

# Live Streaming with Gossip

Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Maxime Monod

▶ **To cite this version:**

Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Maxime Monod. Live Streaming with Gossip. [Research Report] RR-9039, Inria Rennes Bretagne Atlantique; RR-9039. 2017. hal-01479885

**HAL Id: hal-01479885**
**https://inria.hal.science/hal-01479885**

Submitted on 1 Mar 2017

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.




# Live Streaming with Gossip

**Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Maxime Monod**

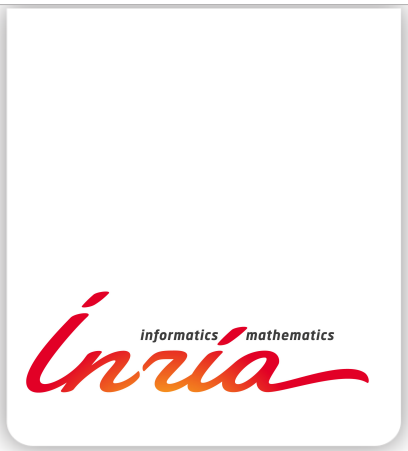
**RESEARCH REPORT**

**N° 9039**

March 2017

Project-Teams ASAP

ISSN 0249-6399 ISRN INRIA/RR--9039--FR+ENG




# Live Streaming with Gossip

Davide Frey*, Rachid Guerraoui†, Anne-Marie Kermarrec‡,

Maxime Monod§

Project-Teams ASAP

Research Report n° 9039 — March 2017 — 43 pages

**Abstract:** Video streaming has become a killer application for peer-to-peer technologies. By aggregating scarce resources such as upload bandwidth, decentralized video streaming protocols make it possible to serve a video stream to huge numbers of users while requiring very limited investments from broadcasters. In this paper, we present HEAP, a novel peer-to-peer streaming protocol designed for heterogeneous scenarios. Gossip protocols have already shown their effectiveness in the context of live video streaming. HEAP, *HEterogeneity-Aware gossip Protocol*, goes beyond their applicability and performance by incorporating several novel features. First, HEAP includes a fanout-adaptation scheme that tunes the contribution of nodes to the streaming process based on their bandwidth capabilities. Second, HEAP comprises heuristics that improve reliability, as well as operation in the presence of heterogeneous network latency. We extensively evaluate HEAP on a real deployment over 200 nodes on the Grid5000 platform in a variety of settings, and assess its scalability with up to 100$k$ simulated nodes. Our results show that HEAP significantly improves the quality of streamed videos over standard homogeneous gossip protocols, especially when the stream rate is close to the average available bandwidth.

**Key-words:** Live streaming, gossip, epidemic dissemination, large-scale distributed systems, peer-to-peer, P2P, overlay

* Inria Rennes-Bretagne Atlantique. davide.frey@inria.fr
† EPFL. rachid.guerraoui@epfl.ch
‡ Mediego. anne-marie.kermarrec@mediego.com
§ Banque Cantonale Vaudoise (BCV). maxime.monod@bcv.ch

**RESEARCH CENTRE**
**RENNES – BRETAGNE ATLANTIQUE**

Campus universitaire de Beaulieu
35042 Rennes Cedex

# Diffusion en directe avec du Gossip

**Résumé :** Le streaming vidéo est devenu une killer application pour les technologies pair-à-pair. En agrégeant les ressources rares telles que le debit maximale téléversement, les protocoles de diffusion vidéo décentralisée permettent servir un flux vidéo à un grand nombre d'utilisateurs tout en limitant les couts. Dans cet article, nous présentons HEAP, un nouveau protocole de streaming pair-à-pair conçu pour des réseaux hétérogènes. Les protocoles de gossip ont déjà montré leur efficacité dans le contexte du streaming vidéo en direct. HEAP, em HEterogeneity-Aware Gossip Protocol, va au-delà de protocoles existantes en incorporant plusieurs caractéristiques nouvelles. Premièrement, HEAP adapte la contribution des noeuds en fonction de leurs debit maximal. Deuxièmement, HEAP inclut des heuristiques qui améliorent la fiabilité, en présence de latence de réseau hétérogène. Nous évaluons HEAP sur un déploiement réel sur 200 noeuds sur la plate-forme Grid5000 avec une variété de paramètres, et évaluons son passage à l'échelle avec jusqu'à $100k$ noeuds simulé. Nos résultats montrent que HEAP améliore significativement la qualité des vidéos diffusées par rapport au protocoles standard, surtout lorsque le débit est proche de la bande passante moyenne disponible.

**Mots-clés :** Diffusion de contenu, protocoles épidémiques, systèmes repartis de large échelle, pair-à-pair

# 1 Introduction

Research and industry have proposed a number of solutions for streaming live video over the Internet, while making the most efficient use of scarce resources such as bandwidth. In this respect, the *p2p* communication paradigm provided a crucial improvement with respect to centralized, server-based solutions. Serving a stream to thousands of users from a single origin requires enormous amounts of bandwidth. A *p2p* system, instead, allows the streaming process to combine the bandwidth capabilities of the users that are currently viewing the streamed content. Disseminating a video at a rate 600kbps to 10000 users no longer requires a server farm with a total bandwidth of 6Gbps. Theoretically, it simply requires that the set of peers watching the stream have an average upload bandwidth of at least the stream rate.

Protocols based on the *gossip paradigm*, also known as *epidemic dissemination* [26, 25] have shown to be natural candidates for building such distributed, peer-to-peer, streaming platforms [50, 49, 72]. Epidemic dissemination relies on random communication between participants in a P2P environment. The epidemic starts to spread when a source randomly chooses a set of communication partners, of size *fanout*, and infects them by sharing a message with them. Each of these participants, in turn, randomly picks fanout other partners and infects them. This paradigm has many advantages including *(i)* fast message propagation, *(ii)* probabilistic guarantee that each message reaches all participants, and *(iii)* high resilience to churn and high scalability [77, 75, 44]. As a result, a number of existing research efforts have started from the assumption that gossip is a great way to stream content in a large distributed system [27, 50, 49, 72, 12]

A closer look at gossip, however, reveals a much more complex reality, characterized by significant challenges. First, the same redundancy that provides gossip's natural resiliency to faults can become an unacceptable burden in the presence of large amounts of data as is the case in streaming applications. As a consequence, existing systems [50, 49, 12] use a three-phase variant of gossip in which redundancy is only present in data advertisements, the actual payload being sent only to nodes that explicitly request it. This variant, however, has the negative effect of making dissemination vulnerable to message loss, disconnections, and node failures.

Second, gossip is inherently *load balancing* in that it asks all nodes to contribute equally to the streaming process. This is in sharp contrast with the highly heterogeneous nature of large-scale systems. The presence of nodes with highly diversified available bandwidths, CPU capabilities, and connection delays leads standard gossip protocols to inefficient resource utilization. Highly capable nodes end up being underutilized, while resource-constrained nodes can be overloaded to the point that they provoke system-wide performance degradation. Finally, resource availability is inherently dynamic. Consider a user watching a live stream of the football world-cup final. In the middle of the game, his son starts using another computer on the same network to upload some pictures and make a Skype call. This dramatically changes the amount of bandwidth available for the video stream thereby disrupting performance.

In this paper, we address these challenges by presenting and evaluating *HEAP*, a novel gossip-based streaming protocol, designed to operate in large-scale heterogeneous environments. *HEAP* achieves efficient dissemination by adapting resource consumption to heterogeneous resource availability, while supporting significant levels of message loss, and operation in the presence of other bandwidth consumers.

For the sake of clarity, and pedagogical reasons, we first present and evaluate a basic version of HEAP consisting of its major basic features: fanout adaptation, capability aggregation, and reliability heuristics. Then we introduce and evaluate two additional heuristics that further improve the performance of HEAP.

Overall, this paper introduces a new gossip protocol with the following contributions.

- **Fanout Adaptation**. We introduce a fanout-adaptation scheme that allows each node to contribute proportionally to the ratio between its own upload bandwidth, and the average upload bandwidth of the system (Section 3.1.1). We show how varying the fanout provides a simple way to adapt the operation of gossip to heterogeneous bandwidth capabilities, while preserving its effectiveness.

- **Capability Aggregation**. We introduce a peer-sampling-based aggregation scheme that provides *HEAP* with up-to-date information regarding average capability values (Section 3.1.2). We show that this simple protocol provides results that are almost indistinguishable from those obtained through global knowledge.

- **Reliability Heuristics**. We introduce two reliability heuristics: *Codec* (Section 3.2.1) an erasure coding scheme, and *Claim* (Section 3.2.2), a content-request scheme that leverages gossip duplication to diversify the retransmission sources used in recovering from message loss. We show that neither heuristic alone provides the reliability required by high-quality video streaming. On the other hand, their combination effectively addresses the lack of redundancy associated with three-phase gossip solutions.

- **Optimization Heuristics**. We introduce two additional heuristics: *Push* and *Skip*, which further improve the performance of HEAP in the presence of heterogeneous node delays (Section 6). They achieve this by addressing two other drawbacks of three-phase gossip: bursty behavior, and latency.

- **Extensive Evaluation**. We carry out an extensive performance evaluation to assess the performance of *HEAP* (Section 5) and its optimizations (Section 7). We deploy *HEAP* as a video streaming application on 200 nodes on the Grid5000 [8] platform, and measure the impact on performance of each of its components as well as that of network characteristics, such as the message-loss rate, or the techniques employed for managing upload bandwidth. Furthermore, we evaluate the scalability of our capability-aggregation scheme by simulation with up to $100k$ nodes.

# 2 High-bandwidth Content Dissemination with Gossip

The first protocols for decentralized video streaming emerged from the domain of application-level multicast [20]. The first efforts in this direction employed a single tree-shaped overlay network for stream distribution [38, 10]. But this quickly proved inefficient because only the internal nodes of an overlay tree can contribute to dissemination: most nodes, the leaves, remain passive participants. This brought about a number of solutions based on a multi-tree approach [19]. The source divides the stream into slices and uses a separate dissemination tree for each slice. The internal nodes in one tree become leaves in the others, thereby equalizing node contributions.

Gossip-based streaming takes the idea of multi-tree dissemination to the extreme. Multi-tree dissemination requires splitting the stream into slices, and building one tree per slice by correctly assigning each node to its role in each tree. This becomes tricky in the presence of churn, and ends up requiring expensive tree-maintenance protocols. Gossip-based streaming avoids the problem of building and maintaining trees by dynamically identifying a new dissemination tree for each stream packet.

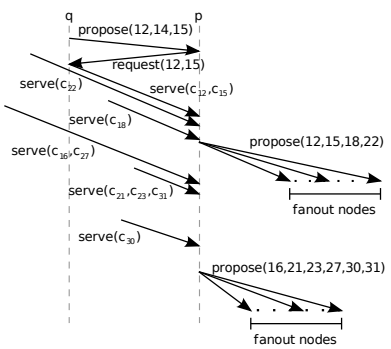
The *gossip paradigm* relies on random communication between participants in a P2P environment. Introduced in [26], gossip takes its inspiration from mathematical models that investigate everyday life phenomena such as rumor mongering and epidemics. An epidemic starts to spread when a source randomly chooses a set of communication partners, of size *fanout*, and infects

them, i.e., it shares data with them. Each of these participants, in turn, randomly picks fanout communication partners and infects them, by sharing the same data. This simple dissemination scheme yields probabilistic guarantees that each message will reach all participants quickly—in a logarithmic number of rounds—with enough redundancy to support high-levels of churn and message loss [44]. This makes gossip an obvious candidate for live streaming in large-scale systems, as documented in the literature [27, 50, 49].

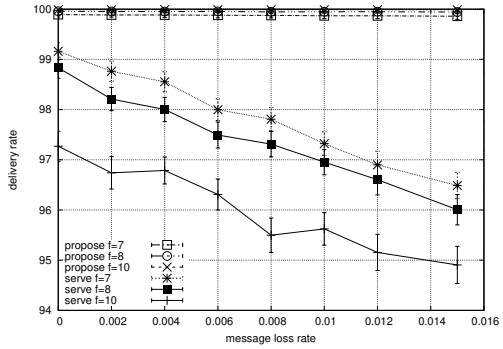
However, a naive application of the above gossip protocol in the context of video streaming would incur significant network overhead. By spreading around a message randomly, gossip creates many duplicates of the same content. While this is a good feature when it comes to guaranteeing dissemination in case of churn, it is a clear disadvantage when spreading large amounts of multimedia content (i.e., time and order-critical data) among participants having limited resources, namely upload bandwidth.

## 2.1 Three-Phase Gossip

To minimize the cost of gossip redundancy while benefiting from the fault tolerance it introduces, we adopt, like several existing protocols [27, 50, 49], the three-phase approach depicted in Figure 1a.




(a) Example




(b) Performance in a constrained-bandwidth scenario

Figure 1: Operation and performance of the three-phase gossip protocols in a realistic scenario (800 kbps bandwidth cap).

- *Propose phase.* Every gossip period, $T_g$, a node that has new stream packets available selects $f$ (fanout) other nodes uniformly at random, and sends them proposal messages advertising the available packets. In Figure 1a, node $p$ proposes packets $12, 15, 18$ and $22$ during the first displayed gossip period, and packets $16, 22, 23, 27, 30$ and $32$ in the subsequent one.

- *Request phase.* As soon as a node receives a proposal for a set of packet identifiers, it determines which of the proposed packets it needs and requests them from the sender. Clearly, the needed packets are those that the node has not yet received. In Figure 1a, node $p$ requests packets 12 and 15 from $q$.

- *Serving phase.* Finally, when the proposing node receives a request message, it provides requesters with the actual stream payload. Specifically, it replies with a message containing

the packets corresponding to the requested identifiers. Nodes only serve packets that they previously proposed. In Figure 1a, node $q$ serves $p$ with packets $c_{12}$ and $c_{15}$.

The use of these three phases makes it possible to create duplicates only on small propose messages while transmitting the actual payload only once to each recipient. The messages sent in the third phase effectively design a custom dissemination tree for each stream packet thereby equalizing the contribution of participating nodes.

## 2.2 Putting Three-Phase Gossip to Work

Like any other gossip-based dissemination protocol, this three-phase solution relies on the ability to select $f$ nodes to communicate with at each round. This raises the question of how to select these $f$ nodes in a decentralized setting. To this end, we adopt a gossip-based membership protocol, also known as Random Peer Sampling (RPS) [41].

An RPS protocol provides each node with a continuously changing sample of the network, the *view*. The view consists of a set of pointers to other nodes. Nodes communicate periodically by exchanging pointers from their views and mixing them like one would do when shuffling a decks of cards. After a sufficient number of iterations, extracting a pointer from a node's view approximates extracting one randomly from the entire network. Random-peer-sampling protocols have been widely studied in the literature [41, 45, 62, 17, 43, 2] and have been applied in a variety of contexts [40, 6, 28, 21, 9, 11]. Yet, their application in video streaming opens important questions. How often should nodes in the RPS shuffle their views? And how large should these views be?

In previous work [33], we showed that the RPS should provide enough pointers so that a node can select $f$ new communication partners at each round. This may be achieved either by increasing the shuffle frequency of the RPS—thereby refreshing the views more often—or by increasing the size of the view—thereby providing a larger pool to choose from. We study this trade-off and provide an answer to the above questions in Section 5.3.

## 2.3 Limitations of Three-Phase Gossip

Even if it makes gossip-based streaming feasible, three-phase gossip still exhibits important limitations. To understand them, we start with an observation. Theory suggests that the fanout, $f$, of nodes should be at least as large as $O(log(N)) + c$, where $N$ is the size of the network and $c$ a constant. Moreover, reliability should intuitively increase when increasing $f$. But, in our previous work [33], we showed that too large values of $f$ cause useless bandwidth consumption and performance degradation. Figure 1b recalls and confirms this result by showing the delivery rates obtained by each of the two sending phases of the three-phase gossip protocol with several fanout values in a 200-node network where each participant has a maximum upload bandwidth of 800kbps, and a message-loss rate between 0% and 1.5%. We provide further details about the corresponding experimental setting in Section 4, but for now, we simply observe that while performance increases when increasing the fanout from 7 to 8, it decreases significantly with a fanout of 10. In the presence of constrained bandwidth, too high fanout values negatively impact performance because heavily requested nodes may easily exceed their bandwidth capabilities.

In addition to confirming that fanout should not be increased arbitrarily, Figure 1b also reveals a difference between the delivery rates of [PROPOSE] and [SERVE] messages. The former achieve high reliability even in the presence of significant message loss. The average delivery rate remains above 99.9% with the worst performing node receiving 99.5% of the [PROPOSE] messages. While this matches the theoretical results about the reliability of gossip, the situation changes dramatically if we analyze the number of actual packets received at the end of phase 3. [SERVE]

messages achieve an average delivery rate between 95% and 99% with no node being able to receive all or even 99% of the stream, regardless of the fanout.

This level of performance appears unacceptable for an application like video streaming [18], and highlights the problems associated with a naive application of three-phase gossip. Fanout values cannot be too high because they would saturate the bandwidth of participating nodes. Similarly too small values limit gossip's ability to disseminate proposal messages to all nodes. Finally, the absence of redundancy in the second and third phases of the protocol drastically increases the impact of message loss. In the remainder of this paper, we present a novel protocol that addresses these constraints and turns three-phase gossip into a top-performance high-quality live streaming solution.

# 3 HEAP

We address the limitations of three-phase gossip described in Section 2.3 by proposing a new protocol, HEAP. HEAP augments three-phase gossip with four novel mechanisms, as depicted in Figure 2: *Fanout Adaptation*, *Capability Aggregation*, *Codec*, and *Claim*. In the following we describe each of these mechanisms by considering their two main goals: *heterogeneity management* (*Fanout Adaptation*, *Capability Aggregation*) and *reliability* (*Codec*, *Claim*).

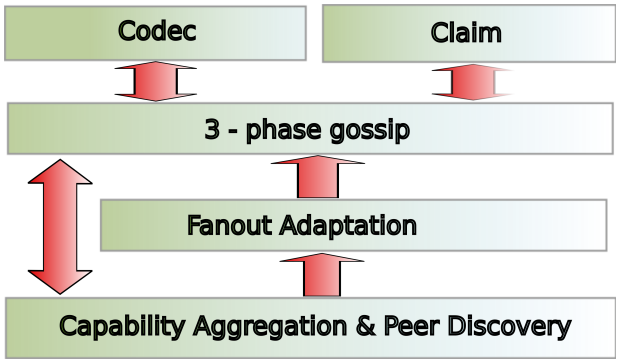



Figure 2: Overall Architecture of *HEAP*

## 3.1 Heterogeneity Management

One of the most appealing characteristics of gossip-based protocols is their inherent load-balancing nature. The random nature of gossip interactions naturally distributes dissemination efforts over all participants. This allows gossip's performance to degrade gracefully in the presence of disconnections and message loss. Nonetheless, this load-balancing aspect quickly turns into a burden when operating in systems composed of highly heterogeneous nodes. If a protocol requires nodes to perform operations that somehow exceed their capabilities, the overall performance of the system quickly degrades.

Decentralized systems can exhibit several forms of heterogeneity, but in the context of video streaming bandwidth heterogeneity has the most effect on performance. Nodes on corporate networks generally have much wider bandwidths than those using cheaper home-based connections. Similarly, the bandwidth available to mobile devices such as Internet-enabled mobile phones depends on available cells and on the number of connected users. If nodes are asked to disseminate more data than their current bandwidth capabilities allow, they will most likely experience

network congestion, increased packet loss, ultimately leading to poor streaming performance throughout the network.

To make things worse, non-corporate connections often exhibit asymmetric characteristics (e.g. ADSL), offering good download, but very constrained upload capabilities. In a streaming application, the download bandwidth of a node will often be much larger than the stream rate. The corresponding upload bandwidth however is often of comparable size if not smaller. This makes the heterogeneity in the upload capability of nodes a very important parameter in the design of a peer-to-peer streaming solution.

### 3.1.1 Fanout Adaptation

HEAP addresses the problem of heterogeneous upload capabilities by tuning the bandwidth consumption of each node so that it matches the corresponding upload capability. The key knob in this adaptation process consists of the *fanout*, the number of partners contacted by a node at each communication round. A node's fanout directly controls the number of [PROPOSE] messages sent during the first phase of the protocol. But it is easy to see that the fanout of nodes also impacts the overall bandwidth consumption.

According to our description in Section 2, each node sends four types of messages: RPS messages to maintain the random overlay structure, [PROPOSE] messages to advertise available stream packets, [REQUEST] messages to request packets that have been proposed, and [SERVE] messages to send the actual stream packets. [SERVE] messages comprise the vast majority of a node's bandwidth consumption because of their larger size and of the high frequency at which they are sent. As a result, we can approximate the amount of upload bandwidth employed by a node, as the number of [SERVE] messages sent per unit of time multiplied by their sizes. Based on this approximation, a node can control its consumed upload bandwidth by controlling the number of [SERVE] messages it sends.

To this end, consider a node, $i$, that sends a [PROPOSE] message for a stream packet, $x$, to $f_i$ other nodes. Let $p_i^x$ be the probability that this proposal be accepted. Node $i$ will send $p_i^x f_i$ [SERVE] messages as a result of this [PROPOSE] message. In a non-congested setting, we can assume $p_i^x$ to be independent of $i$ and of the particular [PROPOSE] message, i.e. $p_i^x = p \, \forall i, x$. This makes the number of [SERVE] messages sent by a node per unit of time directly proportional to the node's fanout. Given two nodes $i$ and $j$ this translates to:

$$f_i = \frac{u_i}{u_j} \cdot f_j. \tag{1}$$

This equation shows that the ratio between the fanouts of two nodes determines the ratio between their average contributions. This suggests a simple heuristic: highly capable nodes should increase their fanouts while less capable ones should decrease it. However, simply setting the fanouts of nodes to arbitrary values that satisfy Equation (1) may lead to undesired consequences. On the one hand, a low average fanout may hamper the ability of 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 [33].

*HEAP* avoids these two extremes by relying on theoretical results. [44] showed that gossip dissemination remains robust and reliable as long as the arithmetic mean—from now on *average*—of all fanouts is of the order of $\ln(n)$, assuming the source has at least a fanout of 1, and regardless of the actual fanout distribution across nodes. This allows us to rewrite Equation (1) by expressing the fanout of a node, $n$, as a function of the desired average fanout, $\overline{f}$, and of the

average bandwidth capability, $\overline{u}$.

$$f_n = \frac{u_n \overline{p}}{\overline{u} p_n} \cdot \overline{f} \tag{2}$$

### 3.1.2 Sample-Based Capability Aggregation

To apply Equation (2), each node, $n$, can compute its upload bandwidth, $u_n$, prior to the start of the streaming process by means of mechanisms such as those in [3, 37, 60]. In addition, each node should obtain information about the average capability of the system. Finally, each node should also be able to select $f_n$ random communication partners at each gossip round in order to distribute stream packets effectively. In *HEAP*, we satisfy these last two requirements with a *sample-based capability aggregation scheme*. In particular, we augment the Random Peer-Sampling (RPS) protocol [41] described in Section 2.2 with information about node capabilities.

An RPS protocol provides each node with a continuously changing random sample of the network. To achieve this, each node maintains a data structure called *view* that maintains references to other nodes. In a standard RPS protocol, each reference consists of the node's IP address and of an associated timestamp specifying when the information in the entry was generated by the associated node. In our context, we augment this information by also including the node's bandwidth capability. Like in a standard RPS protocol, each node periodically selects a random node from its view and exchanges half of its view (plus a fresh entry for itself) with the selected node, which does the same. The two communicating nodes thus end up shuffling their views like two half-decks of cards.

By bundling capability information together with the IP address and timestamp, this augmented RPS protocol allows nodes to estimate the average bandwidth capability of the system. Each node simply keeps track of the capability information associated with the $v_{\mathrm{RPS}}$ entries in its RPS view, and uses the average capability of this subset as an estimator for the average capability of the network. Our experiments reveal that a $v_{\mathrm{RPS}} = 50$ proves to be sufficient to regulate the fanout effectively in networks of up to a few hundred nodes, while a view of $v_{\mathrm{RPS}} = 160$ nodes provides an accurate average-capability estimation in networks of up to 100k nodes.

## 3.2 Reliability

Addressing heterogeneous capabilities is essential for effective data dissemination. Nonetheless, network issues such as message loss may cause drops in performance that can only be addressed by means of additional mechanisms.

Message loss can occur during any of the three phases of the dissemination protocol. Losses in the [PROPOSE] phase can, at least partially, be recovered by the redundancy that is inherent in gossip protocols. However, very severe message-loss rates would still require strong increases in fanout values. Unfortunately, existing work [33] has shown that while theory provides a lower bound of $O(log(n))$ for the fanout, its value cannot be arbitrarily increased in the presence of constrained bandwidth. Our first reliability mechanism, *Codec*, addresses this problem by boosting gossip's efficiency even in the presence of low fanout values.

Our second reliability mechanism, *Claim*, focuses instead on the two subsequent phases of the protocol ([REQUEST] and [SERVE] ). These pose even more problems because they are outside the control of gossip dissemination. While [PROPOSE] messages exhibit natural redundancy, [REQUEST] and [SERVE] messages do not. *Claim* therefore complements three-phase gossip by introducing redundancy in its second and third phases by means of a specialized multi-source retransmission mechanism. We now describe each of these two mechanisms in detail.

### 3.2.1 Codec

*Codec* is a forward error correction (FEC) mechanism that adds redundant encoded packets to the stream so that it can be reconstructed after the loss of a random subset of its packets. A key feature of *Codec* lies in its ability to feed information back into the gossip protocol, thus decreasing the overhead added by FEC. *Codec* increases the efficiency of three-phase gossip in three major ways. First, since each packet is proposed to all nodes with high probability, some nodes do not receive proposals for all packets, even when there is no message loss. FEC allows nodes to recover these missing packets 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 packets 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).

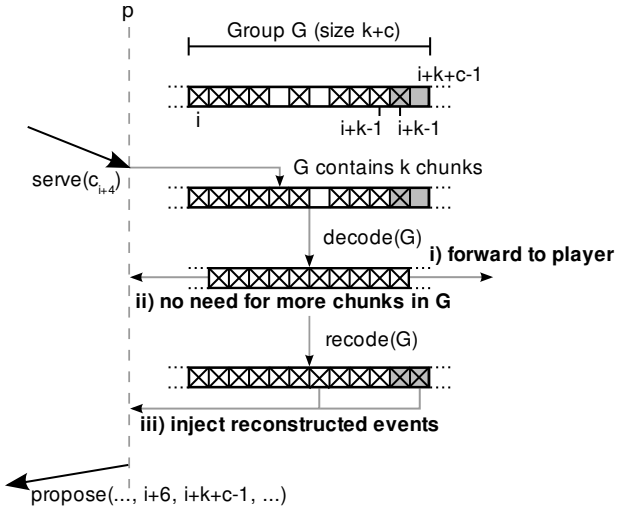



Figure 3: *Codec*: a node $p$ receiving $k$ packets in $G$ decodes the group to reconstruct the $k$ source packets and sends them to the player (step (i)). Node $p$ then signals the protocol not to request any more packets in $G$ (step (ii)). Finally (step (iii)), $p$ re-encodes the $k$ source packets and injects reconstructed packets into the protocol.

**Erasure Coding (FEC)** The source of the stream uses a *block-based* FEC implementation [61] to create, for each group of $k$ source packets, $c$ additional encoded ones. A node receiving at least $k$ random packets from the $k + c$ possible ones can thus decode the $k$ source packets and forward them to the video player (step (i) in Figure 3). If a node receives fewer than $k$ packets, the group is considered *jittered*. Nevertheless, since we are using a systematic FEC (i.e., one that does not alter source packets), a node receiving $j < k$ packets can still deliver the $i \leq j$ source packets that it received. In other words, assuming the $k$ source packets represent a duration, $t$, of an 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** *Codec* employs a FEC technique with negligible CPU costs for coding and decoding [61]. So the cost of FEC in *Codec* essentially consists of network overhead. The source

needs to send $k + c$ packets 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* lies in the observation that a node can stop requesting packets 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 $k + c$ packets (step (ii) in Figure 3). The node will not need to request any more packets in this group from other nodes. The decoding process will instead provide the remaining source packets required to play the stream. This allows the node to save incoming bandwidth and most importantly it allows other nodes to save outgoing bandwidth that they can use for serving useful packets to nodes in need.

A second important characteristic of *Codec* descends from the use of a deterministic FEC encoding. In a deterministic encoding, $k$ source packets produce the exact same $c$ encoded packets independently of the encoding node. This allows *Codec* to introduce an additional optimization (step (iii) in Figure 3). To avoid stopping the dissemination of reconstructed packets (source or encoded packets: packets $i + 6$ and $i + k + c - 1$ in the figure), nodes re-inject decoded packets into the protocol. Thanks to deterministic coding, such injected packets are identical to the ones produced by the source.

### 3.2.2 Claim

The use of *Codec* significantly improves the performance of three-phase gossip. Nonetheless, when used alone, *Codec* does not suffice to provide a satisfactory streaming experience. For this reason, *Claim* introduces a retransmission mechanism that leverages gossip duplication to diversify the retransmission sources of missing information. *Claim* allows nodes to re-request missing content by recontacting the nodes from which they received advertisements for the corresponding packets, leveraging the duplicates created by gossip. Specifically, 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 4.

Nodes can emit up to $r = 5$ re-requests for each packet. To determine how to time their re-requests, nodes keep track of the past response times for delivered packets. We define response time as the time elapsed between the moment a node sends a [REQUEST] and the moment it receives the corresponding [SERVE] message. When requesting a packet, the node schedules the first re-request after a timeout $t_r$, equal to the 99.9th percentile of previous response times. To minimize variability in the percentile value, nodes only apply this rule if they have received at least 500 packets. If they have not, they use a default initial value $t_{r,0}$. Similarly, nodes bound re-request intervals between a minimum of $t_{r,min}$ and a maximum of $t_{r,max}$. To schedule further re-requests, if needed, nodes keep halving the initial timeout until they reach the minimum value, $t_{r,min}$.

Our experiments show that even *Claim* alone cannot guarantee reliable dissemination of all the streaming data to all nodes. On the other hand, its combination with *Codec* is particularly effective and provides all nodes with a clear stream even in tight bandwidth scenarios, and in the presence of crashes.

## 4 Experimental Setting

We evaluated HEAP on 200 nodes, deployed over a cluster of 19 Grid5000 [8] machines. Moreover, we evaluated the scalability of its capability-aggregation scheme by simulation in networks of

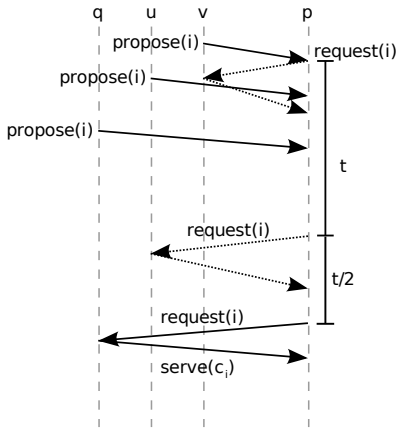



Figure 4: *Claim*: Node $v$ has proposed packet $i$ to node $p$ which requested it. Either the request or the serve was lost and $p$, instead of re-requesting from $v$, now requests $u$, that also proposed packet $i$. If $u$ fails to serve $c_i$, $p$ requests packet $i$ again from another node that proposed $i$. Node $q$ finally serves $p$ with $c_i$.

up to $100k$ nodes. The use of a cluster platform for the evaluation of HEAP 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. The use of simulation, on the other hand, allows us to reach very large network sizes to evaluate the critical component that may affect HEAP's scalability, namely its sample-based capability-aggregation scheme. In the following, we present our default protocol parameters and our evaluation scenario.

## 4.1 Protocol Parameters

The source generates packets of 1397 bytes at a rate of 55 packets per second. This results in an average stream rate of 600 kbps. We set the gossiping period of each node to 200 ms, which leads to an average of 11.26 packet ids per propose message. Where not otherwise specified, we set the average fanout across all nodes to 7. In the standard gossip protocol, each node's fanout is equal to the average fanout. In *HEAP*, nodes derive their fanouts from the average fanout as described in Section 3.1. We configure *Codec* to use a default window size of 100, with 10% of redundancy for a total coded size of 110 packets. At the Random Peer Sampling layer, nodes maintain a view of size $v_{\text{RPS}} = 50$, and exchange gossip messages containing $g_{\text{RPS}} = 25$ entries from their views ($g_{\text{RPS}}$ = gossip size). We vary the parameters of the RPS in Section 5.3. Finally, we consider two configurations for *Claim*: *Fast Claim* and *Slow Claim*. The former uses an initial re-request timeout, $t_{r,0} = 10\,\text{s}$, a minimum re-request timeout, $t_{r,min} = 2\,\text{s}$, and a maximum re-request timeout, $t_{r,max} = 15\,\text{s}$. The latter uses an initial re-request timeout, $t_{r,0} = 500\,\text{ms}$, a minimum re-request timeout, $t_{r,min} = 500\,\text{ms}$, and a maximum re-request timeout, $t_{r,max} = 15\,\text{s}$.

## 4.2 Bandwidth Limitation

The two major approaches to limiting the upload bandwidth available to Internet users consist of the *leaky bucket as a queue* and the *token bucket* (also known as leaky buket as a meter) [71]. The former strictly shapes the outgoing bandwidth of a host so that the instantaneous amount of transmitted data will never exceed the specified limit. The latter, on the other hand, constrains

the average bandwidth to the specified limit but it also allows for short off-limit bursts. In the following, we recall the details of these two approaches.

#### 4.2.1 Leaky Bucket as a Queue

The leaky bucket as a queue algorithm was first proposed by J. Turner [73] with the name of *leaky bucket*. Its name derives from the fact that it can be modelled by a bucket with a hole in the bottom. The bucket is being filled with water (data packets) and the hole causes the water to exit the bucket at a constant rate (the specified bandwidth). If water arrives at a higher rate than it can exit, it accumulates in the bucket. If this happens for too long, the bucket overflows. In the algorithm, the bucket capacity maps to the maximum size of the packet queue. The algorithm drops incoming packets that would cause the queue to exceed its maximum size.

#### 4.2.2 Token Bucket

The token bucket algorithm takes a different approach and constrains the average bandwidth while allowing some bursty traffic to pass as is. Bursts that are too long are however dropped or truncated. Basically, the algorithm maintains a counter variable that measures the currently allowed burst (expressed as a number of bits). At every time interval (e.g. every millisecond), the algorithm, increments the value of this variable by a rate corresponding to the allowed bandwidth. For example, for 600 kbit/s, the algorithm will increment the counter by 600 bits every millisecond. If the counter is already at its maximum value (which corresponds to the size of the maximum allowed burst), then the value is not incremented. When sending a packet, the algorithm first checks if the counter is at least as large as the size of the packet. If so, it decrements the counter by the size of the packet and sends it. Otherwise it leaves the counter unchanged and drops the packet.

### 4.3 Bandwidth Scenarios

In all our experiments, we use one of the two bandwidth-limiting algorithms described above, and give the source enough bandwidth to serve 7 nodes in parallel. Using the same algorithm, we also limit the upload bandwidth of each of the other nodes according to one of four scenarios. Our first scenario consists of a homogeneous setting in which in which all nodes have the same upload capability. We denote the scenario by homo-X, where X represents the upload bandwidth in kbps. For example, homo-691 denotes a scenario where all nodes have 691 kbps of upload bandwidth. The three remaining scenarios reflect heterogeneous bandwidth distributions inspired by those in [81]: ref-724, ref-691, and ms-691.

Table 1 summarizes the composition of the heterogeneous scenarios. The *capability supply ratio* (CSR, as defined in [81]) represents the ratio of the average upload bandwidth to the stream rate. In all the considered settings, the average upload bandwidth suffices to sustain the stream rate. Yet, the lower the capability ratio, the closer we stand to the limit represented by the stream rate. In addition, in the heterogeneous scenarios, some of the nodes' upload bandwidths are well below the limit. Each of the heterogeneous settings comprises three classes of nodes and its skewness depends on the percentage of nodes in each class. In the most skewed scenario we consider (ms691), most nodes are in the *poorest* category and only 15% of nodes have an upload capability larger than the stream rate.

In all the considered scenarios, bandwidth limiters introduce delays in the dissemination of messages. However, data sent over the Internet is also subject to bandwidth-independent delays (e.g. propagation delays). To model this, we also associate each message, regardless of its type, with an additional random delay, uniformly distributed between 50ms and 250ms.

| | | | 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 1: The reference distributions ref-691 and ref-724, and the more skewed distribution ms-691.

### 4.4 Metrics

We evaluate HEAP according to two metrics. First, we define the *stream lag* as the difference between the time the stream was published by the source and the time it is actually delivered to the player on the nodes. Second, we define the *stream quality* as the percentage of the stream that is viewable without jitter. We consider a FEC-encoded window to be *jittered* as soon as it does not contain enough packets (i.e., at least 101 with our default parameters) to be fully decoded. A X%-jittered stream therefore means that X% of all the windows are jittered. Note that, as explained in Section 3.2.1, a jittered window is not entirely lost. Because *Codec* employs systematic coding, a node may still receive 100 out of the 101 original stream packets, resulting in a 99% delivery ratio in a given window.

## 5 Performance Evaluation

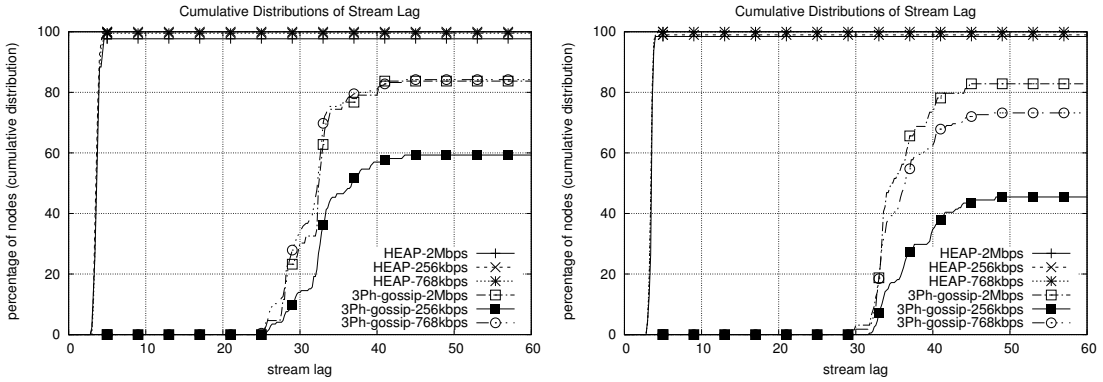
We start our evaluation by examining the performance of HEAP in the presence of heterogeneous bandwidth constraints, and by comparing it with that of the standard three-phase gossip protocol described in Section 2.1. Then, we analyze the reasons for its good performance by examining different scenarios, different bandwidth limiters, and different configurations of the RPS protocol. Finally, we evaluate the impact of HEAP on external applications. To ensure a fair comparison, we augment the standard gossip protocol with the same error correction and retransmission mechanisms as HEAP, namely Codec and Claim.

### 5.1 HEAP Test Drive: Heterogeneous Bandwidth Constraints

Figures 5 and 6 compare the stream lags required by HEAP and Standard Gossip to obtain a non-jittered stream in the three heterogeneous scenarios in Table 1. In all cases, HEAP drastically reduces the stream lag for all capability classes. Moreover, as shown in Figure 6 the positive effect of HEAP significantly increases with the skewness of the distribution.

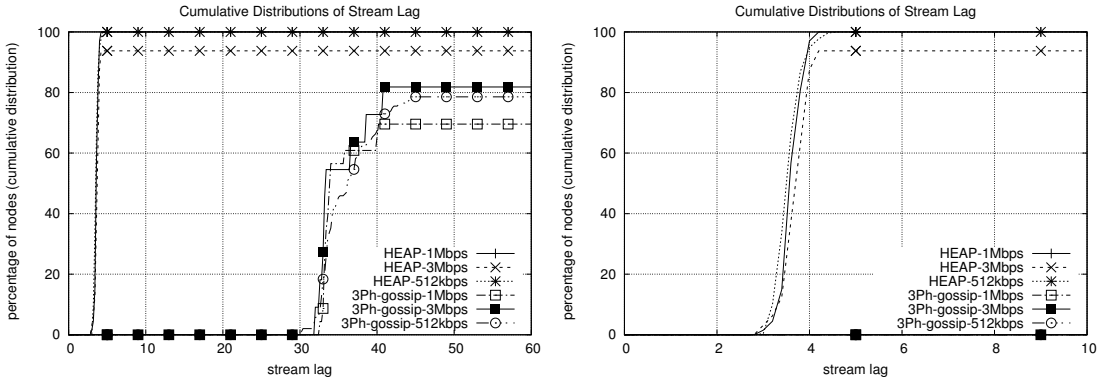
All the plots depict the cumulative distribution of the nodes that manage to view a jitter-free stream as a function of the stream lag, for each capability class and for each of the two protocols. In all cases, HEAP provides a clear stream to almost all nodes with a lag of less than $5s$, while standard gossip induces much higher lags (an order of magnitude higher) to reach much fewer nodes.

Figure 5a presents the results for ref-691. Here, HEAP provides a perfectly clear stream to 100% of the low-capability nodes within $4.4s$, to 99.5% of the mid-capability ones within $5s$, and to 97.7% of the high-capability ones within $4.6s$. Standard gossip, on the other hand, cannot provide a clear stream to any node within less than $24s$ and it only reaches a much lower fraction




(a) ref-691 (b) ref-724

Figure 5: Cumulative distribution of stream lag in the ref-691 and ref-724 scenarios for HEAP and standard three-phase gossip.




(a) ms-691 (b) ms-691 close-up

Figure 6: Cumulative distribution of stream lag in the ms-691 scenario for HEAP and standard three-phase gossip.

of nodes (60% to 84%) with delays that exceed 40$s$, as shown in Table 2. The results for ref-724 (Figure 5b) and ms-691 (Figure 6) have a similar flavor with two minor differences. First, standard gossip turns out to be even less efficient in these two scenarios: even in ref-724, which is the least constrained. Second, the performance of high-bandwidth nodes in HEAP decreases slightly for ms-691, but it remains at very high levels: 97.7% of the nodes receive a perfect stream within 4.6$s$.

Table 3 completes these results by adding the percentage of nodes that receive 99.9% of the stream and the corresponding maximum lag. Even with this other metric, HEAP consistently exhibits higher node percentages and shorter lags than standard gossip. In the most constrained ms-691 scenario, HEAP provides 99.9% of the stream to 100% of the nodes within 3.2$s$, while standard gossip requires 25$s$ to provide the same percentage of stream to the first few percent of the nodes (not shown in the table), and reaches close to 100% of the nodes in over 30$s$.

| | HEAP | | | Standard Gossip | | |
|---|---|---|---|---|---|---|
| bw | low | mid | high | low | mid | high |
| ref-691 | 100% / 4.4$s$ | 99.5% / 5$s$ | 97.7% / 4.6$s$ | 59.3% /43.6$s$ | 84%/ 43.8$s$ | 83.7%/ 40.6$s$ |
| ms-691 | 100% / 4.6$s$ | 100% / 4.2$s$ | 93.8% / 4.2$s$ | 78.5% /44.4$s$ | 69.6%/ 40.2$s$ | 81.8%/ 40.8$s$ |
| ref-724 | 99% / 4.2$s$ | 100% / 4.4$s$ | 98.4% / 4.0$s$ | 45.4% /48.2$s$ | 73.2%/ 48.0$s$ | 82.8%/ 44.8$s$ |

Table 2: Percentage of nodes that receive 100% of the stream and corresponding maximum lag.

| | HEAP | | | Standard Gossip | | |
|---|---|---|---|---|---|---|
| bw | low | mid | high | low | mid | high |
| ref-691 | 100% / 3.4$s$ | 100% / 3.4$s$ | 100% / 3.4$s$ | 94.1% /40.6$s$ | 99.5%/ 35.6$s$ | 100%/ 33.0$s$ |
| ms-691 | 100% / 3.2$s$ | 100% / 2.8$s$ | 100% / 2.8$s$ | 98.5% /39.6$s$ | 100%/ 35.4$s$ | 100%/ 33.0$s$ |
| ref-724 | 100% / 2.8$s$ | 100% / 3.0$s$ | 100% / 3.0$s$ | 85.9% /47.6$s$ | 99.4%/ 42.2$s$ | 100%/ 35.4$s$ |

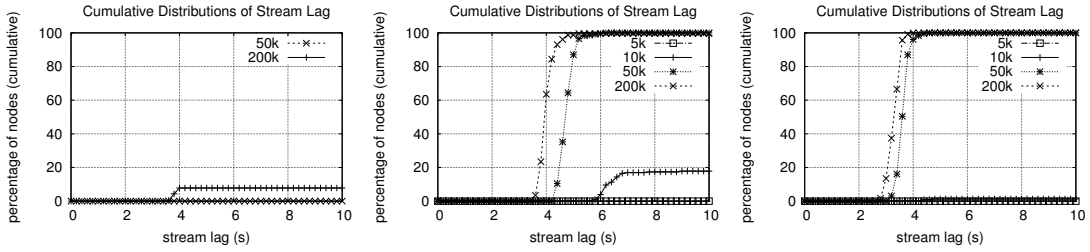
Table 3: Percentage of nodes that receive 99.9% of the stream and corresponding maximum lag.

To summarize, HEAP consistently provides a clear stream to a larger number of nodes than standard gossip and with a lag that is one order of magnitude lower. Moreover, we observe that the lag induced by HEAP is in fact indistinguishable from the lag induced in a homogeneous bandwidth setting.

## 5.2 Impact of Bandwidth-Limiting Techniques

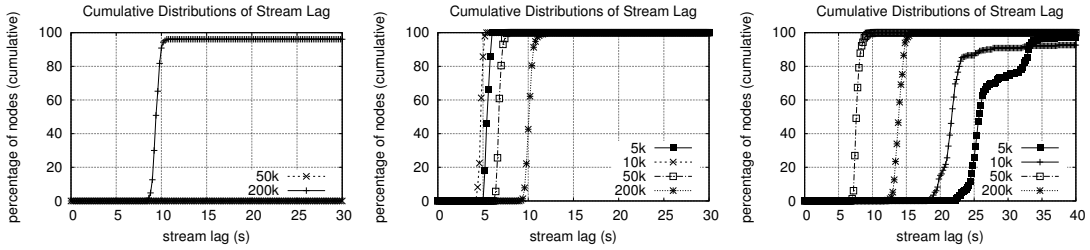
As discussed in Section 4.2, there exist two major families of algorithms to constrain the upload bandwidth on a network: the leaky bucket as a queue (referred to as *leaky bucket* in the following) and the token bucket. Up to now, we have shown results for HEAP and standard gossip running on top of a token-bucket limiter. In this section, we explore the impact of this choice and discuss how each bandwidth limiter affects gossip-based dissemination.

To achieve this, we consider a homogeneous scenario in which all nodes have the same upload bandwidth of 691kbps, and we test three protocol variants: No-Claim, Slow-Claim, and Fast-Claim. No-Claim consists of a standard three-phase protocol augmented with Codec, but without any retransmission mechanism. Slow-Claim and Fast-Claim are two variants of HEAP and both consist of No-Claim augmented with the Claim retransmission mechanism. However, Slow-Claim uses a slower setting with a minimum re-request timeout of 2$s$ and an initial timeout of 10$s$, while Fast-Claim uses a minimum timeout of 500$ms$ and an initial timeout of 500$ms$. Slow-Claim results in a fairly conservative retransmission policy, while Fast-Claim leads to a much more aggressive behavior. In addition to these three variants, we also tested the corresponding variants without Codec. We do not show any plot because, without Codec, none of the nodes managed to view a completely clear stream except for a handful with an unlimited leaky bucket. Finally, we observe that, in a homogeneous setting, HEAP is equivalent to standard gossip augmented with Codec and Claim. So everything we say about the former, also applies to the latter.




(a) No Claim (b) Slow Claim (c) Fast Claim

Figure 7: Stream Lag with various token-bucket burst sizes with no retransmission (left) and with slow and fast retransmission (center and right).



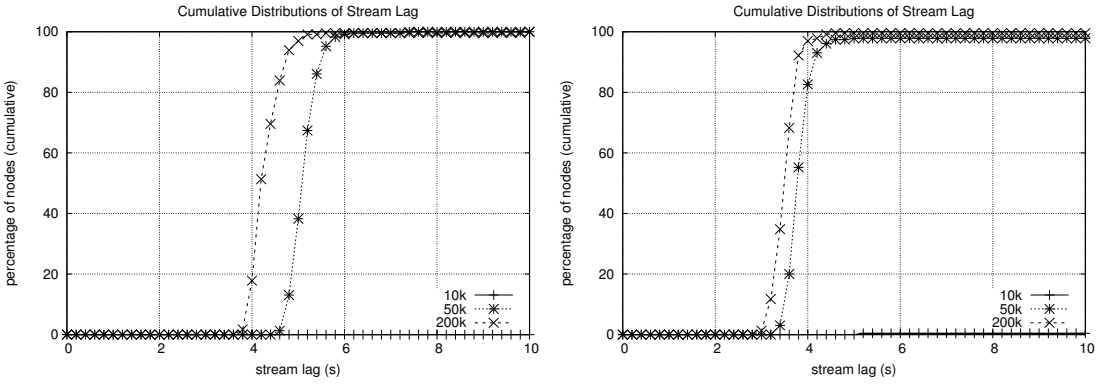

(a) No Claim (b) Slow Claim (c) Fast Claim

Figure 8: Stream Lag with various leaky-bucket queue sizes with no retransmission (left) and with slow and fast retransmission (center and right).

### 5.2.1 Bandwidth Limitation on Ideal Network Channels

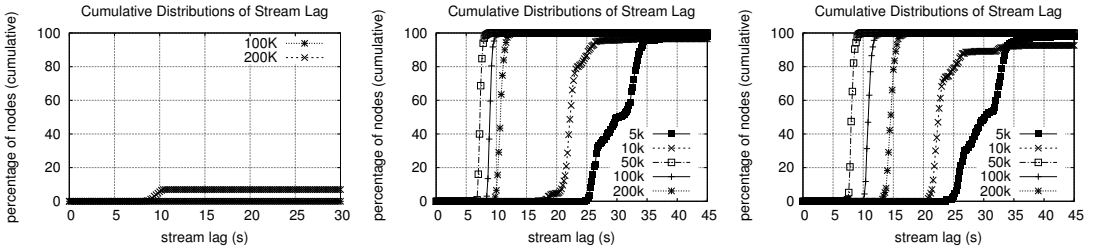
To isolate the effect of limited bandwidth, we start by considering a scenario without any message loss. Figure 7 presents the results for the token-bucket-limiter. Figure 7a highlights the importance of Claim when operating in this setting. Without retransmission, HEAP can only delivery a clear stream to a mere 8% of the nodes, and with a lag of almost $10s$ and only with a token-bucket size of 200KB, a much poorer result than what we presented in Section 5.1. Figure 7b presents instead the results when using Slow-Claim. In this case, HEAP can provide a clear stream to all nodes as long as the token bucket size is at least $50KB$. With a token-bucket size of $200KB$, our default parameter, the average stream lag settles at around $4s$. With Fast-Claim (Figure 7c) we have a similar situation, but the average delay with a $200KB$ bucket decreases from $4s$ to 3.5s.

Figure 8 shows instead the results obtained with a leaky-bucket. Interestingly, a leaky bucket of $200KB$ allows HEAP to provide a full stream to almost all nodes even with the No-Claim variant. The queuing behavior of the leaky bucket strongly reduces the amount of message loss and allows nodes to disseminate the stream reliably even if with a higher delay than on Figure 7. Figure 8a seems to suggest that larger queues are beneficial for the performance of the protocol, but Figures 8b and 8c provide opposing evidence. Slow-Claim allows HEAP to deliver a full stream to all nodes with all bucket sizes. Yet the bucket sizes that yield the lowest delays turn out to be the smallest: 10KB and 5KB. Similarly, Fast-Claim provides a full stream to all nodes for 2 out of 5 bucket sizes, but here the best-scoring size is 50KB, with 200KB being second,




(a) Slow Claim (b) Fast Claim

Figure 9: Stream Lag with various token-bucket burst sizes with 1.5% of message loss as well as slow and fast retransmission (respectively left and right).




(a) No Claim (b) Slow Claim (c) Fast Claim

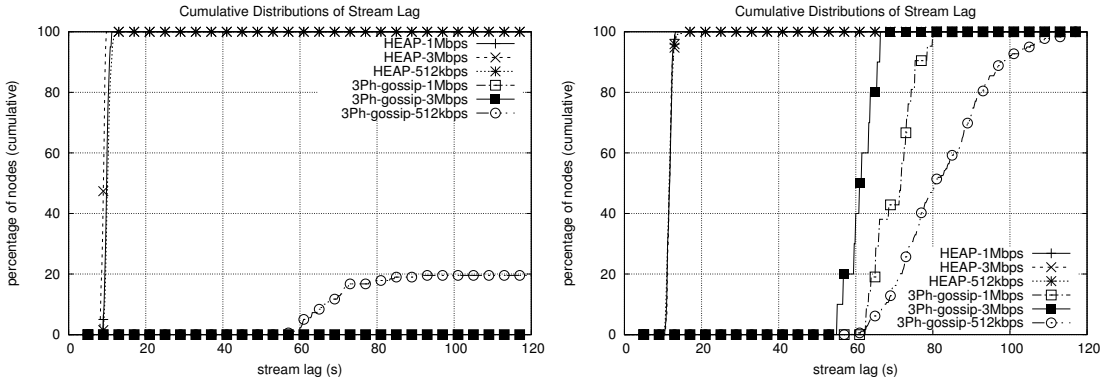
Figure 10: Stream Lag with various leaky-bucket queue sizes with 1.5% of message loss and no retransmission (left) as well as slow and fast retransmission (center and right).

and with 5KB scoring the worst. Moreover, while Fast-Claim performs better than Slow-Claim on a token-bucket, it performs much worse in Figure 8. Fast-Claim's aggressive retransmission behavior causes congestion in the leaky bucket, which translates into additional queuing delays. Overall, when considering Figure 7 and Figure 8, Fast-Claim maximizes performance on a token bucket, while Slow-Claim maximizes it on a leaky bucket.

### 5.2.2 Bandwidth Limitation with Additional Message Loss

Figure 9 examines the same setting as Figure 7, with the addition of a random 1.5% of message loss. We do not show the plot for No-Claim, because this variant cannot provide a clear stream to any node in the presence of both a token-bucket limiter and message loss. Figures 9a and 9b, thus, show respectively the results for Slow-Claim and Fast-Claim. Results resemble those in Figure 7, except that here a token-bucket of $10KB$ makes it impossible to provide a clear stream to any node. The average delays also increase a little, particularly in the case of Slow-Claim. Fast-Claim, therefore remains the best option in the case of a token-bucket limiter.

Figure 10 shows results with message loss and a leaky-bucket limiter. Here, No-Claim can no longer provide satisfactory performance, and Slow-Claim almost loses the performance advantage




(a) ms691 200KB Leaky Bucket

(b) ms691 Unlimited Leaky Bucket

Figure 11: Cumulative distribution of stream lag in the ms691 scenario for HEAP and standard gossip with a leaky bucket of 200KB (left) and with an unlimited leaky bucket (right).

over Fast-Claim highlighted in Figure 8. The delays for bucket sizes of 50KB are almost identical for the two variants. Yet, with a bucket size of 200KB, Slow-Claim remains almost 4s faster than Fast-Claim.

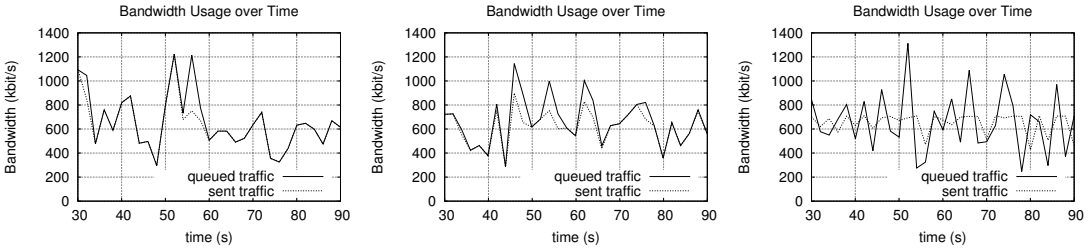
### 5.2.3 Bandwidth Limitation and Heterogeneous Bandwidth

We conclude this analysis of bandwidth limitation, by showing the performance of HEAP and standard gossip in a heterogeneous setting with a leaky-bucket bandwidth limiter. We consider the most skewed scenario discussed in Section 5.1, and rerun the same experiments but with a leaky bucket of 200KB, and with an unlimited leaky bucket. Figure 11a displays the results for the 200KB bucket. As expected, HEAP performs slightly worse than with a token bucket. But it still performs much better than standard gossip. In the ref-724 scenario, HEAP provides all nodes with 100% of the stream within $8s$, while standard gossip requires an average of $25s$. The curves for HEAP also show much greater differences between capability classes than with a token bucket. This is consistent with the fact that the leaky bucket's queue lengthens all delays and thus amplifies the differences between the groups of nodes.

The leaky bucket, however, does not amplify the difference between HEAP and standard gossip. Rather, standard gossip seems to perform even better than with a token bucket. Standard gossip's poor performance results mostly from its inability to prevent congestion at low capability nodes. In the case of a token bucket, congestion results in lost messages, retransmission operations, and more lost messages. In the case of a leaky bucket, congestion results first in delays, and thus it leads to fewer lost messages. Figure 11a clearly shows the impact of this phenomenon on standard gossip through the long step-like pattern. Figure 11b shows the results of the same experiment but with an unlimited leaky bucket. Here there is no step-like behavior and standard gossip eventually manages to deliver a clear stream to all nodes, but HEAP still conserves a significant advantage in terms of stream lag.

### 5.2.4 Understanding Bandwidth Usage

To understand the reasons for the interaction between the protocol variants and the different bandwidth limiters, we now analyze how nodes exploit their bandwidth capabilities. We start



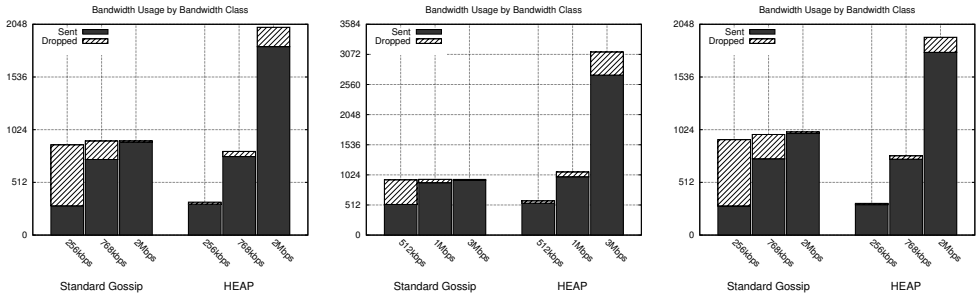

(a) Token Bucket 200k (b) Token Bucket 50k (c) Leaky Bucket 200k

Figure 12: Bandwidth usage over time with no message loss and slow retransmission with two token-bucket burst sizes, and with a leaky bucket.

by considering bandwidth consumption over time. Figure 12 plots the number of bits per second submitted to the bandwidth limiter, as well as the number of bits that get effectively sent. We obtained the plots by counting the amount of data queued and sent in every $2s$ interval. Figure 12a, and Figure 12b show the data for nodes running on token buckets of respectively 200KB and 50KB, while Figure 12c considers a node running on a 200KB leaky bucket.

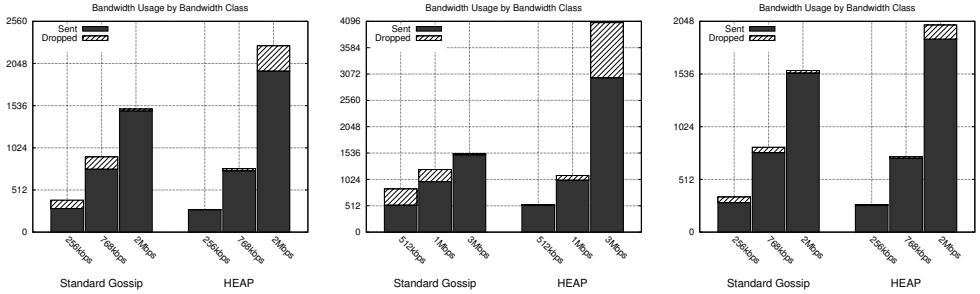
All three figures highlight the spiky nature of gossip traffic. Nodes appear to follow a pattern where short intervals of low-bandwidth usage interleave with high-bandwidth spikes. Such spikes result from the random nature of gossip dissemination: at times nodes receive a large number of request messages, and thus end up serving many other nodes. At other times, they receive fewer requests and therefore end up serving fewer nodes. The leaky-bucket figure shows a fairly regular pattern. Queued bandwidth appears to oscillate around a stable average, which correspond to the almost constant sent bandwidth. The token-bucket ones, on the other hand, show a more irregular pattern because the token bucket can follow occasional bursts of gossip, but only if they are not too close to one another.

Figures 13 and 14 further highlight the differences between the two bandwidth limiters by depicting the breakdown of bandwidth consumption among the three classes of nodes. Each plot considers one of the three heterogeneous scenarios introduced in Section 4.3, respectively with a 200KB token-bucket (Figure 13) and a 200KB leaky-bucket bandwidth limiter (Figure 14). The total height of each vertical bar indicates the average bandwidth that nodes in the corresponding capability class attempt to use. The black part shows the portion of this bandwidth that results in successfully sent messages, while the striped part corresponds to messages that are dropped by the bandwidth limiter. For example, the second bar for standard gossip in Figure 13a shows that nodes with an upload capability of 768 kbps attempt to send 913 kbps but manage to send only 734 kbps (black portion), the remaining 179 kbps (striped portion) being dropped.




(a) ref-691 (b) ms-691 (c) ref-724

Figure 13: Bandwidth consumption by capability class with a 200KB token bucket.



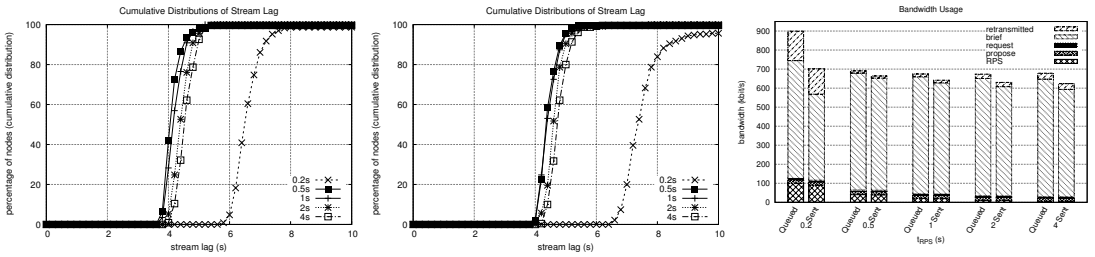

(a) ref-691 (b) ms-691 (c) ref-724

Figure 14: Bandwidth consumption by capability class with a 200KB leaky bucket.

Figure 13 clearly shows the difference in bandwidth usage between HEAP and Standard Gossip. HEAP nodes attempt to use an amount of bandwidth that is very close to their actual capabilities. Standard-Gossip nodes, on the other hand, attempt to send approximately 1 Mbps of data regardless of their capability class. The black portions of the vertical bars (successfully sent data) show that Standard Gossip is completely unable to exploit high capability nodes and therefore ends up saturating lower capability ones. HEAP, on the other hand, effectively exploits all the available bandwidth of all node classes thereby limiting the number messages dropped by the token bucket.

Figure 14 shows a seemingly very different behavior in the case of a leaky-bucket limiter with a 200KB queue. Here, even Standard Gossip seems to exhibit some form of natural self-adaptation. The total height of each vertical bar appears to follow, albeit not so strictly as in the case of HEAP, the corresponding upload capability. However, this seemingly adaptive behavior simply results from the delays introduced by the leaky-bucket limiter. The leaky bucket queues all messages in chronological order regardless of their type. So the the [PROPOSE] messages sent by congested nodes experience higher delays than those sent by non-congested ones; and, in general, low-bandwidth nodes will be more congested than high-bandwidth ones. Since nodes respond to the first [PROPOSE] message they receive, queuing delays cause the proposals of low-bandwidth nodes to be accepted less often, ultimately causing low-bandwidth nodes to send fewer [SERVE] messages.

This pseudo self adaptation does not improve performance as much as the adaptive fanout




(a) homo-691 (b) ms-691 (c) bandwidth breakdown in homo-691

Figure 15: Stream Lag with the random peer sampling running at various frequencies—different values of $t_{\mathrm{RPS}}$—on both homogeneous and heterogeneous bandwidth (ms-691) with a token-bucket configuration, and bandwidth usage in the homogeneous case.

of HEAP. But it replaces the message loss of the token bucket with delays in the delivery of stream packets. This further explains the difference between the results obtained with the two bandwidth limiters: for example between Figure 9 and Figure 10. Standard gossip cannot provide all nodes with the entire stream with a token bucket, while it can but with huge delays with a leaky bucket. HEAP, on the other hand, provides the entire stream to all nodes with either bandwidth limiter and with much lower delays than Standard Gossip.

## 5.3 Sensitivity Analysis

A HEAP node continuously needs to advertise stream packets by sending a propose message to $f$ other nodes every 200ms. For the protocol to work correctly, the node must choose these $f$ other nodes as randomly as possible. As discussed in Section 3.1.2, HEAP achieves this by relying on an augmented random-peer-sampling protocol (RPS). In this section, we evaluate how two important parameters of this protocol impact the performance of HEAP.

### 5.3.1 RPS Gossip Interval

The gossip interval determines the frequency at which the nodes in the RPS view get refreshed. The higher this frequency, the lower the correlation between the two sets of $f$ nodes selected at two subsequent rounds. To minimize correlation, nodes should ideally refresh their views multiple times between two dissemination actions [41]. However, this turns out to be almost impossible with a dissemination action every 200ms. We therefore seek to identify the best tradeoff between the randomness of peer selection and the need to maintain a reasonable gossip frequency.

To achieve this, Figure 15 plots the stream-lag distribution obtained by HEAP in homo-691 and ms-691 with the RPS running at various frequencies. Figure 15a shows the results for homo-691. The best performance corresponds to RPS gossip intervals ($T_{\mathrm{RPS}}$) of 500ms and 1s. Intervals of 2s and 4s perform slightly worse, while an interval of 200ms exhibits the worst performance by adding approximately 2s to the stream lag of the other configurations.

The data in Figure 15b shows similar results in the case of ms-691. In this case, the difference between 500ms and 1s becomes negligible and the negative impact of too fast an RPS becomes even stronger. The curve for 200ms shows a lag increase of almost 3s with respect to the 1s curve.

To explain these results, we analyze how the various messages employed by the protocol

contribute to bandwidth consumption. Figure 15c shows a stacked histogram with the number of bits/s that are queued into the token-bucket and those that are actually sent for each of the RPS configurations in Figure 15a. The two bars for $T_{\mathrm{RPS}} = 200ms$ show that the poor performance associated with this gossip frequency results from the very high amount of bandwidth consumed by the RPS protocol in this setting ($100kbps$). This huge bandwidth consumption causes the protocol to attempt to send a lot more than the allotted $691kbps$. This causes a significant amount of message loss, which, in turn, triggers a significant number of retransmission operations.

With a $T_{\mathrm{RPS}}$ of 500ms, the amount of queued bandwidth is only sightly higher than the allotted maximum and the number of retransmission operations appears a lot more reasonable. Yet, the RPS protocol still consumes as much as $50kbps$, for only a marginal improvement with respect to $T_{\mathrm{RPS}} = 1s$.

The bars for frequencies below one gossip per second highlight what might seem like a contradiction. The average sent bandwidth is lower than the average queued bandwidth—some messages are being dropped—even if the latter is lower than the bandwidth limit. The reason for this behavior lies in the bursty nature of gossip traffic. The average bandwidth reflects, in fact, the congestion bursts resulting from the random nature of gossip dissemination, as discussed in Section 5.2.4. In some rounds a node may receive a large number of requests for stream packets and thus attempt to send more than the maximum bandwidth. In others, it may receive only a few, if any, requests and will therefore send much less than the allotted maximum.

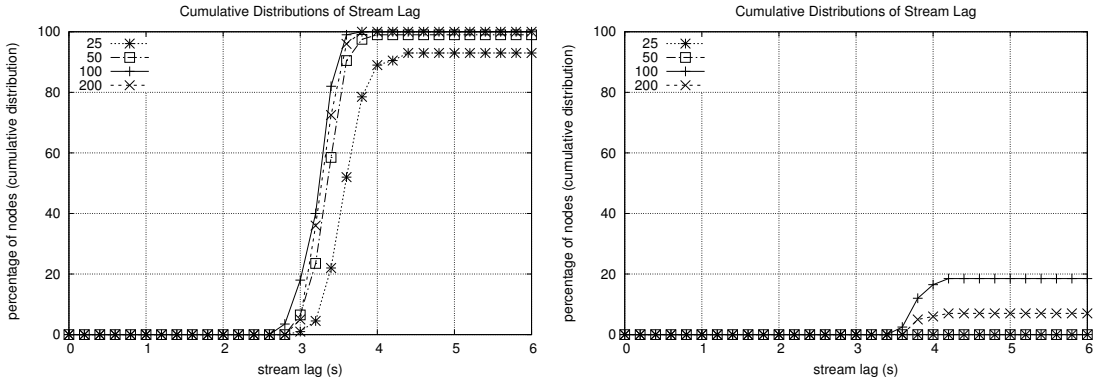
We also observe that starting from $T_{\mathrm{RPS}} = 1s$, the number of retransmission actions slightly increases with $T_{\mathrm{RPS}}$. As expected, slower gossip frequencies cause higher correlation between the nodes chosen at subsequent rounds. This exacerbates the congestion bursts described above resulting in additional message loss and thus in a higher number of retransmission actions.

### 5.3.2 View Size

A second way to minimize the correlation between two subsequent samples of $f$ nodes consists in increasing the size of the population from which these nodes are extracted. In the case of the RPS protocol, we can achieve this by increasing its view size. To evaluate the effect of this parameter, Figure 16a shows the impact of different view sizes on streaming performance, while maintaining a gossip interval of $T_{\mathrm{RPS}} = 1s$.

The plot shows that HEAP achieves good performance with all the considered view sizes. Yet, it achieves the best performance with view sizes of at least 50 nodes. With a fanout of 7, and a $T_{\mathrm{RPS}}$ of 1s, nodes choose 35 nodes per second. A view size of 25 therefore necessarily results in highly correlated views, while a view size of 50 provides sufficient randomness. Larger view sizes provide only minimal improvements, which justifies the choice of a view size of 50 in the rest of this paper.

To appreciate the impact of larger view sizes, we also consider a variant of HEAP in which we disable Claim, the retransmission mechanism described in Section 3.2.2. With a view size of 50, this variant is unable to provide a clear stream to any node. Yet, a larger view size of 100 nodes manages to provide a clear stream to almost 20 nodes with less than $4s$ of lag. Albeit not good enough to get rid of Claim, this result shows that the more uniform peer selection provided by a larger view boosts the performance of HEAP. Yet, setting the view size to 200 decreases, rather than increases, performance. The improvement brought about by larger view sizes is in fact limited by the associated bandwidth consumption—nodes exchange subviews that contain half-a-view-size nodes.




(a) homo-691 (b) homo-691 - no Claim

Figure 16: Stream Lag with various view sizes ($v_{\mathrm{RPS}}$) for the random peer sampling protocol for standard HEAP and for a variant without retransmission.

## 5.4 Scalability

We have so far shown how HEAP provides significant performance improvements in a variety of settings in a network of 200 nodes. We now examine its ability to scale to very large networks. To this end, we first observe that the local nature of the three-phase dissemination protocol naturally scales to arbitrary numbers of nodes, as demonstrated by existing real-world deployments [48]. Similarly, existing research on RPS protocols has shown their effectiveness in very large networks with hundreds of thousands of nodes [41]. As a result, the only doubts about scalability may arise from the use of the RPS for the estimation of the average bandwidth of the system.

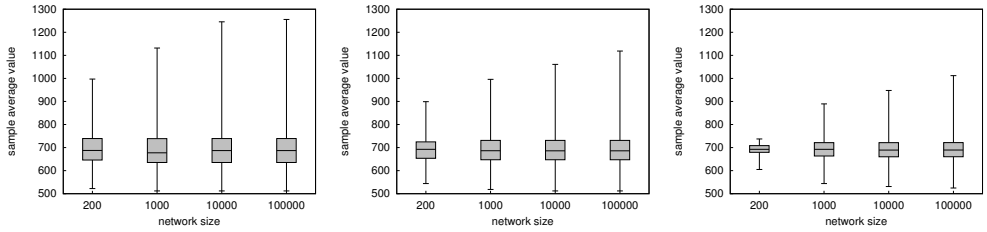
More precisely, the method we use to estimate the average bandwidth of the network is known in statistics as the sample mean, and a well known result states that the ratio between the variance of the initial distribution and that of the sample mean is equal to the size of the sample. So with an RPS view of 50 a node should get an estimation of the average with a variance that is 50 times smaller than that of the bandwidth distribution across the network. However, this theoretical result holds in the presence of a purely random sample, while the RPS only approximates ideal randomness.

To evaluate whether this may affect the accuracy of the bandwidth estimation, we simulated the RPS-based averaging component of HEAP in networks of sizes ranging from 200 to 100000 in each of the three heterogeneous scenarios. We configured the RPS to use a fixed view size regardless of network size, and considered three configurations with $v_{\mathrm{RPS}}$ values of 50, 80, and 160. A configuration with a network size of 200 and $v_{\mathrm{RPS}} = 50$ corresponds to that of our Grid5000 experiments. For space reasons, we only show the results for ms691, but those for the other scenarios are equivalent.

Table 4 shows the ratios between the variance of the sample average and the variance of the bandwidth distribution in the entire network. For very large networks of 10000 nodes and above, the ratio between the variance of the bandwidth distribution and that of the estimated average indeed follows theoretical predictions despite the imperfect randomness of the RPS. But as expected, for smaller networks, the ratio is even higher as the RPS view constitutes a larger proportion of the network. While this suggests that a view of 50 cannot give the same performance in a network of $100k$ nodes as in one of 200, the table also shows that a view of 80 yields approximately the same variance ratio in large networks as that of a view of 50 with

| $v_{\text{RPS}}$ | Variance ratios by network size | | | |
|---|---|---|---|---|
| | 200 | 1k | 10k | 100k |
| 50 | 78.52 | 58.75 | 51.23 | 50.10 |
| 80 | 161.28 | 95.44 | 82.38 | 80.26 |
| 160 | 820.92 | 208.58 | 167.08 | 160.25 |

Table 4: Variance Ratios in the ms-691 scenario.




(a) $v_{\text{RPS}} = 50$ (b) $v_{\text{RPS}} = 80$ (c) $v_{\text{RPS}} = 160$

Figure 17: Whisker plots for sample average obtained from the RPS view with increasing network sizes and RPS views.

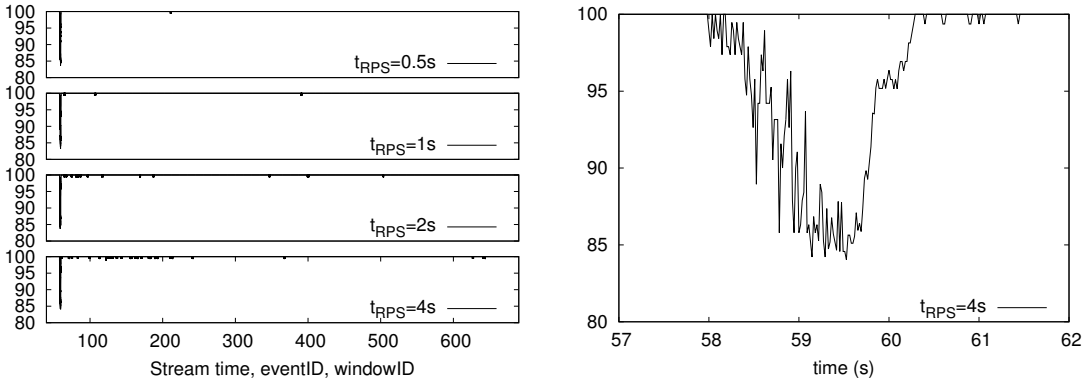
200 nodes. This suggest that a relatively small increase in view size can offset of an increase in network size of three orders of magnitude.

To evaluate this hypothesis, Figure 17 plots the distribution of sample average values in each configuration as box-and-whisker plots. For each, configuration, the box extends from the first to the third quartiles, while the whiskers show the minimum and the maximum values for the sample average. The figure shows that while the distribution spreads out when increasing the size of the network, a 3-fold increase in view size more than offsets a three-order-of-magnitude increase in network size. If we only consider the first and third quartiles, a 2-fold increase suffices ($v_{\text{RPS}} = 80$), while a view of 160 yields even better accuracy with $100k$ nodes than a view of 50 with 200. One might wonder whether this increase in view size would not lead to excessive bandwidth consumption by the RPS protocol. But Figure 16a already demonstrated that increasing the view size from 50 to 200 has almost no impact on the full HEAP protocol. Overall, our analysis confirms the scalability of the RPS-based sample average estimation and thus that of our streaming solution.

## 5.5 Responsiveness to Churn

We now consider another aspect of the impact of the RPS protocol. We examine how it affects the ability to respond to node failures. We consider a worst-case scenario in which a subset of the nodes instantaneously fail and thereby stop contributing to the protocol. Such a catastrophic event impacts the protocol in two major ways. First, nodes that fail may have proposed stream packets to other nodes, and may have received requests that they can no longer honor. Second, their identifiers remain lingering in the RPS views of other nodes that may therefore continue to propose stream packets to them, thereby wasting time and bandwidth resources.

To evaluate the overall effect of these two issues, we ran several experiments in the ref-691 scenario with various values of $T_{\text{RPS}}$ and observed the behavior of HEAP after the failure of 20% (Figure 18a) and 50% of the nodes (Figure 19a). Figure 18a shows that HEAP recovers from




(a) 20% churn

(b) 20% churn - close-up

Figure 18: Percentage of HEAP nodes receiving a clear stream over time with a catastrophic failure of 20% of the nodes happening at t=60s.

the failure of 20% of the nodes almost instantly regardless of the value of $T_{\mathrm{RPS}}$. The plots show the percentage of nodes receiving each stream packet against the time at which the packet was generated at the source. We see that approximately 15% of the non failed nodes experience a short video/audio glitch (they lose a few stream packets) when the failure occurs. Figure 18b presents a close-up view of this glitch for the most unfavorable setting ($T_{\mathrm{RPS}} = 4s$). The plot shows that the glitch actually extends for as little as $2.25s$ and that most of the nodes recover even faster. The glitch starts before $t = 60s$ because of the stream lag of about $2s$—the plot shows stream time and not absolute time.

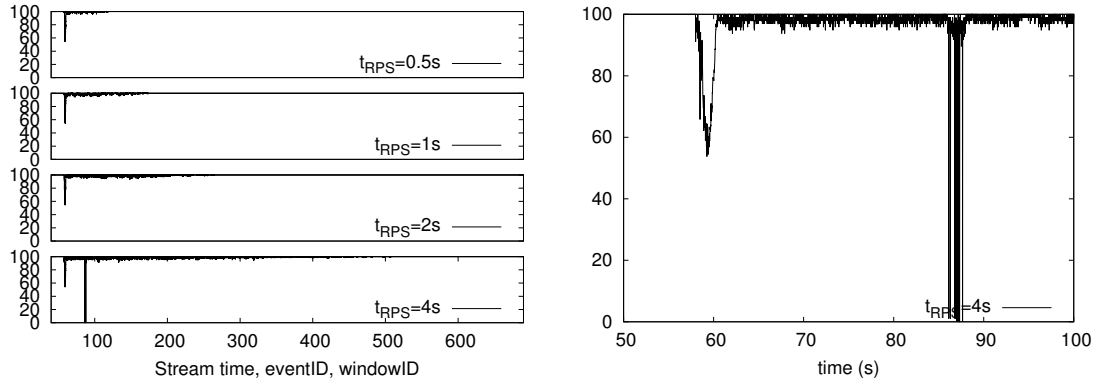
Figure 18a also shows that a small number of nodes also experience smaller glitches later during the experiment. These glitches continue throughout our 11-minute test for $T_{\mathrm{RPS}} = 4s$, but they become very infrequent after around 4 mins. For $T_{\mathrm{RPS}} = 2s$, the glitches become very infrequent after about 2 mins, and the last recorded glitch happens more than 3 mins before the end of the experiment. Finally, for $T_{\mathrm{RPS}} = 1s$ and $T_{\mathrm{RPS}} = 500ms$, we recorded only two, and only one additional glitch, with no glitches after respectively $6m32s$ and $3m33s$.

Figure 19a shows instead the results for the failure of 50% of the nodes. In this case, up to 50% of the nodes experience a glitch in their stream when the failure occurs. Moreover some glitches persist for about one extra minute when $T_{\mathrm{RPS}} = 500ms$, and for up to 8 minutes when $T_{\mathrm{RPS}} = 4s$. In this latter case, we also observe a second major glitch at around $t = 84s$. The close-up view in Figure 19b shows that this glitch results from a few packets being completely missed by all of the nodes. This results from the presence of stale node references in the RPS views of nodes. With $T_{\mathrm{RPS}} = 4s$, $t = 84s$ is only 6 gossip cycles after the failure, and completely purging failed nodes from a view of size 50 may take up to 50 cycles in the worst case.

If we join this analysis on churn with our previous analysis on bandwidth consumption (Figure 15c), we can observe that a value of $T_{\mathrm{RPS}} = 1s$ strikes a good balance between responsiveness—50% failure completely recovered in less than 2 mins—and bandwidth cost—virtually identical to that of $T_{\mathrm{RPS}} = 4s$.

## 5.6 Cohabitation with External Applications

Next, we evaluate the ability of our streaming protocol to operate in the presence of applications that compete for the use of upload bandwidth. To simulate the presence of an external




(a) 50% churn (b) 50% churn - close-up

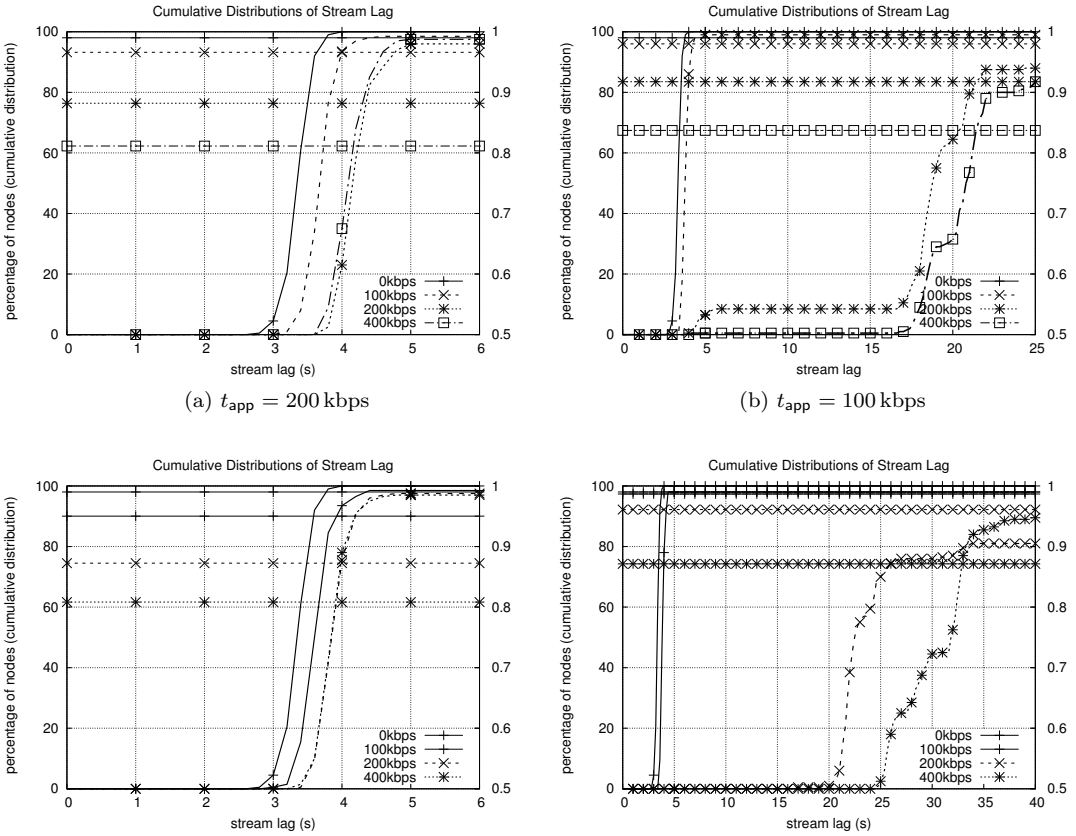
Figure 19: Percentage of nodes receiving a clear stream over time with a catastrophic failure of 50% of the nodes happening at t=60s.

bandwidth-intensive application, each node periodically runs a special bandwidth-consumption task that repeatedly sends UDP messages to the bootstrap node thereby attempting to consume $B_{\mathsf{app}}$ kbps of upload bandwidth. To simulate interactive usage such as uploading files to a website, or sending emails with attachments, nodes run the bandwidth consumption task with an on-off pattern. The task runs for $t_{on}$ periods of $10s$, and then pauses for $t_{off}$ periods of $10s$. We refer to the ratio $d_c = \frac{t_{on}}{t_{off}+t_{on}}$ as the application's *duty cycle*. During the active periods, the task periodically sends a UDP message of size $m$ every $t_{\mathsf{app}}$, resulting in a desired bandwidth consumption of $B_{\mathsf{app}} = \frac{m}{t_{\mathsf{app}}}$. We experimented with values of $t_{\mathsf{app}}$ between 50ms and 400ms, and set the message size $m$ to values corresponding to bandwidths, $B_{\mathsf{app}}$, from 100 kbps to 400 kbps.

Figure 20 depicts the results with varying values of $t_{\mathsf{app}}$ and $B_{\mathsf{app}}$ with a 50% duty cycle ($t_{on}$ $=t_{off}$ $=1$). Each plot shows two sets of curves. For each value of $B_{\mathsf{app}}$, the S-shaped line depicts the cumulative distribution of stream lag in the considered configuration. The horizontal line depicts the delivery rate achieved by the external application in the same experiment.

Figures 20a and 20b show the results with a 50% duty cycle and respectively $t_{\mathsf{app}} = 200$ms and $t_{\mathsf{app}} = 100$ms. With $t_{\mathsf{app}} = 200$ms, HEAP is almost unaffected by the external application and only experiences small increases in stream lag. With $B_{\mathsf{app}} = 100kbs$—14% of the available bandwidth, i.e. an average reduction of 7%—HEAP experiences a lag increase of less than half a second while the external application retains a very high delivery rate of almost 97%. When $B_{\mathsf{app}}$ increases ($B_{\mathsf{app}} = 200\,\text{kbs}\,, 400\,\text{kbps}$), HEAP still performs well, albeit with a lag increase of around 1s. This may seem surprising, but it comes at the cost of lower delivery rates for the external application, which scores at 88% for $B_{\mathsf{app}} = 200$ kbps, and at 81% for $B_{\mathsf{app}} = 400$ kbps.

With $t_{\mathsf{app}} = 100$ms, like in the above scenario, both HEAP and the external application keep performing well when the external application consumes 100 kbps during its active intervals. HEAP experiences a delay of less than 1s over the baseline, while the external application maintains a delivery rate of 98%. When $B_{\mathsf{app}}$ increases, however, the performance of HEAP decreases more drastically than with $t_{\mathsf{app}} = 200$ms. The average lag to receive a clear stream jumps to around 20s, and less than 10% of the nodes can still view a clear stream with a 5s lag with $B_{\mathsf{app}} = 200$ kbps, while none can with $B_{\mathsf{app}} = 300$ kbps. Yet, the external application achieves even higher delivery rates than with $t_{\mathsf{app}} = 200$ms: 92% for $B_{\mathsf{app}} = 200$ kbps and 87% for $B_{\mathsf{app}} = 300$ kbps.



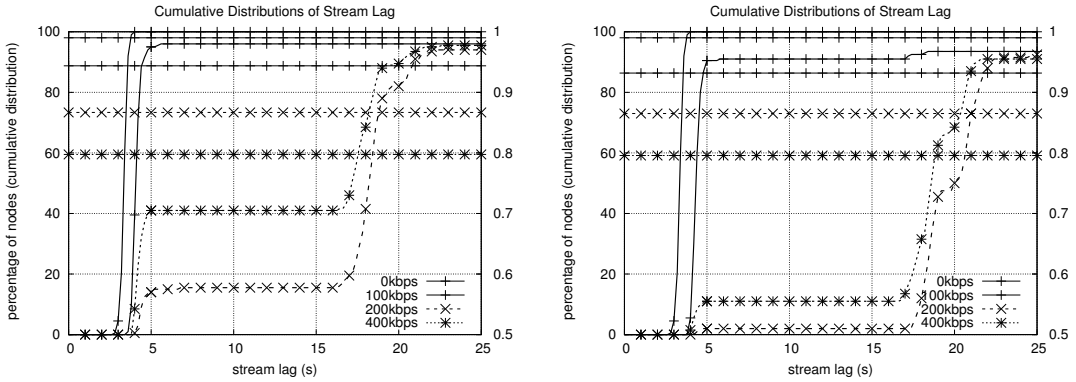

(a) $t_{\mathsf{app}} = 200\,\text{kbps}$ (b) $t_{\mathsf{app}} = 100\,\text{kbps}$

(c) $t_{\mathsf{app}} = 400\,\text{kbps}$ (d) $t_{\mathsf{app}} = 50\,\text{kbps}$

Figure 20: Stream Lag for HEAP and delivery rate for the external application for several values of $t_{\mathsf{app}}$ (different plots) and $B_{\mathsf{app}}$ (different lines) with an application duty cycle of 50%.

From this analysis, we can already observe an interesting trend. For a given value of $B_{\mathsf{app}}$, longer $t_{\mathsf{app}}$ intervals tend to favor HEAP, while shorter ones tend to favor the external application. To confirm this observation, Figures 20c and 20d display two even more extreme scenarios: $t_{\mathsf{app}} = 400\text{ms}$ and $t_{\mathsf{app}} = 50\text{ms}$. In the former, the performance of HEAP is almost unaffected even with $B_{\mathsf{app}}$ values of 400 kbps, although with a corresponding delivery rate of 81% for the external application. In the latter, instead, HEAP can still provide very good performance for $B_{\mathsf{app}} = 100\,\text{kbps}$, but it incurs very high delays for higher $B_{\mathsf{app}}$ values. Yet, even in the most unfavorable scenario, HEAP can provide a clear stream to over 90% of the nodes with a lag of less than $40s$. The corresponding delivery rates for the external application remain higher than 87%.

Figure 21 complements these results with those obtained in scenarios with higher duty cycles for the external application. In particular, Figure 21a depicts a duty cycle of 66%, and Figure 21b a duty cycle of 75% for $t_{\mathsf{app}} = 200\text{ms}$. Clearly, increasing the duty cycle increases the average bandwidth consumption of the external application, thereby increasing its impact on HEAP. Yet, the performance for $B_{\mathsf{app}} = 100\,\text{kbps}$ remains very good, with 97% of the nodes receiving a clear stream within less than $5s$ with a 66% duty cycle, and 94% of the nodes achieving the




(a) $d_c = 66.6\%$ (b) $d_c = 75\%$

Figure 21: Stream Lag for HEAP and delivery rate for the external application for higher duty cycles ($d_c$) and several values of $B_{\mathsf{app}}$ (different lines) with $t_{\mathsf{app}} = 200ms$.

same result with a 75% duty cycle. Performance for higher values of $B_{\mathsf{app}}$ drops, but even with $B_{\mathsf{app}} = 400$ kbps, HEAP manages to yield a clear stream to over 95% of the users albeit with a pretty large lag: $21s$ with a duty cycle of 66%, and $22s$ with a duty cycle of 75%. The corresponding delivery rates for the external application remain above 80% in all cases.

We conclude this section by observing that with $t_{\mathsf{app}} = 400ms$ (plot not shown), HEAP provides good performance with a lag of less than $5s$ even with a 75% duty cycle and a $B_{\mathsf{app}}$ value of 400 kbps. The external application achieves a corresponding delivery rate of 78% in the worst case.

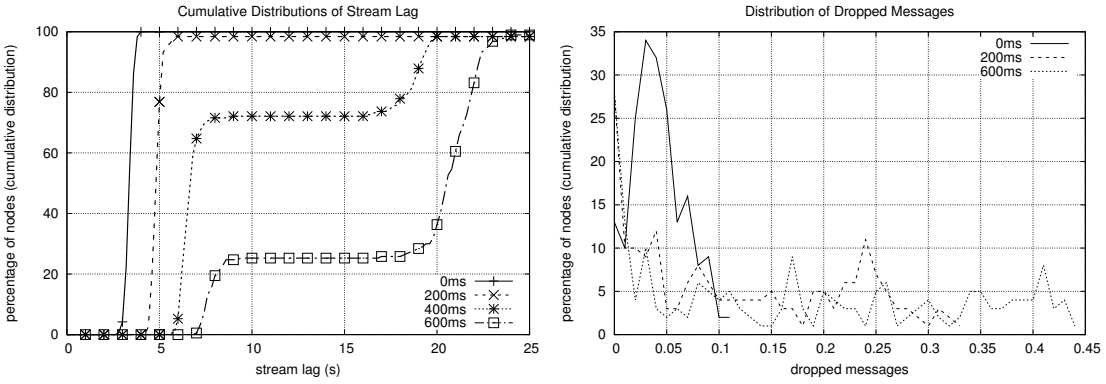
# 6 Improving Performance and Applicability

Section 5 demonstrated the effectiveness of HEAP with respect to standard gossip when operating in a heterogeneous setting and dissected its performance by analyzing the strengths and the weaknesses of gossip-based streaming. In the following, we start from these weaknesses and introduce two new features that further optimize the performance of our protocol.

## 6.1 Skip-Proposal Optimization

Our first optimization stems from the observation of Figure 12. As discussed in Section 5.2.4, the figure shows an extreme variability in the bandwidth consumption of nodes over time. This variability results from the inherent randomness of gossip dissemination. Consider two nodes, $A$, and $B$. Node $A$ receives a stream packet $p$ directly from the source. Most, if not all, the nodes to which $A$ proposes a packet will not have received it and will thus request it. Node $B$ instead receives the same packet at one of the last dissemination hops. Even if $B$ proposes the packet to a large number of other nodes, most of these will already have received it. Hence, $B$ will receive very few requests, if any.

This simple example shows that the dissemination of a stream packet may require very different amounts of bandwidth from different nodes: node $A$ will employ much more upload bandwidth than node $B$. Yet, the very nature of gossip averages out this inequality in dissemination costs over the protocol's execution. By continuously changing communication partners, nodes




(a) Stream Lag (b) Dropped Messages

Figure 22: Stream lag and distribution of the fraction of messages dropped by the token bucket with various values of heterogeneous latency.

receive packets across all dissemination hops in a few rounds. This leads to an even distribution of average available bandwidth over sufficiently long intervals.

### 6.1.1 Heterogeneous Node Latency

This self-averaging capability lies at the basis of the good performance highlighted in Section 5.2. Yet, it relies on the assumptions that nodes experience the same average processing and communication delays, which is not always the case in reality. In many situations, we can identify a subset of nodes that consistently respond more slowly than others to the messages they receive. This may result from a number of reasons such as CPU overload, or high-latency connections (e.g., in mobile networks).

To evaluate the impact of such a heterogeneous setting, we artificially introduce an additional delay to the messages sent by nodes. Before the experiment, each node chooses a uniformly-random delay value in $[0, d]$ms, and sticks to it for the duration of the experiment. Unlike the random delay described in Section 4.3, which varies at each sent message, this fixed delay remains the same across all the messages sent by a node. As a result, it establishes a heterogeneous latency distribution across all the nodes.

Figure 22 shows the results of experiments carried out with a 200KB token bucket in a scenario with no fixed delay and in three scenarios with fixed delays ($d = 200ms$, $d = 400ms$, $d = 600ms$). Figure 22a depicts the cumulative stream lag distribution, while Figure 22b presents the distribution of dropped messages across all the nodes.

Figure 22a highlights the important impact of node latency on the performance of the protocol. With a value of $d = 200$ms, the average stream lag increases by almost 1.5s, and with $d \geq 400$ms, we can clearly distinguish two groups of nodes. The first group receives a clear stream with a relatively short lag, but still much longer than with $d = 0$ (6s with $d = 400$ms, and 7.5s with $d = 600$ms). The second group, instead, experiences significantly longer delays of 18s with $d = 400$ms and 22s with $d = 600$ms. The first group corresponds to nodes that receive all the stream packets without having to rely on too many retransmission operations, while the second involves nodes for which some of the received packets have had to go through multiple retransmission operations thereby incurring much longer delays.

Figure 22b confirms the impact of node latency on dropped messages by showing the distribution of the number of messages dropped by the token bucket. In the presence of uniform delays ($d = 0ms$), the distribution of dropped messages remains well centered around an average of 5%. However, as soon as $d$ increases, the distribution flattens out; and when $d = 400ms$, some nodes experience a message-drop rate of up to 45%.

Although not shown in the plot, we observed that the nodes with the lowest delays experience the highest message-drop rates. Consider two nodes, $F$ (fast) and $S$ (slow) with fixed delays of $50ms$ and $400ms$. Let $F$ and $S$ receive a stream packet at the same time. Both will propose the stream packet to other nodes. However, a node that receives both a proposal from $F$ and one from $S$ will certainly respond to the one from $F$ because the one from $S$ has arrived too late. Nodes with low delay values will therefore be requested more often, and will therefore try to send more [SERVE] messages, which will in turn cause them to exceed their bandwidth capabilities.

### 6.1.2 Skip-Proposal Heuristic

To address this new source of heterogeneity, we slightly extend our system model by giving nodes the ability to sense the current status of their bandwidth limiters. In the case of a token-bucket limiter, nodes sense the available burst size: i.e., how large a message they can send instantly. In the case of a leaky-bucket, they instead sense the current size of the sending queue. In both cases, this may be achieved either through system calls where available, or by employing an application-level bandwidth shaper that matches the expected capability of the network connection.

This simple sensing ability allows us to integrate our protocol with a simple but effective heuristic. Nodes that are currently hitting their bandwidth limits stop sending proposals to other nodes until their bandwidth usage drops. However, they continue to send [REQUEST] and [SERVE] messages normally.

It is important to observe that skipping proposals turns out to be more beneficial than skipping [SERVE] messages. Nodes send proposals redundantly using gossip, while they unicast [SERVE] messages to nodes that have explicitly requested them. So a node $A$ that does not send a [SERVE] message to a node $B$ forces $B$ to rely on the retransmission mechanisms, which ultimately harms $B$'s ability to receive the corresponding stream packets in a timely manner. On the other hand, if node $A$ decides not to send a proposal to $B$, $B$ will most likely receive the same proposal from another node within a very short time.

We consider two versions of this skip-proposal optimization: *Skip* and *SkipDrop*. In *Skip*, nodes store skipped proposal messages in a queue and send them at later rounds. In *Skip-Drop*, they simply drop the skipped messages. We evaluate both versions of the optimization in Section 7.1.

## 6.2 Push Optimization

To introduce our second optimization, we observe that both the basic protocol we described in Section 2.1 and our main contribution, HEAP, rely on a three-phase approach. A node sends stream packets only to nodes that actually requested them after receiving a proposal message.

However, this approach only serves a purpose for packets that have already been received by a majority of nodes. Consider, for example a packet that has just been generated by the source. None of the other nodes may have received it, which makes a three-phase approach completely useless. But even a packet that has traveled for just one or two hops will remain unknown to most of the participating nodes.

Our second optimization therefore consists of a variant of the protocol that pushes stream packets by sending [SERVE] messages instead of proposal messages in the initial phases of a packet's

dissemination. The probability to send a packet to a node that has already received it clearly depends on the amount of time the packet has traveled. It therefore makes sense to push stream packets for the first few hops, up to a push threshold, $\theta_P$, and then revert back to the standard three-phase model to complete the dissemination process.

This optimization takes inspiration from Pulp [30], a hybrid gossip protocol that combines a push-based and an anti-entropy protocol, However, the anti-entropy protocol employed by Pulp blindly attempts to pull information from other nodes without knowing if the information is there. HEAP instead employs a three-phase approach and only pulls (requests) stream packets if these are known to be available.

# 7 Extension Evaluation

We now evaluate the two extensions introduced in Section 6. First, we show how the skip-proposal optimization makes it possible to achieve very good performance in the presence of slow nodes. Then, we demonstrate how pushing stream packets in the first few hops can significantly improve the stream lag.

## 7.1 Skip Proposal Optimization

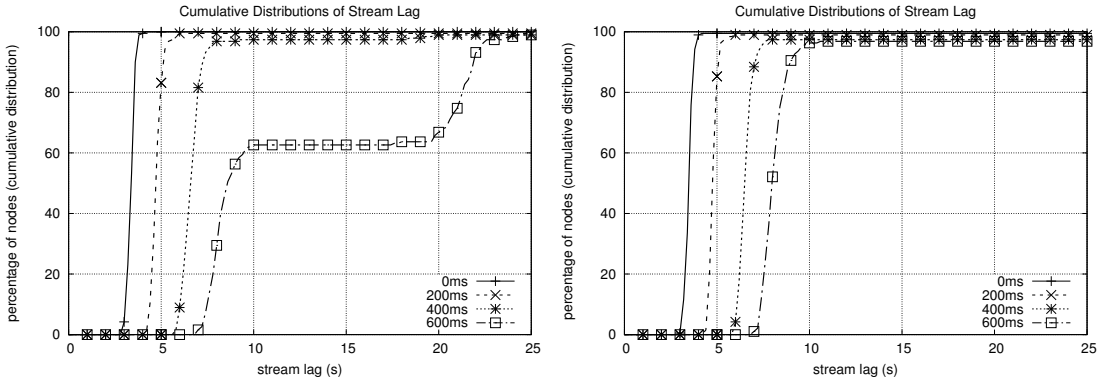
To evaluate the effect of the skip-proposal heuristic presented in Section 6.1, we consider the same scenario as in Figure 22a. Figures 23 and 24 show respectively the stream lag and the distribution of dropped messages when using the *Skip* (Figures 23a and 24a) and the *SkipDrop* versions of the heuristic (Figures 23b and 24b).

Figure 23a shows that skipping proposals has a significant impact for values of $d = 400$ms and 600ms. While the additional stream lag due to the presence of slow nodes remains, the *Skip* heuristic completely removes the vicious cycle of dropped messages and retransmission when $d = 400$ms. The worst stream lag recorded for this scenario is therefore only 4.5s worse than for $d = 0$. The heuristic also provides a significant benefit for $d = 600$ms, but it does not completely eliminate the interaction between dropped messages retransmission. Over 60% of the nodes manage to view a clear stream with a lag of less than 10s, i.e. 5.5s later than with $d = 0$. However, as many as 40% still experience huge lags of over 20s. The *SkipDrop* version of the heuristic completely solves even this problem. As shown in Figure 23b, it allows all nodes to view the stream within a reasonable delay even when $d = 600$ms.

Figure 24 shows how these good results stem from the ability of *Skip* and *SkipDrop* to equalize bandwidth consumption across nodes, thereby reducing the number of lost messages by a significant fraction. In both figures, most of the nodes drop no or very few messages. With *Skip*, there are still nodes that drop 40% of the messages, but these amount to less than 1% of the nodes. With *SkipDrop*, the result is even more impressive: the maximum drop-rate falls to a value of 16%, and even here, for less than a handful of nodes. In both cases, this constitutes a significant improvement with respect to the distribution in Figure 22.

## 7.2 Push Optimization

We now move to the evaluation of the push optimization. Figures 25 through 28 analyze the performance of the push-based optimization we presented in Section 6.2. The four figures compare various configurations of the push-optimized *HEAP* (*Push* in the following) with two baselines: standard *HEAP*, and *sourcepush*—a variant that applies the push optimization only at the source while the remaining nodes operate using the standard *HEAP* protocol. The plots evaluate *Push* with various combinations of push-threshold and push-fanout values. For clarity, we separated




(a) *Skip* (b) *SkipDrop*

Figure 23: Stream Lag for various heterogeneous delay values while using the *Skip* and *SkipDrop* versions of the heuristic.

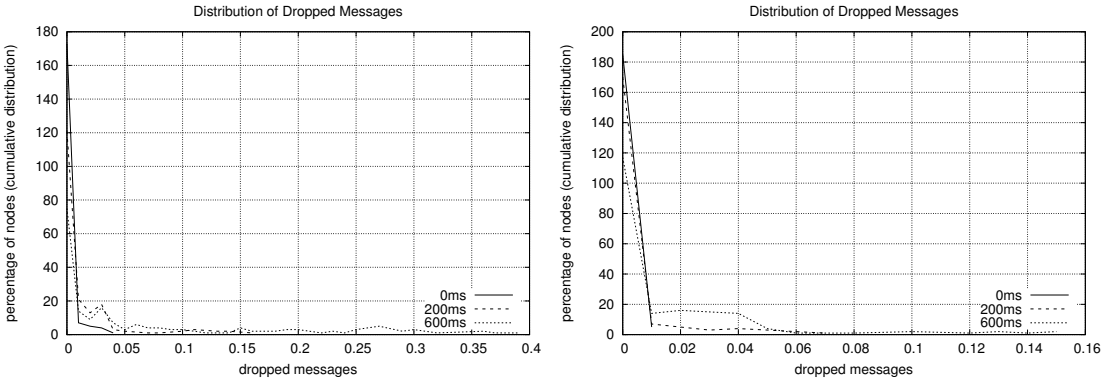
data for different push-threshold values in different plots—Figures 25 and 27 for $\theta_P = 2$, and Figures 26 and 28 for $\theta_P = 3$. Figures 25 and 26 present the results with a token-bucket bandwidth limiter while Figures 27 and 28 present those obtained with a leaky bucket. In each figure, the left plot depicts results obtained with homogeneous bandwidth, while the right one those with the ms-691 heterogeneous distribution.

Let us start by analyzing Figure 25. Both plots show that the simple *sourcepush* optimization already provides a non-negligible improvement with respect to basic *HEAP*, with an average delay of 3.0s against 3.3s for basic *HEAP*. *Push* goes beyond this improvement: in both bandwidth scenarios, *Push* with $\theta_P = 2$ and $f_P = 7$ obtains an average delay of 2.8s. The plot also shows the importance of the push-fanout parameter, $f_P$: performance improves with increasing values of $f_P$.

Figure 26 completes the picture for the token-bucket configuration by showing the performance of *Push* with a push threshold $\theta_P = 3$. The results confirm the expectations we stated in Section 6.2: the push optimization only provides a significant advantage in the first hops of the dissemination process. The only variant that performs well with $\theta_P = 3$ is the one with $f_P = 3$, and even this yields similar performance as *sourcepush* in the homogeneous case (Figure 26a), and slightly worse performance in the heterogeneous ms-691 setting (Figure 26b). The configurations with $f_P = 7$, which achieved the best performance in Figure 25, perform instead much worse than the *HEAP* baseline.

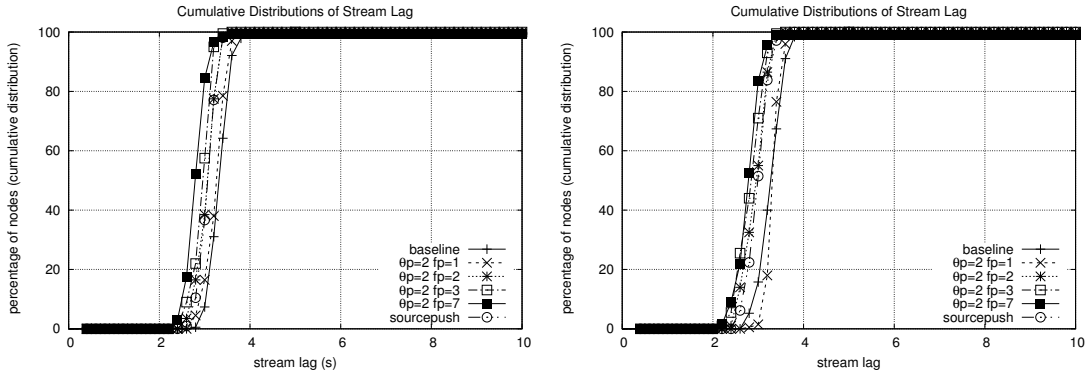
Figures 27 and 28 show that the push optimization plays a more significant role in the presence of a leaky-bucket bandwidth limiter. In Figure 27, *Push* with $\theta_P = 2$ and $f_P = 2$ (or $f_P = 3$), yields an average lag improvement of 4s over the *HEAP* baseline and of 3s over *sourcepush* in both bandwidth scenarios. Yet, $f_P = 7$ achieves very poor performance with none of the nodes receiving a clear stream. In Figure 28, *Push* with $\theta_P = 3$ also achieves very good performance, but this time with values of $f_P = 1$ and $f_P = 2$. The $f_P = 3$ option performs instead much worse than the baseline.

Overall, these results show that the push optimization offers a significant advantage in the first hops of the dissemination process, confirming the results of [30]. Moreover, its impact turns out to be particularly significant in the presence of a leaky-bucket limiter. We can easily extrapolate that the associated improvements would be even greater in larger networks where




(a) *Skip* (b) *SkipDrop*

Figure 24: Distribution of dropped messages for various heterogeneous delay values while using the *Skip* and *SkipDrop* versions of the heuristic.




(a) homo-691 (b) ms-691

Figure 25: Impact of push in the first two hops with a token-bucket configuration, with homogeneous (left) and heterogeneous (right) bandwidth.

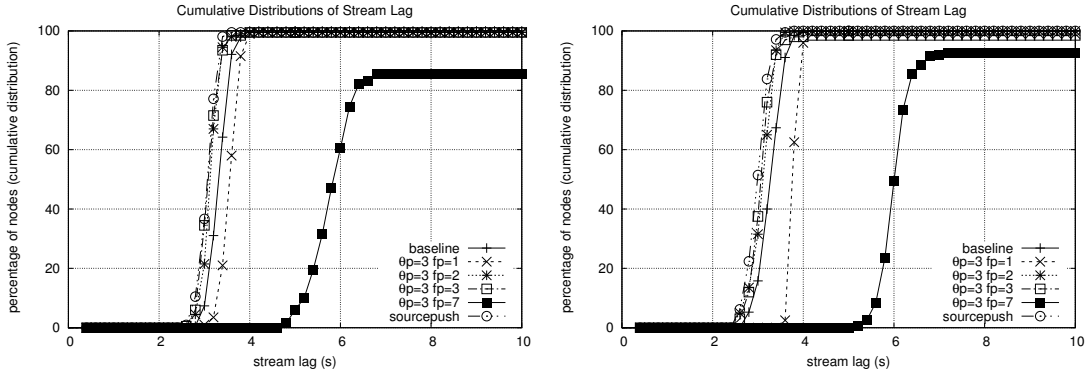
we could successfully reach even higher values of $\theta_P$.

# 8 Related Work

Video streaming has attracted a lot of attention from research and industry for the last fifteen years [19, 74, 53, 59, 66, 12]. In the following, we review the major contributions to this area by distinguishing tree-based, mesh-based, and gossip-based protocols.

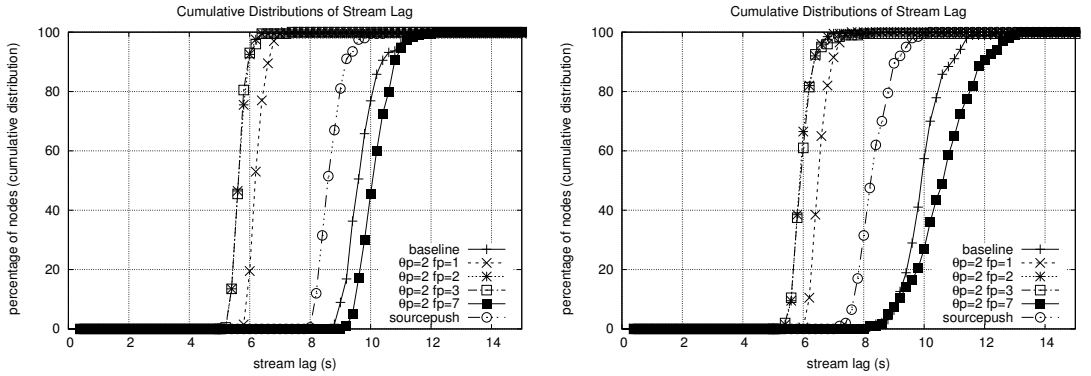
*Tree-based protocols.* The first decentralized video streaming protocols appeared as a natural evolution of application-level multicast [23, 24, 32, 22]. These consisted of overlay tree structures, built either with stand-alone [34] protocols, or as a subset of an underlying mesh overlay such as a DHT [68, 69].

But these approaches all suffered from two major drawbacks. First, trees tend to be partic-




(a) homo-691 (b) ms-691

Figure 26: Impact of push in the first three hops with a token-bucket configuration, with homogeneous (left) and heterogeneous (right) bandwidth.



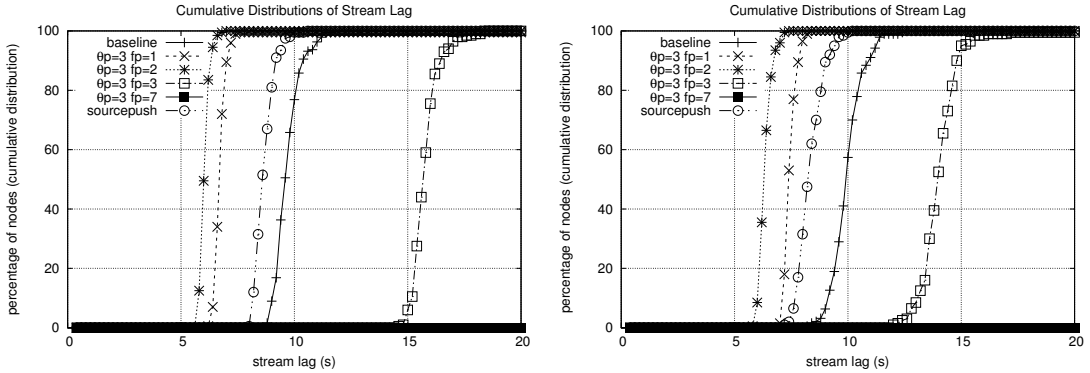

(a) homo-691 (b) ms-691

Figure 27: Impact of push in the first two hops with a leaky-bucket configuration, with homogeneous (left) and heterogeneous (right) bandwidth.

ularly vulnerable to churn. Failed nodes have a huge impact on their children since they isolate their sub-branches from the rest of the tree. Second, even in the absence of churn, trees inherently force internal nodes to carry out all the forwarding work, while leaves do nothing. Since internal nodes constitute only a fraction of the nodes in a tree, this leads to a very inefficient use of bandwidth.

As a first attempt to address these drawbacks, several authors investigated the need to take into account bandwidth heterogeneity and churn in tree-based protocols. [15] and [70] propose a set of heuristics that lead 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 approach.

But above all, the limitations of tree-based solution led researchers to explore multi-tree schemes such as Splitstream [19] or Chunkyspread [74]. These protocols combine multiple intertwining trees in which the internal nodes of a tree act as leaves in the others. The source




(a) homo-691 (b) ms-691

Figure 28: Impact of push in the first three hops with a leaky-bucket configuration, with homogeneous (left) and heterogeneous (right) bandwidth.

then splits the stream over the diverse paths defined by these various trees. This enhances reliability and equalizes bandwidth usage, two features that come for free in gossip protocols where the neighbors of a node continuously change. Some of these multi-tree protocols also support heterogeneous bandwidth distributions. For example, Chunkyspread accounts for heterogeneity using the SwapLinks protocol [76]. 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.

More recently, Streamline [55] has also addressed the problem of operation with heterogeneous bandwidth by building a spanning distribution tree over an adaptive mesh structure. However, the paper only uses a single tree thereby leaving a large percentage of nodes underutilized. Later work by the same authors [5] focuses instead on using server replication to improve availability in video-on-demand systems.

EagleMacaw [4] also targets video-on-demand and builds on [5] to address the trade-off between bandwidth efficiency and stability in dissemination trees. In particular, it uses two trees: one that optimizes latency, and another that optimizes reliability. However, the reliable tree is only exploited in the event of a partition in the efficient tree. As a result, the leaves in the efficient tree remain passive most of the time.

*Mesh-based protocols.* Mesh-based systems [48, 51, 58, 54, 57, 22] represent a further departure from the single-tree approach. Like gossip, they exploit an unstructured topology. But they maintain this topology static. Some of these protocols, namely the latest version of Coolstreaming [48] and GridMedia [81] dynamically build multi-trees on top of an 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 properly operating. Not surprisingly, it was shown in [48, 51] 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.

The work in [78] adresses the problem of building an optimized mesh in terms of network proximity and latency, in the presence of *guarded* peers, i.e., peers that are behind a NAT or firewall. This work led to mixing application level multicast with IP multicast whenever

possible [80]. The core of this research is now commercially used in [79] but little is known on the dissemination protocol. At the time the prototype was used for research, some nodes were fed by super peers deployed on PlanetLab and it is reasonable to think that those super peers are now replaced by dedicated servers in the commercial product. It is for instance known that the dissemination protocol of PPLive [59] substantially relies on a set of super peers and thus does not represent a purely decentralized solution [36].

Finally, Bullet [46] is a 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.

*Gossip-based protocols.* Gossip protocols were initially introduced to disseminate *small updates* [26, 52, 14, 29, 44] in applications such as distributed data bases. Like for trees, their application to video streaming finds its root in application-level multicast protocols [29, 44, 14]. However, even if such early work considered streaming as an application, these protocols failed to take bandwidth considerations into account.

CREW [27] constituted one of the first gossip protocols focusing on high-bandwidth dissemination, file sharing in particular. BAR Gossip [50] and FlightPath [49] tackled instead the issue of live streaming in the presence of byzantine nodes. But they focused more on tolerating malicious nodes than in providing efficient use of bandwidth. They remain interesting in our context because they offered the first examples of three-phase protocols like the one we described in Section 3. Moreover, CREW [27] also includes some adaptation features with nodes deciding to stop offering data when their bandwidth is exhausted. Similarly, in Smart Gossip [47], nodes of a wireless network may decide not to gossip depending on the number of nodes in their surrounding. As another early example of adaptation, Gravitational Gossip [42] adjusts the fanin of nodes (i.e., the number of times a node is chosen as a gossip target) based on the quality of service they expect by biasing the underlying peer-sampling.

More recently, several authors have proposed gossip-based streaming platforms similar to ours. Peerstreamer [72], developed within the NAPA wine [12] project, proposes a streaming platform that addresses differences in the upload capabilities of participating nodes by using heterogeneous numbers of propose packets ("offers" in the PeerStreamer jargon). However, instead of adapting the value of a node's fanout as we do in HEAP, Peerstreamer varies the number of parallel threads that propose stream packets to other nodes [13]. In a follow-up project [7], some of the authors behind Peerstreamer studied the performance of this decentralized solution in a scenario consisting of several wireless community networks (WCN) [31].

Another line of work has studied the impact of packet-scheduling strategies in the context of decentralized streaming protocols and proposed analytically optimal, or close-to-optimal protocols [81, 16, 35, 51, 58, 56, 1]. For example, [1] proves the optimality of latest-useful and of a deadline-based chunk scheduling algorithms.

Recent work has also started to investigate how the peer-to-peer and gossip-based paradigms can support HTTP streaming [63, 65, 67]. HTTP streaming differs from traditional streaming in that HTTP clients request packets from the server when they need them, rather than being served proactively by the server. Peer2View [63] and SmoothCache [65, 67] insert a layer between client and server consisting of a network of peer-to-peer agents. Each agent collaborates with the others to pre-fetch packets and make them available to clients. The same authors have extended the work in these two papers by introducing the ability to stream over WebRTC [64]. Two major difference between these systems and our approach lie in their use of servers for operations like membership management, and in their evaluation in much less constrained bandwidth scenarios. For example, [67] defines a resource-factor, equivalent to the capacity-supply ratio (CSR) in

Section 4.3, larger than 2 while our least constrained scenario has a CSR of 1.2.

Finally, HEAP exploits the RPS to obtain an estimation of the average bandwidth capability of nodes. A straightforward extension could consist in replacing this approximation with the result of gossip-based aggregation protocols like the one presented in [39]. Given the resilience of HEAP to imprecise estimations we deemed this unnecessary, but in the presence of stable bandwidth scenarios the use of an averaging protocol might provide a slight boost in performance.

# 9 Concluding Remarks

In this paper we presented HEAP, a novel gossip-based video streaming solution designed to operate in heterogeneous and bandwidth-constrained environments. HEAP preserves the simplicity and proactive (churn adaptation) nature of traditional homogeneous gossip, while significantly improving its effectiveness. Through its four components, HEAP is able to adapt the fanouts of nodes to their bandwidth and delay characteristics while, at the same time, implementing effective reliability measures through FEC encoding and retransmission. Experimental results show that HEAP significantly improves stream quality over a standard homogeneous gossip protocol, while scaling to very large networks.

# acks

Where not otherwise indicated, our experiments were carried out using the Grid'5000 [8] testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). The authors wish to thank Vivien Quèma, Boris Koldehofe, Martin Mogensen, and Arnaud Jegou for their suggestions on earlier versions of this work.

# References

[1] Luca Abeni, Csaba Kiraly, and Renato Lo Cigno. *On the Optimal Scheduling of Streaming Applications in Unstructured Meshes*, volume 5550 of *Lecture Notes in Computer Science*, pages 117–130. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-01399-7.

[2] Emmanuelle Anceaume, Yann Busnel, and Sébastien Gambs. Uniform and ergodic sampling in unstructured peer-to-peer systems with malicious nodes. In *Proceedings of the 14th International Conference on Principles of Distributed Systems*, OPODIS'10, pages 64–78, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] Mugurel Ionut Andreica and Nicolae Tapus. Efficient upload bandwidth estimation and communication resource allocation techniques. In *Proceedings of the 9th WSEAS international conference on signal, speech and image processing, and 9th WSEAS international conference on Multimedia, internet & video technologies*, SSIP '09/MIV'09, pages 186–191, Stevens Point, Wisconsin, USA, 2009. World Scientific and Engineering Academy and Society (WSEAS).

[4] S. Ataee and B. Garbinato. Eaglemacaw: A dual-tree replication protocol for efficient and reliable p2p media streaming. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 112–121, Feb 2014.

[5] S. Ataee, B. Garbinato, and F. Pedone. Restream - a replication algorithm for reliable and scalable multimedia streaming. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 68–76, Feb 2013.

[6] Ozalp Babaoglu, Márk Jelasity, Anne-Marie Kermarrec, Alberto Montresor, and Maarten van Steen. Managing clouds: A case for a fresh look at large unreliable dynamic networks. *SIGOPS Oper. Syst. Rev.*, 40(3):9–13, July 2006.

[7] Luca Baldesi, Leonardo Maccari, and Renato Lo Cigno. Improving p2p streaming in wireless community networks. *Comput. Netw.*, 93(P2):389–403, December 2015.

[8] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In IvanI. Ivanov, Marten Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013. `http://www.grid5000.fr`.

[9] Ranieri Baraglia, Patrizio Dazzi, Matteo Mordacchini, Laura Ricci, and Luca Alessi. Group: A gossip based building community protocol. In *Proceedings of the 11th International Conference and 4th International Conference on Smart Spaces and Next Generation Wired/Wireless Networking*, NEW2AN'11/ruSMART'11, pages 496–507, Berlin, Heidelberg, 2011. Springer-Verlag.

[10] Mayank Bawa, Hrishikesh Deshpande, and Hector Garcia-Molina. Transience of peers & streaming media. *SIGCOMM Comput. Commun. Rev.*, 33(1):107–112, January 2003.

[11] Marin Bertier, Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, and Vincent Leroy. The gossple anonymous social network. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware '10, pages 191–211, Berlin, Heidelberg, 2010. Springer-Verlag.

[12] R. Birke, E. Leonardi, M. Mellia, A. Bakay, T. Szemethy, C. Kiraly, R. L. Cigno, F. Mathieu, L. Muscariello, S. Niccolini, J. Seedorf, and G. Tropea. Architecture of a network-aware p2p-tv application: the napa-wine approach. *IEEE Communications Magazine*, 49(6):154–163, June 2011.

[13] Robert Birke, Csaba Kiraly, Emilio Leonardi, Marco Mellia, Michela Meo, and Stefano Traverso. A delay-based aggregate rate control for p2p streaming systems. *Comput. Commun.*, 35(18):2237–2244, November 2012.

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

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

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

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

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

[19] 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.

[20] 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.

[21] Joydeep Chandra, Santosh Kumar Shaw, and Niloy Ganguly. Hpc5: An efficient topology generation mechanism for gnutella networks. *Comput. Netw.*, 54(9):1440–1459, June 2010.

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

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

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

[25] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The bayou architecture: Support for data sharing among mobile users. In *Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*, WMCSA '94, pages 2–7, Washington, DC, USA, 1994. IEEE Computer Society.

[26] 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.

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

[28] Fady Draidi, Esther Pacitti, Didier Parigot, and Guillaume Verger. P2prec: A social-based p2p recommendation system. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 2593–2596, New York, NY, USA, 2011. ACM.

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

[30] Pascal Felber, Anne-Marie Kermarrec, Lorenzo Leonini, Etienne RiviÃ¨re, and Spyros Voulgaris. Pulp: An adaptive gossip-based dissemination protocol for multi-source message streams. *Peer-to-Peer Networking and Applications*, 5(1):74–91, 2012.

[31] Rob Flickenger. *Building Wireless Community Community Networks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2 edition, 2003.

[32] Paul Francis. Yoid: Extending the Internet Multicast Architecture. `http://www.icir.org/yoid/`, 1997.

[33] Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Maxime Monod, and Vivien Quéma. Stretching Gossip with Live Streaming. In *DSN*, 2009.

[34] Davide Frey and Amy L. Murphy. Failure-tolerant overlay trees for large-scale dynamic networks. In Klaus Wehrle, Wolfgang Kellerer, Sandeep K. Singhal, and Ralf Steinmetz, editors, *Proceedings P2P'08, Eighth International Conference on Peer-to-Peer Computing, 8-11 September 2008, Aachen, Germany*, pages 351–361. IEEE Computer Society, 2008.

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

[36] 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.

[37] Manish Jain and Constantinos Dovrolis. End-to-end available bandwidth: measurement methodology, dynamics, and relation with tcp throughput. *IEEE/ACM Trans. Netw.*, 11:537–549, August 2003.

[38] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole, Jr. Overcast: Reliable multicasting with on overlay network. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 14–14, Berkeley, CA, USA, 2000. USENIX Association.

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

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

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

[42] Kate Jenkins, Kenneth Hopkinson, and Ken Birman. A Gossip Protocol for Subgroup Multicast. In *Proc. of ICDCS Workshops*, 2001.

[43] Gian Paolo Jesi, Edoardo Mollona, Srijith K. Nair, and Maarten van Steen. Prestige-based peer sampling service: Interdisciplinary approach to secure gossip. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1209–1213, New York, NY, USA, 2009. ACM.

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

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

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

[47] Pradeep Kyasanur, Romit Roy Choudhury, and Indranil Gupta. Smart Gossip: An Adaptive Gossip-based Broadcasting Service for Sensor Networks. In *Proc. of MASS*, 2006.

[48] 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.

[49] 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.

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

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

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

[53] Yong Liu, Yang Guo, and Chao Liang. A survey on peer-to-peer video streaming systems. *Peer-to-Peer Networking and Applications*, 1(1):18–28, 2008.

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

[55] A. Malekpour, F. Pedone, M. Allani, and B. Garbinato. Streamline: An architecture for overlay multicast. In *Network Computing and Applications, 2009. NCA 2009. Eighth IEEE International Symposium on*, pages 44–51, July 2009.

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

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

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

[59] PPLive. Pplive. `http://www.pptv.com`, 2016.

[60] R. S. Prasad, M. Murray, C. Dovrolis, K. Claffy, Ravi Prasad, and Constantinos Dovrolis Georgia. Bandwidth estimation: Metrics, measurement techniques, and tools. *IEEE Network*, 17:27–35, 2003.

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

[62] R. Roverso, J. Dowling, and M. Jelasity. Through the wormhole: Low cost, fresh peer sampling for the internet. In *IEEE P2P 2013 Proceedings*, pages 1–10, Sept 2013.

[63] R. Roverso, S. El-Ansary, and S. Haridi. Peer2view: A peer-to-peer http-live streaming platform. In *2012 IEEE 12th International Conference on Peer-to-Peer Computing (P2P)*, pages 65–66, Sept 2012.

[64] R. Roverso and M. HÃ¶gqvist. Hive.js: Browser-based distributed caching for adaptive video streaming. In *Multimedia (ISM), 2014 IEEE International Symposium on*, pages 143–146, Dec 2014.

[65] Roberto Roverso, Sameh El-Ansary, and Seif Haridi. *NETWORKING 2012: 11th International IFIP TC 6 Networking Conference, Prague, Czech Republic, May 21-25, 2012, Proceedings, Part II*, chapter SmoothCache: HTTP-Live Streaming Goes Peer-to-Peer, pages 29–43. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[66] Roberto Roverso, Sameh El-Ansary, and Mikael Högqvist. On http live streaming in large enterprises. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 489–490, New York, NY, USA, 2013. ACM.

[67] Roberto Roverso, Riccardo Reale, Sameh El-Ansary, and Seif Haridi. Smoothcache 2.0: Cdn-quality adaptive http live streaming on peer-to-peer overlays. In *Proceedings of the 6th ACM Multimedia Systems Conference*, MMSys '15, pages 61–72, New York, NY, USA, 2015. ACM.

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

[69] 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.

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

[71] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.

[72] Stefano Traverso, Luca Abeni, Robert Birke, Csaba Kiraly, Emilio Leonardi, Renato Lo Cigno, and Marco Mellia. Neighborhood filtering strategies for overlay construction in p2p-tv systems: Design and experimental comparison. *IEEE/ACM Trans. Netw.*, 23(3):741–754, June 2015.

[73] J. Turner. New directions in communications (or which way to the information age?). *Communications Magazine, IEEE*, 24(10):8–15, October 1986.

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

[75] Ymir Vigfusson, Ken Birman, Qi Huang, and Deepak P. Nataraj. Optimizing information flow in the gossip objects platform. *SIGOPS Oper. Syst. Rev.*, 44(2):71–76, April 2010.

[76] V. Vishnumurthy and P. Francis. On Heterogeneous Overlay Construction and Random Node Selection in Unstructured P2P Networks. In *INFOCOM*, 2006.

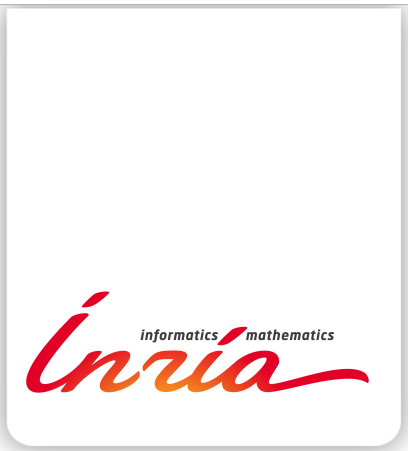
[77] Spyros Voulgaris, Márk Jelasity, and Maarten van Steen. *A Robust and Scalable Peer-to-Peer Gossiping Protocol*, pages 47–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

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

[79] Zattoo. Zattoo. `http://www.zattoo.com`, 2016.

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

[81] 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.




**RESEARCH CENTRE**
**RENNES – BRETAGNE ATLANTIQUE**

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399