3, pp. 173–190, 2007.

[34] G. Fairhurst and L. Wood, "Advice to link designers on link Automatic Repeat reQuest (ARQ)," Network Working Group, RFC 3366, 2002.

[35] E. Bortnikov, M. Gurevich, I. Keidar, G. Kliot, and A. Shraer, "Brahms: Byzantine Resilient Random Membership Sampling," in *PODC*, 2008.

[36] R. Guerraoui, K. Huguenin, A.-M. Kermarrec, and M. Monod, "On Tracking Freeriders in Gossip Protocols," in *P2P*, 2009.

[37] J. J. D. Mol, J. A. Pouwelse, M. Meulpolder, D. H. J. Epema, and H. J. Sips, "Give-to-Get: Free-riding resilient Video-on-Demand in P2P Systems," in *MMCN*, 2008.

[38] D. Frey, R. Guerraoui, A.-M. Kermarrec, B. Koldehofe, M. Mogensen, M. Monod, and V. Quéma, "Heterogeneous Gossip," in *Middleware*, 2009.