An Adaptive Peer-Sampling Protocol for Building Networks of Browsers

Brice Nédelec, Julian Tanke, Pascal Molli, Achour Mostefaoui, Davide Frey

An Adaptive Peer-Sampling Protocol for Building Networks of Browsers

Brice Nédelec[1], Julian Tanke[1], Pascal Molli[1], Achour Mostéfaoui[1], and Davide Frey[2]

[1]L2SN, University of Nantes, 2 rue de la Houssinière, BP 92208, 44322 Nantes Cedex 3, France,
first.last@univ-nantes.fr
[2]INRIA Bretagne-Atlantique, Campus Universitaire de Beaulieu , 35042 Rennes Cedex, France,
davide.frey@inria.fr

October 19, 2017

Abstract

Peer-sampling protocols constitute a fundamental mechanism for a number of large-scale distributed applications. The recent introduction of WebRTC facilitated the deployment of decentralized applications over a network of browsers. However, deploying existing peer-sampling protocols on top of WebRTC raises issues about their lack of adaptiveness to sudden bursts of popularity over a network that does not manage addressing or routing. Spray is a novel random peer-sampling protocol that dynamically, quickly, and efficiently self-adapts to the network size. Our experiments show the flexibility of Spray and highlight its efficiency improvements at the cost of small overhead. We embedded Spray in a real-time decentralized editor running in browsers and ran experiments involving up to 600 communicating web browsers. The results demonstrate that Spray significantly reduces the network traffic according to the number of participants and saves bandwidth.

1 Introduction

Peer-sampling protocols [24, 38, 39] constitute a fundamental mechanism for a number of large-scale distributed applications both on the Cloud [11] and in a peer-to-peer setting [16, 40, 43]. By providing each node with a continuously changing partial view of the network, they make applications resilient to churn [6] and inherently load balancing [15]. In the context of video streaming, for example, a peer-sampling protocol makes it possible to distribute the streaming load over all peers without requiring the creation and the maintenance of rigid structures like multiple trees [15, 29].

The recent introduction of WebRTC [3] has renewed the research interest in a variety of applications that require peer-sampling protocols such as video streaming [34, 35], content-delivery networks [44], or real-time collaborative editors [32]. However, deploying existing peer-sampling protocols on top of WebRTC raises important technical challenges. (1) WebRTC does not manage addressing nor routing; this makes connection establishment much more costly than on IP networks and more likely to fail. (2) Browsers run on desktops, laptops and mobile phones. This requires protocols that reduce resource consumption as much as possible. (3) The ability to launch WebRTC sessions through simple HTTP links exposes applications to sudden bursts of popularity. Consider the example of a user who is streaming a video directly from his mobile phone to some of his friends. The user suddenly witnesses some

dramatic event, and his friends spread the news by twitting the stream's address. Instantly a huge number of users connect and start watching the stream on their laptops and phones. The streaming mechanisms and the protocols it relies on must be able to adapt to this sudden burst of popularity, maintaining their quality of service, while being able to return to their initial configuration when the popularity burst subsides.

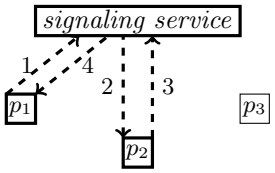
Unfortunately, existing peer-sampling protocols lack adaptiveness or reliability in the context of WebRTC. On the one hand, SCAMP [19, 20] provides adaptiveness by maintaining partial views of size $\ln(n) + k$, n being the number of nodes and k being a constant. However, SCAMP is not reliable in the context of WebRTC, for its connection establishment process is based on random walks and cannot handle the connection failures that often occur in WebRTC. On the other hand, the most popular approaches like CYCLON [39] and the whole RPS protocol family [24] are reliable in the context of WebRTC but are not adaptive: developers have to configure fixed-size views at deployment time. This forces developers to oversize partial views to handle potential bursts, either wasting resources or not provisioning for large-enough settings. A simple solution to address this problem estimates the actual number of network nodes by periodically running an aggregation protocol [31], then resizes the partial views. However, this aggregation protocol would have to run quite frequently, resulting in significant network overhead to anticipate a popularity burst that may never happen. Our approach provides adaptiveness and reliability by locally self-adjusting the size of partial views at join, leave, and shuffle times. At a marginal cost, partial view size converges towards $\ln(n)$. This not only makes our approach adaptive but also outputs the size of the network to the application level allowing applications to adapt to sudden burst in popularity at no additional cost. We demonstrate the impact of our approach on two use cases that use RPS to broadcast messages. The first one implements a live video streaming based on three-phase gossip [15, 28, 29]. In this case, adaptiveness allows message delivery to remain stable even in presence of burst in popularity. The second one implements a real-time collaborative editor. In this case, adaptiveness allows the editors to adjust the traffic to the size of the network. Some distributed hash table protocols such as D2HT [6] require both the knowledge of the size of the network to compute the view of each node, and a random peer-sampling protocol to provide connectivity in the presence of high churn. Our approach provides both at the same cost.

In this paper, we address the challenge of a dynamically adaptive peer-sampling, by introducing SPRAY, a novel random peer-sampling protocol inspired by both SCAMP [19, 20] and CYCLON [39]. SPRAY improves the state-of-the-art in several ways. (i) It dynamically adapts the neighborhood of each peer. Thus, the number of connections scales logarithmically with the network size. (ii) It only uses neighbor-to-neighbor interactions to establish connections. Thus, a node needs to rely only on a constant number of other nodes to establish a connection regardless of network size. (iii) It quickly converges to a topology with properties similar to those of random graphs. Thus, the network becomes robust to massive failures, it can quickly disseminate messages. (iv) Experiments show that SPRAY provides adaptiveness and reliability at a marginal cost. (v) Use cases demonstrate how the estimation of the network size positively impacts two RPS-based broadcast protocols either on traffic or message delivery.

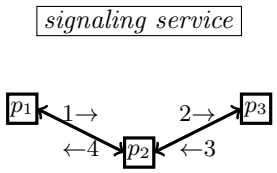
The rest of this paper is organized as follows: Section 2 reviews the related work. Section 3 states the scientific problem. Section 4 details the SPRAY protocol. Section 5 presents experimentation results of SPRAY and compares them to state-of-the-art. Section 6 details our use-cases. We conclude in Section 7.

2 Related work

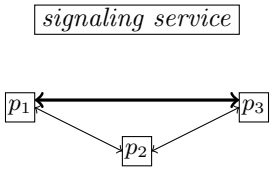
WebRTC enables real-time peer-to-peer communication between browsers even in complex network settings with firewalls, proxies or Network Address Translators (NAT). However, WebRTC manages neither addressing nor routing. To establish a connection, two browsers need to exchange offers and acknowledgments through a common mediator, e.g., mails, dedicated signaling services [2], or existing We-



(a) p_1 connects to p_2 using the signaling service. 1: p_1 pushes its offer ticket; 2: p_2 pulls the ticket; 3: p_2 pushes its response; 4: p_1: pulls the response and establishes a connection with p_2. p_3 does the same with p_2.



(b) p_1 connects to p_3 using p_2 as mediator. 1: p_1 sends its offer ticket to p_2; 2: p_2 forwards it to p_3 and registers p_1 as the emitter; 3: p_3 sends its response to p_2; 4: p_2 forwards it to the emitter p_1 which connects to p_3.



(c) The resulting network overlay: a fully connected network composed of 3 members.

Figure 1: Creating an overlay network on top of WebRTC.

bRTC connections [1]. Figure 1a describes the very first connection of one peer p_1 with another p_2 using a signaling server. Figure 1b shows instead that p_3 can later use p_2 as mediator instead of the signaling service. Figure 1c shows the resulting network. Note that if p_2 crashes or leaves during the forwarding process, the connection establishment will fail, even if alternative routes exist as WebRTC does not manage routing.

Using signaling services and existing WebRTC connections makes it easy to deploy random peer-sampling protocols [21] in browsers that can run on mobile phones or tablets connected to mobile networks [10]. However, the complexity of the WebRTC connection process further requires nodes to establish as few connections as possible in order to reduce the generated traffic and limit resource consumption. Unfortunately existing protocols fail at this task.

Random peer-sampling protocols [21, 24] produce overlay networks by populating the partial views with references to peers chosen at random among the network members following a uniform distribution. Solely relying on local knowledge, they converge to a topology exposing properties similar to those of random graphs [8, 13]. Among others, they efficiently provide connectedness and robustness. A wide variety of gossip-based protocols use random peer-sampling at their core. For instance, topology optimization protocols [23, 41] aim at improving localization, latency, or at addressing user preferences.

The representatives of random peer-sampling protocols using a fixed-size partial view comprise Lpbcast [14], HyParView [27] Newscast [38], and Cyclon [39]. The developers deploying applications using such protocols must foresee the dimensions of the networks to setup properly their partial views size. These decisions cannot be easily retracted afterwards at runtime. As a consequence, applications tend to overestimate the required view sizes.

For example, a Cyclon-based application may maintain 7 connections in the browser despite requiring only 4. While this causes little harm in standard IP-based networks. Maintaining too many active WebRTC connections may uselessly overload a device.

This calls for a dynamic peer-sampling service that can adapt to a dynamic number of participants.

A way to achieve this adaptive behavior may consist in using one of the several protocols for network-size estimation. For example, [18] samples and analyzes a subset of the network and deduces the overall network size using probabilistic functions. Sketching techniques [5] use instead hashing to represent a large number of distinct node identities and estimate the size of the network thanks to the resulting collisions in hash values. Finally, averaging techniques [22] use aggregations that converge over exchanges to a value which depends on the network size. Unfortunately, while they can be accurate in their estimations, these approaches add a communication overhead which makes them too expensive for mobile phones and similar devices.

The sole representative of adaptive-by-design random peer-sampling is SCAMP [19, 20]. Its interesting property lies in its partial view sizes that grow logarithmically with the size of the network. However, SCAMP suffers from other drawbacks. In particular, SCAMP uses random walks to establish new connections between peers. In WebRTC, each hop in these random paths must be traveled back and forth to finalize the connection establishment (see Figure 1). This drastically increases the probability that SCAMP will fail in establishing connections. In the presence of churn or message loss, the number of connections decreases quickly, eventually leading to network partitions (see Section 5.8).

3 Problem statement

We consider a set V of peers labeled from p_1 to p_N. The peers can crash or leave at any time without giving notice, and they can recover or re-enter the network freely. We consider only non-byzantine peers.

A multiset of arcs $E = (p_f, p_t) \in V \times V$ allows p_f to communicate with p_t but not the converse. The multiset of arcs is dynamic, i.e., arcs appear and disappear over time. By $\mathcal{N}^t \in V \times E$, we denote the state of the *network overlay* at time t.

The multiset of arcs starting from p_f is called the *partial view* of p_f, noted $\mathcal{P}_f$. The destination peers in this view are the *neighbors* of p_f. A partial view can contain arcs that are stale when neighbors have left or crashed.

The peers communicate by means of messages. They only send messages to their neighbors. A departed peer cannot send messages. An arc may fail to deliver a message. A stale arc systematically fails.

Building an adaptive-by-design random peer-sampling that meets WebRTC constraints raises the following scientific problem:

Problem Statement 1 *Let t be an arbitrary time frame, let V^t be the network membership at that given time t and let $\mathcal{P}_x^t$ be the partial view of peer $p_x \in \mathcal{V}^t$. An adaptive and reliable random peer-sampling should provide the following properties:*

1. *Partial view size:* $\forall p_x \in V^t, |\mathcal{P}_x^t| = \mathcal{O}(\ln |V^t|)$
2. *Connection establishment by a node:* $\mathcal{O}(1)$

The first condition states that the partial view size is relative to the size of the network at any time. It also states that partial views should grow and shrink, at worst, logarithmically compared to the size of the network. This is a condition for adaptiveness and not a requirement for network connectivity. The second condition states that a node should be able to establish a connection by relying only on a constant number of intermediary peers, independently of network size. Lpbcast, HyParView, Newscast, and CYCLON fail to meet the first condition of the problem statement since they do not adapt their views to the network size. SCAMP fails to meet the second condition of the problem statement since each connection implies an unsafe random dissemination. The next session introduces SPRAY, a protocol that meets both of these constraints without any global knowledge.

4 Spray

SPRAY is an adaptive-by-design random peer-sampling protocol inspired by SCAMP [19, 20] and CYCLON [39]. It comprises three parts representing the lifecycle of a peer in the network. First, the joining process injects a logarithmically growing number of arcs into the network. Hence, the number of arcs

scales with the network size. Next, each peer runs a periodic process in order to balance the partial views both in terms of partial view size and uniformity of the referenced peers within them. Quickly, the overlay network converges to a topology exposing properties close to those of random graphs. Finally, a peer is able to leave at any time without giving notice (equivalent to a crash) while the network properties do not degrade.

The key to adaptiveness consists in keeping a consistent global number of arcs during the whole protocol. Indeed, unlike CYCLON, SPRAY is always on the edge of a logarithmic number of arcs compared to the network size. Since SPRAY nodes never create additional arcs after joining, any removal is a definitive loss. Thus, firstly, SPRAY's joining adds some arcs to the network. Secondly, SPRAY's shuffling preserves all arcs in the network. Thirdly, SPRAY's leaving cautiously removes some arcs, ideally the number of arcs introduced by the last joining peer.

Occasionally, keeping the global number of arcs constant forces the shuffling and the leaving processes to create *duplicates* in partial views. Thus, a partial view may contain multiple occurrences of a particular neighbor. In this paper, we show that the number of duplicates remains low and does not impact network connectivity.

4.1 Joining

SPRAY's joining algorithm is the only part of the protocol where the global number of arcs in the network can increase. To meet the first requirement of the problem statement, this number of arcs must grow logarithmically compared to the network size. Therefore, each peer assumes that it has such a logarithmically growing view to propagate the identity of the joining peer. Algorithm 1 describes SPRAY's joining protocol. Line 6 shows that the contacted peer only multicasts the new identity to its neighborhood. Afterwards, to limit the risk of connection failures, each neighbor immediately adds the joining peer to its own neighborhood. This fulfills the second condition of the problem statement. In total, the number of arcs in the network increases of $1 + \ln(|V^t|)$ using only neighbor-to-neighbor interactions, which leads to a total number of $|V^t| \ln(|V^t|)$ arcs [19].

Algorithm 1 The joining protocol of SPRAY running at Peer p.

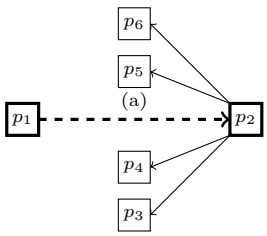
```
1: INITIALLY:
2:     P ← ∅;                          ▷ the partial view is a multiset
3:
4: EVENTS:
5:     function onSubs(o)                                   ▷ o : origin
6:         for each ⟨q, _⟩ ∈ P do sendTo(q, 'fwdSubs', o);

7:     function onFwdSubs(o)                                ▷ o : origin
8:         P ← P ⊎ {⟨o, 0⟩};
```

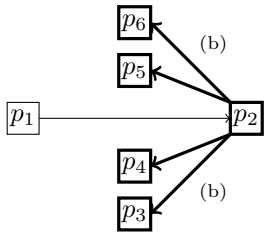
The partial view is a multiset of pairs $\langle n, age \rangle$ which associates each neighbor, n, with an age, age. The multiset allows nodes to manage duplicates, while age plays the same role as in CYCLON i.e. it accelerates the removal of crashed/departed peers by shuffling with the oldest neighbors first. The $onSubs$ event is called each time a peer joins the network. $onSubs$ forwards the identity of the joining peer to all neighbors, independently of the age. The $onFwdSubs$ event is called when a peer receives such forwarded subscription. It adds the peer as one of its neighbor with an age set to 0 meaning that it is a brand new reference.

Figure 2 depicts a joining scenario. Peer p_1 contacts p_2 to join the network composed of $\{p_2, p_3, p_4, p_5, p_6\}$. For simplicity, the figure shows only the new arcs and the neighborhoods of p_1 and p_2. Peer p_1 directly adds p_2 in its partial view. Peer p_2 forwards the identity of p_1 to its neighborhood. Each of these neighbors adds p_1 in their partial view. In total, SPRAY establishes 5 connections and the network is connected.

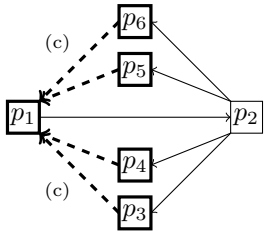
Unfortunately, the partial views of the newest peers are clearly unbalanced and violate the first condition of our problem statement. The periodic protocol described in the next section will re-balance the partial views.



(a) p_1 contacts p_2 to join the network. p_1 adds p_2 to its neighborhood. p_1 sends its request to p_2.



(b) The $onSubs(p_1)$ event is raised at p_2 which forwards the subscription to its neighbors.



(c) The $onFwdSubs(p_1)$ event is raised at p_{3-6}. The peers add p_1 to their neighborhood.

Figure 2: Example of the joining protocol of SPRAY.

4.2 Shuffling

Unlike CYCLON, SPRAY shuffles partial views of different sizes. The shuffling balances the partial view sizes and randomly mixes the neighborhoods between peers. Nevertheless, the global number of arcs in the network remains unchanged.

In SPRAY's shuffling protocol, each of the involved peers sends half of its partial view to the other. After integration, their view sizes both tend to the average of their initial sizes. The sum of their elements remains unchanged. In order to keep the arc number invariant, the partial views of SPRAY are multisets. If a peer receives an already known reference, it still stores it, yet as a duplicate. Thus, the SPRAY's shuffling protocol never increases nor decreases the arc count.

If duplicates have a negative impact on network properties, most of them disappear after shuffling and they proportionally become negligible as the network grows.

Algorithm 2 shows the SPRAY protocol running at each peer. It is divided between an active thread looping to update the partial view, and a passive thread which reacts to messages. The functions which are not explicitly defined are the following:

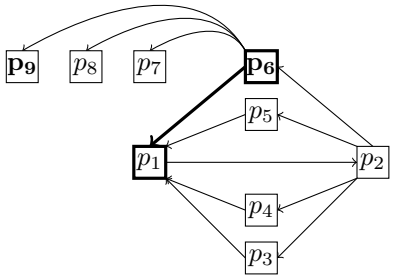
- INCREMENTAGE(*view*): increments the age of each element in the view and returns the modified view.
- GETOLDEST(*view*): retrieves the oldest peer contained in the view.
- GETSAMPLE(*view*, *size*): returns a sample of the view containing *size* elements.
- REPLACE(*view*, *old*, *new*): replaces all the occurrences of the *old* element in the view by the *new* element and returns the modified view.
- RAND(): generates a random floating-point number between 0 and 1.

In the active thread, Function LOOP is called every Δt time units. First, the function increments the age of each neighbor in $\mathcal{P}$. Then, it chooses the oldest peer q to exchange a subset of its partial view. Peer p selects a sample of its partial view, excluding one occurrence of q and including itself. The size of this sample is half of its partial view, with at least one peer: the initiating peer (see Line 5). It sends this subset to Peer q and awaits its sample. If Peer q does not send its sample in time (it may retry to tackle with message losses), Peer p considers q as crashed or departed and executes the corresponding function (see Section 4.3). The answer of q contains half of its partial view. Since peers can appear multiple times in $\mathcal{P}$, the exchanging peers may send references to the other peer, e.g., Peer p's sample can contain references to q. Such sample, without further processing, would create self-loop (q's partial view contains references to q). To alleviate this undesirable behavior, all occurrences of the other peer are replaced with the emitting peer (see Line 6, 18). Afterwards, both of them remove the sample they sent from their view and add the received sample. It is worth noting that Peer p removes at least 1 occurrence of q and Peer q adds at least 1 occurence of p, for the sake of connectedness.

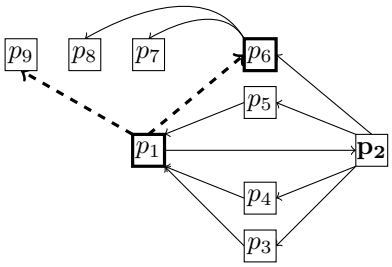
Figure 3 depicts SPRAY's shuffling procedure. This scenario follows from Figure 2: Peer p_1 just joined the network. Peer p_6 initiates an exchange with p_1 (the oldest among p_6's partial view). It randomly chooses $\lceil |\mathcal{P}_6| \div 2 - 1 \rceil = 1$ peer among its neighborhood. In this case, it picks p_9 from $\{p_7, p_8, p_9\}$. It sends the chosen peer plus its own identity to Peer p_1. In response, the latter picks $\lceil |\mathcal{P}_1| \div 2 \rceil = 1$ peer from its partial view. It sends back its sole neighbor p_2 and directly adds the received neighbor to its partial view. After receipt, Peer p_6 removes the sent neighbor from its partial view, removes an occurrence of p_1, and adds the received peer from p_1. Peers $\{p_6, p_9\}$ compose p_1's partial view. Peers $\{p_2, p_7, p_8\}$ compose that of p_6.

The example shows that, at first, the initiating peer has 4 peers in its partial view, while the receiving peer has only 1 peer. After the exchange, the former has 3 neighbors including 1 new peer. The receiving peer has 2 neighbors, and both of them are new. Thus, the periodic procedure tends to even out the partial view sizes of network members. It also scatters neighbors in order to remove the highly clustered groups which may appear because of the joining protocol.

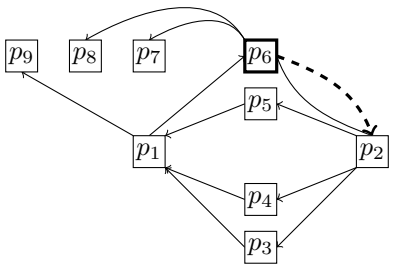
Concerning the convergence time of the shuffling algorithm, there exists a close relationship between SPRAY and the proactive aggregation protocol introduced in [22]. Given a distribution of values associ-



(a) Peer p_6 initiates the exchange with p_1 by sending to the latter the multiset $\{p_6, p_9\}$.



(b) Peer p_1 receives p_6's message. It sends back the multiset $\{p_2\}$ and adds $\{p_6, p_9\}$ to its partial view.



(c) Peer p_6 receives p_1's response, it adds $\{p_2\}$ to its partial view.

Figure 3: Example of the shuffling protocol of SPRAY.

Algorithm 2 The cyclic protocol of SPRAY running at Peer p.

1: **ACTIVE THREAD:**
2: **function** LOOP( ) ▷ Every Δt
3: $\mathcal{P} \leftarrow incrementAge(\mathcal{P})$;
4: **let** $\langle q, age \rangle \leftarrow getOldest(\mathcal{P})$;
5: **let** $sample \leftarrow getSample(\mathcal{P} \setminus \{\langle q, age \rangle\}, \lceil |\mathcal{P}| \div 2 \rceil - 1) \uplus \{\langle p, 0 \rangle\}$;
6: $sample \leftarrow replace(sample, q, p)$;
7: **let** $sample' \leftarrow sendTo(q, 'exchange', sample))$;
8: **while** $(\neg timeout(sample') \wedge \neg sample')$ **do** $waitForQ$; ▷ Awaiting q's sample
9: **if** $\neg timeout(sample')$ **then**
10: $sample' \leftarrow replace(sample', p, q)$;
11: $\mathcal{P} \leftarrow (\mathcal{P} \setminus sample) \uplus sample'$;
12: **else**
13: $onPeerDown(q)$; ▷ see Algorithm 3
14:
15: **PASSIVE THREAD:**
16: **function** ONEXCHANGE(o, $sample$) ▷ $o : origin$
17: **let** $sample' \leftarrow getSample(\mathcal{P}, \lceil |\mathcal{P}| \div 2 \rceil)$;
18: $sample' \leftarrow replace(sample', o, p)$;
19: $sendTo(o, sample')$;
20: $sample' \leftarrow replace(sample', p, o)$;
21: $\mathcal{P} \leftarrow (\mathcal{P} \setminus sample') \uplus sample$;

ated with peers, and under the assumption of a sufficiently random peer-sampling, such an aggregation protocol yields the following mean μ, and variance σ^2, at cycle i;

$$\mu_i = \frac{1}{|V^t|} \sum_{x \in V^t} a_{i,x} \qquad \sigma_i^2 = \frac{1}{|V^t|-1} \sum_{x \in V^t} (a_{i,x} - \mu_i)^2$$

where $a_{i,x}$ is the value held by Peer p_x at cycle i. The estimated variance must converge to 0 over cycles. In other terms, the values tend to be the same over cycles. In the SPRAY case, the value $a_{i,x}$ is the partial view size of Peer p_x at cycle i. Indeed, each exchange from Peer p_1 to Peer p_2 is an aggregation resulting to: $|\mathcal{P}_1| \approx |\mathcal{P}_2| \approx (|\mathcal{P}_1| + |\mathcal{P}_2|) \div 2$. Furthermore, at each cycle, each peer is involved in the exchange protocol at least once (they initiate one), and in the best case $1 + Poisson(1)$ (they initiate one and, on average, each peer receives another one). This relation being established, we know that SPRAY's partial view sizes converge exponentially fast to the global average size. Additionally, we know that each cycle decreases their variance in the overall system at a rate comprised between $1 \div 2$ and $1 \div (2\sqrt{\mathrm{e}})$.

The shuffling algorithm provides adaptiveness at the cost of duplicates. Averaging the partial view sizes over exchanges quickly converges to a network topology where the partial views are balanced.

4.3 Leaving or crashing

In SPRAY, peers can leave the network without notice. As such, we make no distinction between node departures and crashes. Peers that crash or leave the network do not save information about the network. Consequently, if they join the network afterwards, they are considered as completely new peers.

SPRAY must react properly to departures and crashes. Without reaction, the network would collapse due to an over zealous removal of arcs. Indeed, a peer joining the network injects $1 + \ln(|V^t|)$ arcs. Nevertheless, after few exchanges, its partial view becomes populated with more neighbors. Then,

if this peer leaves, it removes $\ln(|V^t|)$ arcs from its partial view, and another $\ln(|V^t|)$ arcs from peers which have this peer in their partial views. Therefore, without any crash handler, we remove $2\ln(|V^t|)$ connections instead of $1+\ln(|V^t|)$. To solve this, each peer that detects a crash may reestablish a connection with anyone in its neighborhood. The peer reestablishes a connection with probability $1-1 \div |\mathcal{P}|$. Since $|\mathcal{P}| \approx \ln(|V^t|)$ peers have the crashed peer in their partial view, it is likely that all of them will reestablish a connection, except one. Therefore, when a peer leaves, it approximately removes the number of connections injected by the most recently joined node. Duplicates created by this procedure will disappear over time as peers shuffle their partial view.

Algorithm 3 The crash/departure handler of SPRAY running at Peer p.

```
1:  function onPeerDown(q)    ▷ q: crashed/departed peer
2:      let occ ← 0;
3:      for each ⟨n, age⟩ ∈ P do    ▷ remove and count
4:          if (n = q) then
5:              P ← P \ {⟨n, age⟩};
6:              occ ← occ + 1;
7:      for i ← 0 to occ do ▷ probabilistically duplicates
8:          if (rand() > 1 ÷ (|P| + occ)) then
9:              let ⟨n, _⟩ ← P[⌊rand() * |P|⌋];
10:             P ← P ⊎ {⟨n, 0⟩};

11: function onArcDown(q, age)    ▷ q: arrival of the arc down
12:     P ← P \ {⟨q, age⟩};
13:     let ⟨n, _⟩ ← P[⌊rand() * |P|⌋];
14:     P ← P ⊎ {⟨n, 0⟩};    ▷ systematically duplicates
```

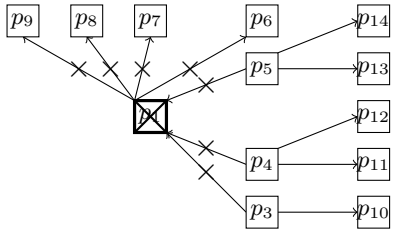
Algorithm 3 shows the manner in which SPRAY deals with departures and crashes. Function onPeerDown shows the reaction of SPRAY when peer q is detected as crashed or departed. A first loop counts the occurrences of this neighbor in the partial view, and removes all of them. Then, the second loop probabilistically duplicates the reference of known peers. The probability depends on the partial view size before the removals.

Figure 4 depicts SPRAY's crash/leaving handler. The scenario follows from prior examples after few other exchanges. Peer p_1 leaves the network without giving notice. With it, 7 connections are down. Peers p_3, p_4, and p_5 have the crashed/left peer in their partial view. Peer p_5 has $1 - 1 \div |\mathcal{P}_5| = 2 \div 3$ chance to replace the dead connections. In this case, it duplicates the connection to p_{13}. Identically, p_3 and p_4 detect the crash/leaving and run the appropriate operation. Only p_3 duplicates one of its connections. In total, 5 connections have been removed.

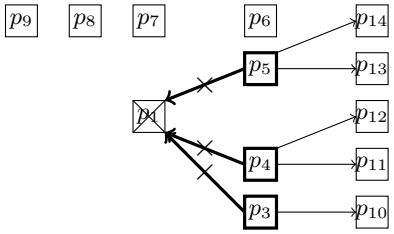
The example shows that some peers reestablish connections if they detect dead ones. The probability depends on the partial view size of each of these peers. On average, one of these peers will likely remove the arc while the other peers will duplicate one of their existing arcs. In this case, Peer p_1 injected 5 connections when it joined. It removes $7 - 2 = 5$ connections when it leaves. The global number of connections remains logarithmic with respect to the number peers in the network. Nevertheless, we observe that connectedness is not entirely guaranteed – only with the high probability implied by random graphs. Indeed, if Peer p_1 is the sole bridge between two clusters, adding arcs is not enough to ensure connectedness.

Algorithm 3 also shows that SPRAY distinguishes peer crashes and arc crashes. Indeed, Function onArcDown deals with connection establishment failures. In this function, the failing arc is systematically replaced with a duplicate. Therefore, the arc count stays invariant even in the presence of connection establishment failures. The distinction between the functions onPeerDown and onArcDown is necessary because the former is supposed to remove a small arc quantity over departures, contrarily to the latter. Without this small removal, the global arc count would grow unbounded with network turnover.

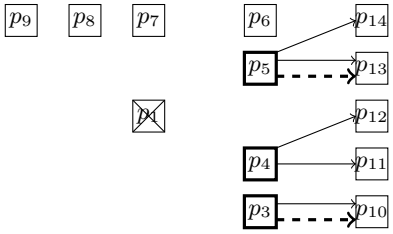
In the context of WebRTC, SPRAY calls the onArcDown function when a connection establishment fails. It may happen due to network misconfiguration or to message losses. SPRAY calls the onPeerDown function when the connection was established once but the neighbor is no longer responding.



(a) Peer p_1 crashes. A lot of connections are down.



(b) The peers p_{3-5} notice that they cannot reach p_1 anymore.



(c) The peers p_3 and p_5 create a duplicate of one of their existing neighbor.

Figure 4: Example of the crash/leaving handler of Spray.

5 Experimentation

In this section, we evaluate how the adaptiveness of Spray impacts common metrics of peer-sampling performance including clustering coefficient, average shortest path length, in-degree distribution, robustness, and arc count. We compare Spray with a representative of fixed-size partial view approaches, namely Cyclon. We expect Spray and Cyclon to exhibit similar behaviors when Cyclon is configured in advance to the logarithm of the targeted network size. We expect Spray to save resources when Cyclon provides larger views, and to be more robust when Cyclon provides smaller views. We expect Spray-based network size estimators to provide an estimate at least accurate enough to inform about the order of magnitude of the actual network size. Finally, we expect Spray to keep a negligible number of duplicates in its partial views. We also evaluate the impact of the WebRTC connection establishment on Cyclon, Scamp, and Spray, in networks subject to message loss. We expect a normal behavior for Cyclon and Spray while Scamp becomes quickly partitioned.

The experiments run on the PeerSim simulator [30], a well-known program written in Java that allows simulations to reach high scale in terms of number of peers. Our implementation of the evaluated random peer-sampling protocols is available on the Github platform[1].

5.1 Clustering and convergence time

Objective: To observe how adaptiveness impacts clustering and convergence time.

Description: The average local clustering coefficient [42] measures peers' neighborhood connectivity in the network:

$$\overline{C} = \frac{1}{|V^t|} \sum_{x \in V^t} C_x$$

where C_x is the local clustering coefficient of Peer p_x. The higher the coefficient, the more peers are

[1] `https://github.com/justayak/peersim-spray`

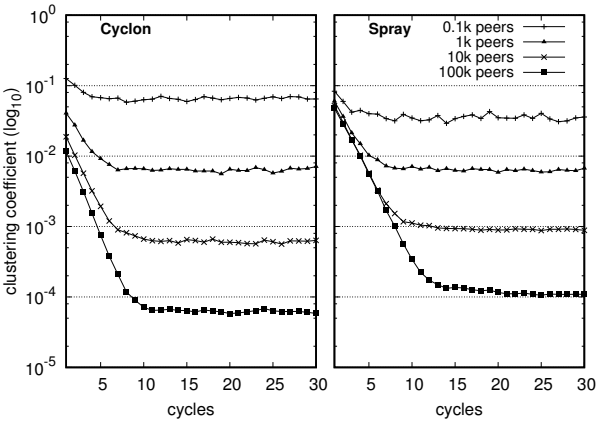


Figure 5: Clustering coefficient.

tied together. The coefficient of complete graphs is 1. The coefficient of random graphs is close to 0.

For each approach, the experiment comprises 4 runs respectively building a network of 0.1k, 1k, 10k, and 100k peers. Peers join the network at once. Peers join the network through a contact peer chosen at random following a uniform distribution.

For the sake of fairness, we configure the representative of fixed-size approaches, CYCLON, to provide 7 neighbors which is close to the number of neighbors provided by SPRAY when the network size reaches 1k peers ($\ln(1000) \approx 7$). Compared to SPRAY, CYCLON is oversized for 0.1k peers and undersized for 10k peers and 100k peers. During exchanges, the peers using CYCLON shuffle 3 out of their 7 neighbors.

Results: Figure 5 shows that CYCLON starts with a lower clustering coefficient than SPRAY. Yet, CYCLON and SPRAY have roughly similar convergence time, for it converges exponentially fast. Figure 5 also shows that both approaches converge to a low clustering coefficient which is characteristic of random graphs. Nevertheless, CYCLON and SPRAY do not reach the same values after convergence. Except when the network comprises 1k peers, SPRAY's values are either below when CYCLON is comparatively oversized; or above when CYCLON is comparatively undersized.

These results show that both SPRAY and CYCLON can be deployed at once. They start ready-to-use but do not have their optimal properties. However, no matter the size of the deployment, they will obtain these properties very quickly.

Reasons: Peers join one after the other in 1 round. Peers join using a contact peer chosen at random among already connected peers. Therefore, old peers are chosen more often during this starting round. In the starting topology, old peers are more clustered than newest ones. In particular, since SPRAY does not enforce a limit on the size of partial views, oldest peers have larger partial views than newest peers. Thus, oldest peers are more tied together. Consequently, SPRAY starts with a higher clustering coefficient than CYCLON.

The clustering coefficient measures how much peer's neighbors are connected together. It directly depends on the partial view size of each peer which, in CYCLON, is constant. Thus, when the number of peers is multiplied by 10, the clustering coefficient after convergence is divided by 10. On the other hand, peers using SPRAY have a partial view the size of which reflects the network size. When the network has 1k peers, SPRAY and CYCLON have roughly the same average partial view size (SPRAY 7.1 vs CYCLON 7), hence almost identical clustering coefficient measurements. By extending the reasoning, this also explains why SPRAY yields lower $\overline{C}$ values when CYCLON is comparatively oversized, and why it yields higher values when CYCLON is comparatively undersized.

5.2 Information dissemination

Objective: To observe how adaptiveness impacts the average shortest path length, i.e., the speed of information dissemination.

Description: The average path length is the average of the shortest path length between peers in the graph. It counts the minimum number of hops to reach a peer from another given peer. It basically represents the traveling time of any information to reach all the peers at least once. We average the path length on a subset of the network membership. We run the simulation on SPRAY 100 times to avoid any side effects due to randomness. The experiment comprises 3 runs for CYCLON: we set CYCLON to

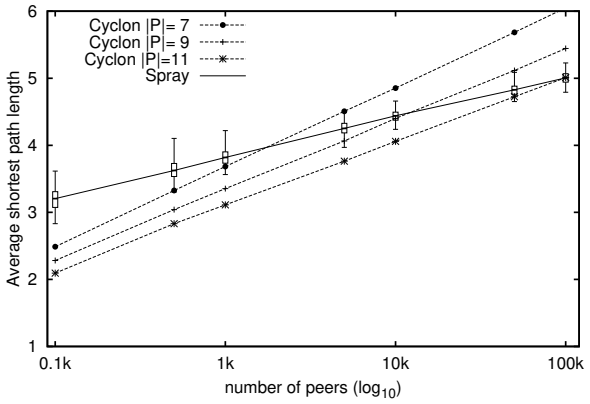


Figure 6: Average shortest path length.

provide 7 neighbors which is close to the number of neighbors provided by SPRAY when the network reaches 1k peers; we set CYCLON to provide 9 neighbors which is close to the number of neighbors provided by SPRAY when the network reaches 8k peers; we set CYCLON to provide 11 neighbors which is close to the number of neighbors provided by SPRAY when the network reaches 60k peers. We perform the measurements after convergence. The checkpoints for the measurements are 0.1k, 0.5k, 1k, 5k, 10k, 50k, and 100k peers.

Results: Figure 6 shows that both CYCLON and SPRAY have a relatively small average shortest path length. Figure 6 also shows that each run of CYCLON is divided in three parts compared to SPRAY. CYCLON starts with smaller values than SPRAY. SPRAY and CYCLON become equivalent when the latter is configured to the logarithm of the actual network size. After this point, SPRAY has smaller values than CYCLON. Overall, SPRAY scales better than CYCLON since the gradient (slope) of the former is lower than any configuration of the latter. Empirically, we observe that CYCLON builds networks with an average shortest path length of $\frac{\ln(|V^t|)}{\ln(|P|)}$ where $|P|$ is constant, while SPRAY builds networks with an average shortest path length of $\frac{\ln(|V^t|)}{\ln(\ln(|V^t|))}$.
These results show that developers could use both SPRAY and CYCLON to build efficient information dissemination protocols. Nevertheless, they should use SPRAY over CYCLON if their concerns are about latency and the network size is unknown or can change over time.

Reasons: We performed the measurements after convergence. After convergence, the network overlay become closely related to random graphs. In particular, the diameter and average shortest path length remain small.
Before reaching the point where the number of neighbors provided by SPRAY is equivalent to the number of neighbors provided by CYCLON, CYCLON provides more neighbors than SPRAY. Since the number of arcs is greater, it yields a lower average path length. Conversely, when the number of peers in the network goes over the point where SPRAY and CYCLON are equivalent, SPRAY provides more neighbors than CYCLON. Therefore, the lower average path length is smaller using SPRAY. Overall, SPRAY scales better than any configuration of CYCLON, for it adds arcs as the network grows.

5.3 Load-balancing

Objective: To observe how adaptiveness impacts the in-degree distribution, i.e., the load-balancing among peers.

Description: The in-degree of a peer shows how well it is represented in others' partial views. The in-degree distribution of the network can highlight the existence of weakly connected peers and strongly connected hubs. These peers have an in-degree far from the average in-degree value. It directly impacts robustness, for few crashes or departures may disconnect a weakly connected peer; and disconnections of hubs deeply impact the rest of the network.
For each approach, the experiment comprises 4 runs respectively building a network of 0.1k, 1k, 100k, and 500k peers. We configure CYCLON with partial views of size 7 which is close to the number of neighbors provided by SPRAY when the network reaches 1k peers. We perform the in-degree measurements after convergence.

Results: Figure 7 shows the in-degree distribution of CYCLON and SPRAY. The x-axis denotes the in-degree. The y-axis denotes the percentage of peers with such in-degree in the network. The top part of

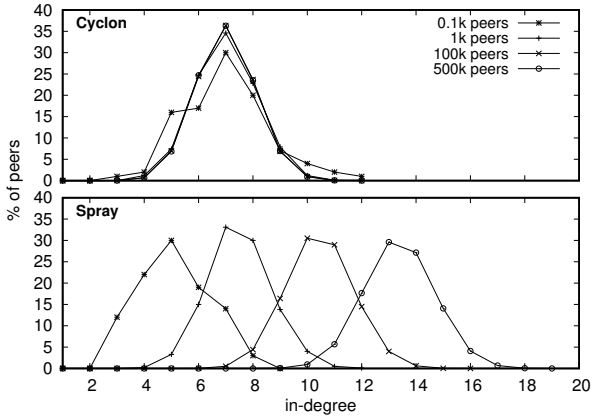


Figure 7: In-degree distribution.

Figure 7 focuses on CYCLON. We observe that the in-degree distributions are identical independently of the network size. For instance, the distribution of 0.1k peers is identical to the one of 500k peers. The mean value is roughly 7 and we observe a strong peak on this value. On the other hand, the bottom part of Figure 7 focuses on SPRAY. We observe that the distribution of SPRAY follows the average partial view size, which itself follows the growth of the size of the network.

Figure 7 also shows that peers are very gathered around the mean partial view size for both SPRAY and CYCLON. For instance, for the run with 500k peers using SPRAY, the mean in-degree value is 13.37 and 88% of peers have an in-degree between 12 and 14 included. Even the highest in-degree value 18 is not far from the average in-degree. Thus, there are no weakly connected peers nor strongly connected hubs. These results show that both SPRAY and CYCLON are resilient to random failures, for peers are equally important in terms of connectedness. These results also show that developers could use both SPRAY and CYCLON to build dissemination protocols that would balance the download among peers. Using SPRAY, the download would increase and decrease following the fluctuations in network size.

Reasons: Once configured, CYCLON must handle any number of peers in the network with a fixed-size partial view and since the partial view size is constant, the in-degree of peers stays stable. On the other hand, in SPRAY, each joining peer brings an increasing number of arcs in the network. Thus, the in-degree of peers slowly grows reflecting the network size. Hence, the distribution in the bottom part of Figure 7 shifts slowly to higher in-degree values as the network size grows.

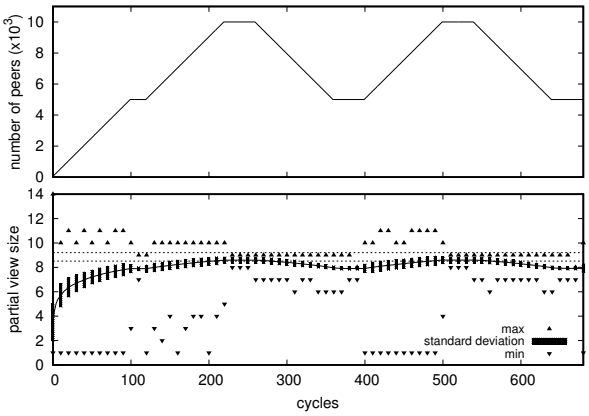
SPRAY does not peak on a particular value as CYCLON does because the average partial view size for a particular network size may fall in-between integer values. For instance, if peers using SPRAY have partial views that contain 6.5 neighbors on average, it means that half these peers have 6 neighbors while the other half have 7 (out-degree). As consequence, the average in-degree value will also be 6.5. However, while SPRAY and CYCLON constrain the size of partial views respectively to the average and to a constant value, they do not constrain the size of in-views, hence the distributions of Figure 7.

5.4 Adaptive partial views

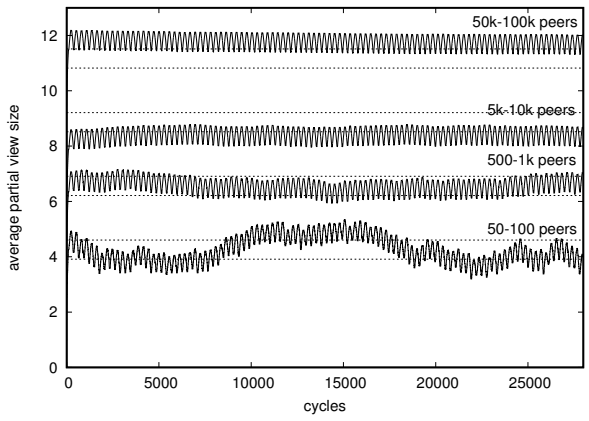
Objective: To show that using SPRAY, the partial views grow and shrink logarithmically compared to the network size.

Description: In this experiment we focus on dynamic networks where peers can join and leave. In this experiment, we create 4 networks that vary from 50 to 100 peers, from 500 to 1k peers, from 5k to 10k peers, and 50k to 100k peers. First, we create a network of half the maximum targeted size. Second, we repeatedly inject then disconnect half the maximum targeted size in periods of 240 cycles including a 40 cycles break in between injection and disconnection phases. For instance, in the network the size of which varies from 50k to 100k, 500 peers join the network at each cycle during the injection phases until it reaches 100k peers; then 40 cycles last; then 500 peers leave the network at each cycle during the disconnection phases until it reaches 50k peers, and so on. During the experiments, we measure the maximum, the minimum, the average, and the standard deviation of the size of partial views.

Results: Figure 8a focuses on the first 680 rounds of the experiment about the network varying from 5k to 10k peers. The top part of the figure shows the



(a) Mininum, maximum, average, and standard deviation of the size of partial views.



(b) Average size of partial views over long period of time.

Figure 8: Partial view size measurements in networks subject to churn.

number of peer over time. The bottom part of the figure shows the measurements made on partial views. We observe that (i) partial views grow logarithmically when the network size increases; (ii) partial views shrink logarithmically when the network size shrinks; (iii) while the extreme values of partial view size are spread during joining phases (from 1 to 12), standard deviation remains small, meaning that most of peers have the same partial view size. Figure 8b shows the results of the 4 runs over 28k cycles, i.e., 100 injection/removal phases. The x-axis denotes the time in cycles. The y-axis denotes the average partial view size of peers. We observe that (i) the average partial view size does not necessarily fit exactly the theoretical expectation; (ii) the average partial view size varies even for a network reaching the same number of peers; (iii) variations are more important as the number of peers is lower.

These results show that developers could use SPRAY to deploy protocols such as distributed hash tables [9] or routing protocols [26] that could benefit from random peer-sampling protocols properties and that normally require the use of network-size estimators to work.

Reasons: A peer joining the network injects at least one arc plus a number of arcs depending on the contact peer. Since we choose the contact peer at random, and the partial views are averaged over shuffles, this number of injected arcs is the average partial view size on average. It leads to a logarithmic progression of the number of arcs. However, the peer joining the network only have one peer in its partial view while adding more arcs to its contact's neighborhood. Since we choose the contact peer at random, it may be chosen multiple times during 1 cycle which increases quickly its neighbors' number of arcs above the average. Consequently, we observe a large difference between the minimum and the maximum, but the standard deviation remains low. The latter is even more lowered after shuffling, for it makes partial views converge to their average size.

When a peer leaves the network, there is a logarithmic number of peers that detect the departure. They probabilistically duplicate an arc in such manner that one of them do not. Overall, it removes a logarithmic number of arcs plus one which correspond to the number of arcs injected during the latest joining. Consequently, the partial views shrink logarithmically when the network size diminishes.

If we choose multiple time a same contact peer, its neighbors' number of arcs increases above the average. If one of these neighbors is chosen as contact peer before any shufflings, the global number of arcs increases too quickly. On the other hand, we may choose the newest peers as contact node hence increasing the number of arc too slowly, for they have fewer neighbors. Depending on these choices, the average partial view size can grow too high or too low compared to the theoretical value. Yet, it remains logarithmic, for these choices are made at random following a uniform distribution. These choices are more influential when the network size is low, for any addition or removal of arcs is proportionally more important.

5.5 Network-size estimation

Objective: To show the accuracy of SPRAY's partial views when used as a network-size estimator.

Description: To evaluate the accuracy of SPRAY as a network-size estimator, we define estimation quality as the ratio of measured estimations to the actual network size. This allows us to see how far from the actual network size the estimation can be. In our experiment, peers use SPRAY and join the network by groups of 1k peers until the network reaches a size of 100k peers. We evaluate two estimators based on the partial views of SPRAY. The first estimator uses the local partial view only. SPRAY provides a peer x with a view of size $|P_x^t| \approx \ln(|V^t|)$. To estimate the network size, peer x processes:

$$\widehat{V^t} = \exp\left(|P_x^t|\right)$$

The second estimator uses the local partial view and asks for the neighbors' partial view size. Peer x averages the values and computes the estimate:

$$\widehat{V^t} = \exp\left(\frac{|P_x^t| + \sum_{i \in P_x^t} |P_i^t|}{|P_x^t| + 1}\right)$$

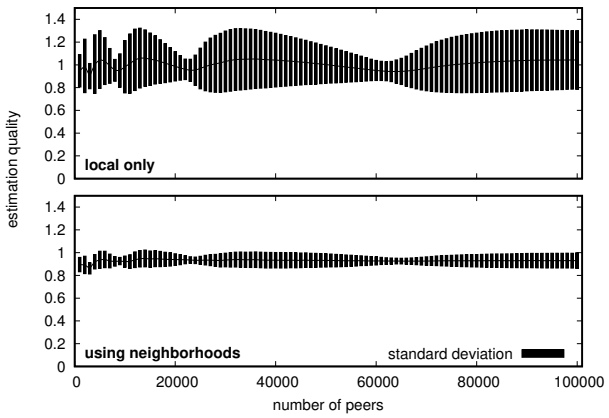


Figure 9: Quality of network-size estimation.

Results: Figure 9 shows mean and standard deviation of the estimation quality with increasing network sizes. We observe that the estimation quality depends on the network size. Using local knowledge only, most peers are able to compute an estimation ranging from $\pm 10\%$ to $\pm 30\%$ of the actual network size. We also observe that increasing the available knowledge using peers' neighborhoods improves the estimation by decreasing the variation of estimates around the mean (from $\pm 5\%$ to $\pm 10\%$).

These results show empirically that applications that need to estimate the network size can use SPRAY's partial view size as input to know the order of magnitude of the actual network size. The use of local knowledge only leads to an inaccurate estimate but it comes at no cost while full-fledged network-size estimators generate traffic [31]. Applications can easily increase accuracy on-demand by asking their neighbors for their partial view sizes for a limited cost.

Reasons: The standard deviation of the estimation quality grows and shrinks while the network size increases, particularly in the case of local-only knowledge. Figure 9 shows that the variance of estimates narrows before quickly raising at increasingly sized intervals. Since the size of views is an integer value, the logarithm of the estimation for a peer at a given time can be off by 1. For instance, when the network reaches 65k peers, the average partial view size just starts to go above 11, meaning that some peers have 12 neighbors in their partial views. Since the local-knowledge estimation of a peer x is $\widehat{V}_x^t = \exp(|P_x^t|)$, increasing peer x's view size by 1 results in a much larger estimate. Hence the sudden increase in standard deviation at such network sizes. The standard deviation decreases as the actual network size approaches the overestimated size.

This effect can be mitigated by averaging the local value with the neighbors' partial view sizes: the computed value is no longer an integer but an aggregation of representative values chosen at random.

5.6 Robustness

Objective: To show SPRAY's robustness to failures.

Description: To evaluate robustness, we count the number of weakly and strongly connected components of the network. Counting strong components allows us to estimate the effectiveness of information dissemination protocols built on top of random peer-sampling protocols. For example, with two strong components, bidirectional communication between the two components cannot be achieved. The first strong component may be able to reach the second, but the second may not reach the first. However, the network remains in a repairable state, for at least one link units these components. After some shuffling, the two strong components add links to each other. Thus, the two strong components merge into one. Therefore, counting the weak components allows us to estimate to what extent the network can be repaired, i.e., two weak components exist when there is no link to unit them.

In our experiment, we configure CYCLON's view size to 9 which is approximately equivalent to SPRAY's view size when the network reaches 8k peers. The network contains 10k members. We perform the removals after the approaches have converged to a stable overlay network. We remove a random set of peers at once following a uniform distribution. We remove from 25 to 95 percent of peers every 5 percents, i.e., 16 runs for each approach. We perform a last measurement at 99 percent. We measure the number of components immediately after each removal.

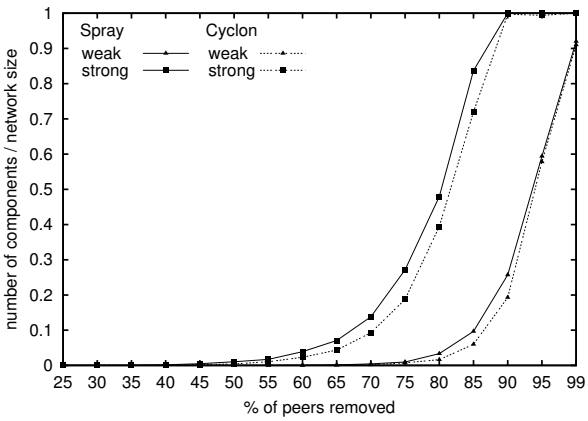


Figure 10: Robustness to massive failures.

Results: Figure 10 shows the robustness of Spray and Cyclon to massive random failures. The x-axis denotes the percentage of peers removed at once. The y-axis denotes the ratio of strong/weak components over the network size after removals. First, Figure 10 shows that both the random peer-sampling protocols Spray and Cyclon suffer from deteriorated behavior at high removal percentages, Cyclon being slightly better in this term. Figure 10 shows that the number of strong components starts to increase at 45 percents and to quickly increase at 70 percents. Hence, the information dissemination starts to degrade at 45 percents of removals: messages broadcast by some peers cannot reach all peers. Fortunately, Figure 10 also shows that the approaches are able to recover from dire state until high removal ratio. Indeed, the number of weak components starts to increase at 70 percents of removals, meaning that there is no link between some parts of the network to unit them. In other terms, some parts of the network became completely disjoint. Shuffling cannot be initiated between members of each other disjoint parts. Consequently, the network cannot fully repair itself above 70 percents of removals.
These results show that both Spray and Cyclon fit contexts such as the internet where numerous peers can join and leave freely in short periods of time.

Reasons: The random peer-sampling approaches Cyclon and Spray yield very similar results because the number of neighbors provided by both approaches is very close when the network reaches 10k. Yet Cyclon performs slightly better for two reasons. First the variance in the degree distribution of Spray causes Cyclon to have more arcs in this specific experiment. Second, Spray wastes some of its neighbors due to the presence of duplicate entries in its views (yet the number of duplicates remains small – see Figure 11). Still both protocols preserve the ability to disseminate information until very large removal percentages. Another way to interpret this result consists in observing that when all peers have similar degrees, (see Figure 7), removing a particular peer does not greatly affect connectedness. As suggested above, the direction of arcs impacts more information dissemination than the peer-sampling protocol itself. If an arc constitutes the last link between two clusters of the network, the messages from one of these clusters cannot reach the other one. Yet, this arc is enough for the peer-sampling protocol to start shuffling the views, ultimately populating them with members from both cluster. Hence, the network repairs itself. When clusters become fully disjoints, neither Cyclon nor Spray is able to repair the network.

5.7 Duplicates

Objective: To show that a small proportion of peers contains duplicates in their partial view.

Description: Using Spray as random peer-sampling protocol, we measure the amount of peers which have a partial view containing at least one duplicated reference. We perform the measurements on networks containing 0.1k, 1k, 10k, 100k, and 500k peers. We measure the number of duplicates after convergence. We put this in relation with a theoretical approximation from the birthday paradox where the view size would be the number of randomly chosen people and the identity of peers would be the birthdates. The probability of a peer to *not* have duplicates is approximately:

$$-\exp\left(\frac{-\ln(|V^t|) * (\ln(|V^t|) - 1)}{2 * |V^t|}\right)$$

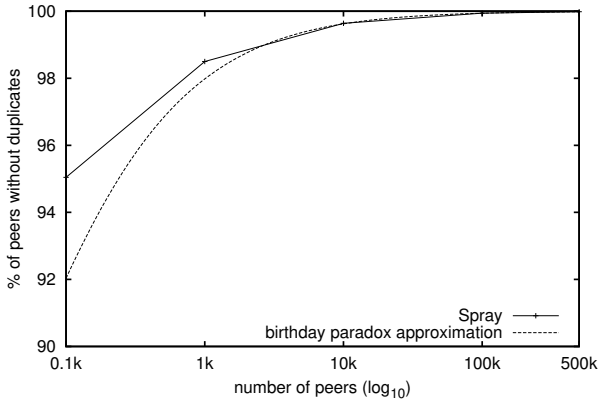


Figure 11: Duplicates in networks of different size.

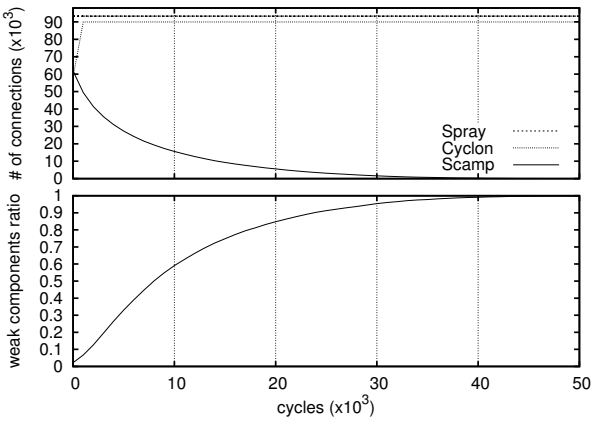


Figure 12: Number of arcs and partitioning in networks subject to failures in the connection establishments.

Results: Figure 11 shows the proportion of peers using a partial view containing duplicates. We observe that there always exist partial views with at least one duplicate. The proportion is more important when the network size is small (e.g. 5 percents for 0.1k peers). It becomes a minor overhead when the network size is larger (e.g. less than 1 percent for 10k peers). The birthday paradox approximation seems to follow very closely the experimental results. It empirically indicates that there exists a relation between the duplicates and the birthday paradox. The proportion of peers without duplicates tends to 100 percents as the network size grows.

Reasons: While the number of peers in the network grows linearly, the neighborhood size of each peer only grows logarithmically. This significant difference between the growths leads to the fact that the chances of a particular peer to have at least twice the reference to another peer becomes smaller as the number of peers in the network increases.

5.8 Failures in connection establishment

Objective: To show that Spray does not suffer from failures in connection establishments, contrarily to Scamp.

Description: We measure both the arc count and the number of weak components in the network. The simulations involve Cyclon configured with partial views containing 9 neighbors, Scamp[2], and Spray. They run over 50k cycles. The network initially contains 10k members. To establish a connection, we use the WebRTC three-way handshake, i.e., the initial peer emits an offer ticket, the arrival peer stamps the ticket, the initial peer finalizes the connection using the stamped ticket (see Section 2). The probability that the ticket fails to traverse a hop is set to 10^{-3}.

Results: The top part of Figure 12 shows the arc count of the random peer-sampling protocols while the bottom part of Figure 12 shows the weak components of the network. As expected, we observe that the arc counts of Cyclon and Spray stays constant over cycles: 90k and 93k arcs for Cyclon and Spray respectively. On the other hand, Scamp suffers from failures in connection establishment. This directly impacts network connectedness as measured by the weak-component ratio. The network of Scamp quickly degrades.
These results show that both Spray and Cyclon fit the constraints set by the connection establishement protocol of WebRTC.

Reasons: The arc counts of Cyclon and Spray

[2]A modified version of Scamp whose periodic protocol works properly when there is no connection failures. Available at `https://github.com/justayak/peersim-spray`

remains constant over time but for different reasons. In Cyclon, the shuffling protocol makes sure that the partial view is filled to its maximum. When a peer removes a broken connection, it replaces it with a fresh one in the following round. In Spray, when the protocol tries to use a broken connection, it replaces it with another known one. Thus, the arc count stays constant and the shuffling protocol makes sure that duplicates disappear over time (the arc moves to another peer where it is not a duplicate). Scamp, however, does not establish new connections with the neighbors of its neighbors. Each hop of connection establishment process is an opportunity for failure. Let P_f be the probability that an element of the dissemination path (either a peer or a connection) crashes or leaves during a hop of the three-way handshake, without any possible recovery. Let P_E be the probability that a connection establishment cannot be completed. Without three-way handshake, P_E is straightforward:

$$P_{E,\,1way}^{Scamp} = 1 - (1 - P_f)^{k+1} \tag{1}$$

This corresponds to the probability that each element (arc and peer) in the path of size $k+1$ stays alive during the random walk. In the context of WebRTC, the offer ticket must travel back to its emitter. As a consequence, the elements of the random walk cannot fail until the stamped ticket travels back. We obtain:

$$\begin{aligned} P_{E,\,3way}^{Scamp} &= 1 - ((1 - P_f)^{2(k+1)}(1 - P_f)^{2k} \dots (1 - P_f)^2) \\ &= 1 - (1 - P_f)^{k^2+3k+2} \end{aligned} \tag{2}$$

In other terms, the first chosen arc and peer in the path must stay alive $2k+2$ hops, the second chosen arc and peer must stay alive $2k$ hops and so on. This long duration leads to a quicker degeneration of the connection count.

6 Use case: broadcasting messages

In this section, we highlight how protocols built on top of Spray can benefit from its adaptiveness. We focus on broadcasting messages. Firstly, we consider the case of video streaming using a three-phase gossip protocol [28, 15, 29]. Using Spray, we expect a stable message-delivery rate even when the size of the network fluctuates. Using Cyclon, we expect degraded performance when the size of the network is larger than expected at configuration time. Secondly, we use the case of *gossiping* in decentralized real-time collaborative editing. Using Spray, we expect the generated traffic to scale logarithmically with respect to the size of the network.

The first experiment runs on the PeerSim simulator [30] and the code is available on the Github platform[3]. The second experiment is a simulation involving up to 600 interconnected Web browsers on the Grid'5000 testbed. The code is available on the Github platform[4].

6.1 Streaming

Live video streaming has grown in importance over recent years and companies offering peer-to-peer streaming gather large numbers of viewers. For example, HiveStreaming[5] declared having a customer network of 400k peers [35]. The uploading bitrate required to serve such a large number of users would lead to huge operational costs, which can instead be saved through direct peer-to-peer exchanges.

6.1.1 Operation

A number of existing systems [16, 29, 45] use some variant of gossip to achieve effective stream dissemination in the presence of churn. In the following, we consider the three-phase variant adopted by [15, 28, 29, 45]. Nodes gossip advertisement messages for available packets to *fanout* other nodes, which then pull the packets they need. Algorithm 4 details the operation of this protocol. First, with a time interval of δ ($200ms$ in [15, 16]), each peer advertises the packets it can serve to f (fanout) neighbors. Second, a peer receiving such an advertisement requests the packets it needs. Third, the advertising peer sends the requested packets.

[3] `https://github.com/justayak/peersim-spray`
[4] `https://github.com/Chat-Wane/CRATE`
[5] `https://www.hivestreaming.com/`

In their analysis of this three-phase model, the authors of [15] observed that, in bandwidth-constrained scenarios, peers should vary their communication partners as often as possible in order to equalize everyone's contribution. There are two ways to achieve this in a large scale setting: (i) refreshing the view of the peer-sampling protocol (e.g. CYCLON, or SPRAY) before sending each advertisement packet, or (ii) using a view that is much larger than the gossip fanout and refreshing it less often. Due to the relatively small size of view-exchange messages, solution (ii) achieves the best trade-off, while solution (i) is extremely impractical and waste a significant amount of bandwidth just for packet headers—[15, 16] have each node send 5 advertisement packets per second.

For this reason, [17] follows solution (ii) and sets the fanout to $\ln(N) + c$ (according to theory [25]), N being the number of participating peers, and c being a positive constant.

6.1.2 Experimentation

Objective: To show how broadcast protocols can benefit from an adaptive random peer sampling protocol.

Description: In this experiment, we reproduce the first gossip phase of the protocol in [15, 17, 29] and model an application that streams live content to 100 peers. We configure CYCLON's partial view to 6 times the threshold fanout required for 100 nodes ($6 \ln(100) \approx 6 \cdot 5 = 30$, analogously to [17]), and consider two configurations for the broadcast fanout $\ln(100) + 1 \approx 6$ and $\ln(100) + 3 \approx 8$, which provide high reliability with 100 nodes. Similarly, we configure SPRAY to have partial views size equal to $6 \cdot \ln(|V^t|)$—instead of adding only 1 arc targeting its contact (see Section 4.1), the newcomer adds 6 arcs—and sets the fanout to one sixth of the view size plus 1 or 3. This gives both protocols the same configuration for 100 peers. We then consider network sizes growing from 100 to 2000 peers, and measure the ratio of fully delivered messages to 1k sent messages, i.e., we evaluate the fraction of all messages that reach all the members of the network.

Results: Figure 13 shows the results of this experiment. First, we observe that the ratio of fully delivered messages using CYCLON suffers from a steady decrease. The highest fanout leads to better results. In contrast, SPRAY remains stable despite the presence of small jumps. A fanout set to $\ln(|V^t|) + 1$ gives a full delivery ratio above 90%. A fanout set to $\ln(|V^t|) + 3$ is very close to 100%. Overall, the broadcast mechanism benefits from the adaptive nature of SPRAY's partial view, which allows it to scale according to the size of the network.

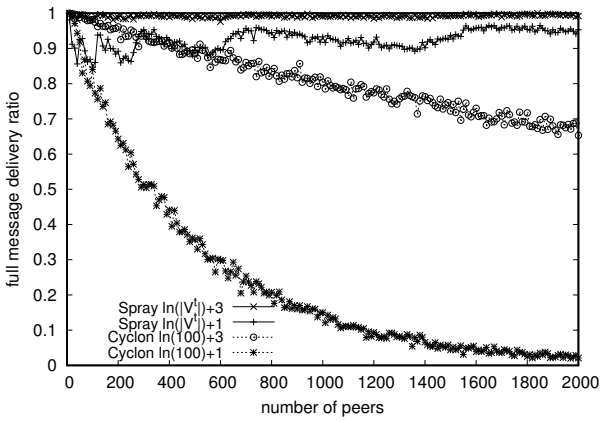


Figure 13: Ratio of messages fully delivered.

Reasons: To ensure a fair use of the bandwidth contributed by peers, we configure the view size to be larger than the fanout. This allows stream packets in three-phase gossip to follow "almost" random subtrees of the complete overlay. But it also leads the first gossip phase to require a fanout of at least $\ln(N) + c$ [25], as the random sub-graphs of the overlay grafted in the first phase may fail to reach all nodes if the fanout is too low. In our case, the broadcast protocol built on top of CYCLON has a constant fanout set for a specific network size. As long as the network is smaller than this value, the full delivery ratio stays high. However, it quickly decreases with larger network sizes. SPRAY instead allows the application fanout to follow the growth of the partial view size, which scales logarithmically compared to the network size. This allows the streaming application to adapt to changes in the size of the network. As a result, the full delivery ratio provided by the broadcast mechanism on top of SPRAY remains sta-

Algorithm 4 Three-phase gossip.

```
1: INITIALLY:
2:     B                                                          ▷ buffer of packets
3:     b                                                          ▷ number of advertised packets
4:     f                                                          ▷ fanout f ≤ |P|
5:
6: ACTIVE THREAD:
7:     function ADVERTISEMENTLOOP(( ))                            ▷ every δ time
8:         let advertiseTo ← getPeers(P, f);                      ▷ f distinct random peers from P
9:         let advertisement ← getIdentifiers(B, b);              ▷ b distinct random packet Ids
10:        for each q ∈ advertiseTo do sendTo(q, 'advertisement', advertisement);
11:
12: PASSIVE THREAD:
13:    function ONADVERTISEMENT(o, ads)                           ▷ o : advertiser
14:        for each (id ∈ ads) do
15:            if id ∉ B then sendTo(o, 'request', id);
16:
17:    function ONREQUEST(o, id)                                  ▷ o : requester
18:        let packet ← getMessage(B, id);                        ▷ get the requested packet from the buffer
19:        sendTo(o, packet);                                     ▷ finally send the packet
```

ble. Since the values manipulated by peers—partial view size and fanout—are integers, the measurements make small jumps when a rounded value increases enough to be incremented.

Objective: To show how the full delivery ratio behaves during a buzz, i.e., a sudden massive increasing of network size shortly followed by departures.

Description: We configure SPRAY and CYCLON like in the previous experiment. For CYCLON, we use a partial view size of 30 and two broadcast fanout values: $\ln(100) + 1$ and $\ln(100) + 3$. For SPRAY, we use a dynamic view of $6 \cdot \ln(|V^t|)$ and broadcast fanout values of $\ln(|V^t|) + 1$ and $\ln(|V^t|) + 3$. The simulation starts with 100 peers. The network quickly reaches 10k peers during the popularity burst. Then the network size decreases to 3k members. In this experiment, we measure the full delivery ratio over each group of 100 consecutive messages.

Results: Figure 14 shows the results of this experiment. The top part of the figure shows the evolution of the network size over time. The bottom figure shows the full delivery ratio over time. Like

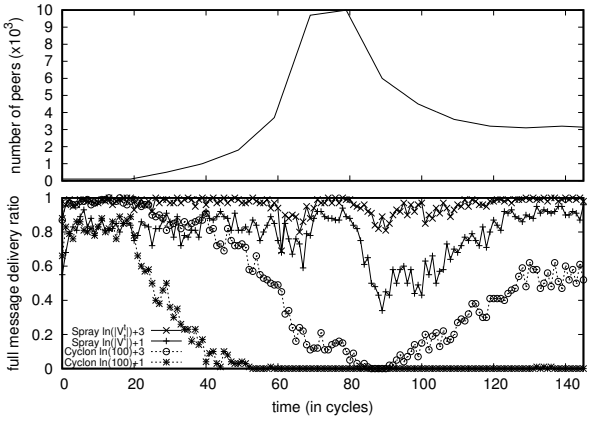


Figure 14: Full message delivery ratio during a buzz.

in the previous experiment, Figure 14 shows that using Cyclon and a predefined fanout works well until the network size reaches an unexpected size. During the peak, the delivery ratio falls quickly. Conversely, the broadcast mechanism built on top of Spray automatically adapts the fanout to the size of the network. Thus, it does not experience performance losses while a large number of nodes suddenly join, even though we observe a decrease in delivery ratio during the departures of peers, in particular with a fanout of $\ln(|V|) + 1$. This demonstrates that, in the context of three-phase gossip, Spray makes broadcast more resilient to quick membership changes.

Reasons: The fanout of configurations involving Cyclon is constant. When the number of peers in the network exceeds the expectations, the delivery ratio quickly degenerates. Concerning Spray, the fanout follows the evolution of the network thanks to adaptive partial views. Both Spray and Cyclon detect departing nodes during the shuffling phase. The associated delay leads to the use of some stale arcs from partial views, which explains the temporary decrease in delivery ratio during the shrinking phase. Since both Cyclon and Spray clean their partial views over time, the delivery ratio recovers its expected value.

6.2 Real-time editing

Crate is a real-time editor that allows authors to write anytime and anywhere, whatever the number of participants, without any third party [32]. Compared to trending Cloud-based editors such as Google Docs, it alleviates privacy, scalability, and single-point-of-failure issues while remaining easy to use. It can be used for small groups but also during events such as massive online lectures, TV shows, or conferences that gather larger groups. For instance, online lecture sessions reach thousands of students. Transitions between small groups and large groups is supported transparently thanks to Spray. Distributed real-time editing is a pertinent context for Spray for group sizes differ depending on the document, and change quickly over time.

Crate builds a network of browsers where each browser is able to communicate with a logarithmically scaling number of browsers compared to the global network size. Each change performed on documents transits through neighbors and reaches all members in a scalable way [7]. Unlike the state-of-the-art [38, 39], Spray allows the diffusion cost to adapt to the network size without any probing mechanism or any central authority. Without Spray, developers would have to overestimate the parameters of peer-sampling protocols to get a system that behaves well for large as well as for small networks. This would increase the cost of maintaining connections alive (e.g. in a WebRTC context) and more importantly the cost of application protocols (e.g. broadcasting protocols). Spray, on the other hand, ensures that small networks do not pay the price of large networks without requiring the intervention of developers or end users.

6.2.1 Operation

To provide availability and responsiveness in documents, collaborative editors consider multiple authors, each hosting a replica of their shared document [36]. On updates, the local replica is directly modified. Then, each update is propagated to all the editing session where it is integrated. Strong eventual consistency [4] states that a system is correct if and only if the replicas converge to an equivalent state when they integrated the same updates. In other terms, the users read identical documents.

Crate uses a distributed sequence data type [37] which provides two commutative operations: the insertion and the removal of an element. Commutativity allow users to avoid the difficult, time consuming, and error-prone task of solving conflicts due to concurrent edits. However, commutativity comes at the price of metadata attached to each character. These metadata (referred to as *identifiers*) are unique and immutable. The structure representing an identifier is a list $[\ell_1.\ell_2 \dots \ell_k]$ the size k of which is determined during allocation. Crate uses an allocator function [33] to keep these identifiers under acceptable growth, i.e., a polylogarithmic upper bound on their space complexity compared to the document size: $\mathcal{O}((\log |document|)^2)$.

An editor sends each pair of identifier and character

to all members of the editing session. For this purpose, CRATE uses a very simple broadcast protocol built on top of SPRAY following the principles of epidemic dissemination (also known as gossiping) [12]. Such mechanisms make extensive use of partial views to efficiently disseminate messages.

Algorithm 5 gossip.

```
1: function BROADCAST(m)                          ▷ m : message
2:     for each ⟨q, _⟩ ∈ 𝒫 do
       sendTo(q, 'broadcast', m);

3: function RECEIVEBROADCAST(m)                   ▷ m : message
4:     if ¬alreadyReceived(m) then
5:         deliver(m);
6:         broadcast(m);                          ▷ rebroadcast
```

Algorithm 5 shows the instructions of this protocol. When a peer emits a message, it sends it to its neighborhood. Each peer receiving such messages for the first time forwards it to its neighborhood too. Messages transitively reach all network members.

Since the gossiping algorithm depends of the neighborhood provided by SPRAY, and since the latter grows logarithmically compared to the network size, the communication complexity of an application is upper-bounded by $\mathcal{O}(m \ln |V^t|)$, where m is the space complexity of a message. Since CRATE's identifiers are sublinearly upper-bounded compared to the document size, CRATE scales well.

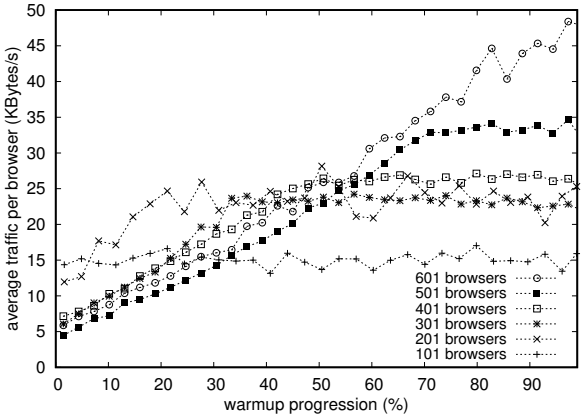
6.2.2 Experimentation

We highlight the improvements brought to protocols built on top of SPRAY on a real-life use case about decentralized collaborative editing in Web browsers.

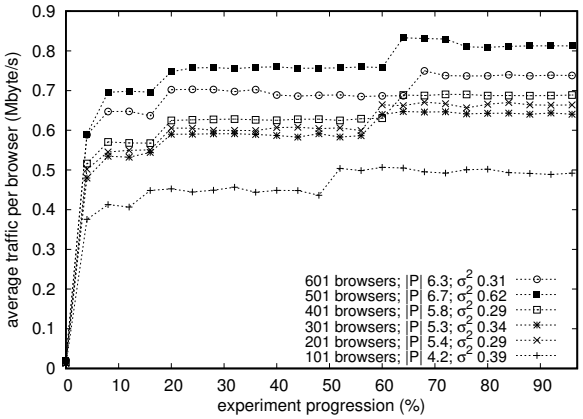
Objective: To show that the traffic generated by SPRAY stays low compared to the traffic generated by broadcasting. To show the influence of SPRAY's adaptiveness over the traffic generated by decentralized editors running on a network of browsers.

Description: Experiments run on Grid'5000 where machines host 5 browsers each. Browsers open CRATE and connect to an editing session through a signaling server. Runs comprise from 101 browsers to 601 browsers with 100 browsers increments, i.e., 6 different runs. The first editor creates the editing session which is progressively joined by the other writers (1 joiner per 5 seconds). Each member starts sharing the access to the editing session as soon as it joins it. Outsiders join the network through one of them chosen at random. Once all peers have joined the editing session, the document editing starts. The insertion rate is 100 insertions per second uniformly distributed among peers. Each experiment runs during 8 hours of which 7 hours are dedicated to editing. The document size reaches millions of characters.

Results: Figure 15a and Figure 15b show the average traffic per second generated by members involved in collaborative editing. The x-axis denote the time progression of the warmup and the experiment in percentage over 45 minutes and 7 hours respectively. The y-axis denote the average traffic generated by Web browsers. The y-axis scale of Figure 15a is in KByte while the y-axis scale of Figure 15b is in MByte. The legend shows the average and variance of partial view size associated with each run. Figure 15a shows that the traffic generated by the random peer sampling protocol SPRAY increases as peers join the editing session over time. As expected, the more peers in the network the more traffic is generated. Nevertheless, it stays an order of magnitude below the traffic generated by broadcasting. In Figure 15b, the height of plots corresponds to the multiplicative factor coming from the messages dissemination. As expected, this factor grows logarithmically regarding the network size. Thus, 101 browsers have an average traffic lower than 601 browsers because their partial views are smaller in average. On the opposite, using CYCLON, the traffic would have been the same for all runs. Since it commonly overestimates partial views to accommodate with any network size, the traffic would have been higher. It is important since even small partial view size differences significantly impact traffic. Figure 15b also shows that the average partial view size follows the natural logarithmic expectation. Yet, the run involving 501 browsers has a slightly higher average partial view size than the run involving 601 browsers. Because the joining part of SPRAY establishes a number of WebRTC connections depending on the first contact member, there are variations between independent runs. Still,



(a) Warmup time: peers join the session. Only SPRAY generates the outgoing traffic.



(b) Editing time: peers write a document. Both SPRAY and CRATE generate the outgoing traffic.

Figure 15: Average traffic per second generated by a real application.

Spray scales logarithmically overall. Figure 15b finally shows that the variance of partial views σ^2—displayed in the legend—stays small, which indicates that the network reached a state where neighborhood sizes are balanced, hence, where the load is balanced.

Reasons: Random peer sampling protocols generate little traffic, for they only exchange small messages every period of time. The measurements of Figure 15a also take into account the traffic generated by WebRTC connections that constantly send few bytes to check if the connection is alive, i.e., heartbeat messages. Adding peers to the editing session increases partial view sizes, hence, the generated traffic. Broadcast protocols use neighborhoods to disseminate messages. Each member receives and forwards each operation which transitively reaches all members. Thus, the traffic depends on messages size multiplied by neighborhoods size logarithmically scaling thanks to Spray. The growth during each run corresponds to the polylogarithmic growth of identifiers from the editors. Since the document size increases over time, the LSeq's identifiers grow accordingly which impacts on messages size [33].

7 Conclusion and perspectives

In this paper, we described Spray, an adaptive-by-design random peer-sampling approach designed to fit WebRTC's constraints. Spray provides: (i) logarithmically growing partial views reflecting the global network size, (ii) constant time complexity on connection establishments using solely neighbor-to-neighbor interactions. Our experiments demonstrate how Spray's adaptiveness improves the performance of random peer-sampling when the size of the network changes. In particular, the average shortest path length scales better and the in-degree evolves with the network size. Adaptiveness comes at the price of duplicates in the partial views. However, our simulations supported by theoretical analysis show that the number of duplicates remains very low and becomes negligible in large networks.

To highlight the improvements brought by Spray, we demonstrated how a broadcast protocol can take advantage of its adaptiveness to adjust the fanout and handle a sudden burst of popularity. We also built Crate, a decentralized collaborative editor directly accessible in web browsers with a full implementation of Spray on top of WebRTC. We launched experiments involving up to 600 browsers showing the benefits of adaptive neighborhoods.

Spray makes building scalable decentralized applications in browsers easy and accessible. Developers do not require to foresee the network size targeted by their applications. Just the same as for broadcast, many topology optimization protocols (e.g. about geolocalization, latency, or preferences) rely on random peer-sampling protocol such as the generic algorithms T-Man [23] and Vicinity [41]. Spray allows such protocols to benefit from its self-adjusting partial views. Protocols such as D2HT [6] require both an estimator of the network size and a random peer-sampling protocol. Since Spray provides both at same cost, a Spray-based D2HT would be an interesting perspective.

Peer-sampling protocols build overlay networks. These overlay networks do not necessarily reflect the underlying topology made of routers. Using WebRTC, a connection establishment may fail depending on network configurations. Section 4.3 states that such failures should be handled by replacing failed links by duplicates to known peers. As future work, it would be interesting to see if this strategy leads to an overlay network that self-adapts to the real underlying topology.

Acknowledgments

This work was partially funded by the French ANR project SocioPlug (ANR-13-INFR-0003), and by the DeSceNt project granted by the Labex CominLabs excellence laboratory (ANR-10-LABX-07-01).

Experiments presented in this paper were carried out using the Grid'5000 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`).

References

[1] P. http://ozan.io/p/.

[2] Peerjs. http://peerjs.com.

[3] Webrtc. `http://www.webrtc.org`.

[4] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Queue*, 11(3):20:20–20:32, Mar. 2013.

[5] C. Baquero, P. Almeida, R. Menezes, and P. Jesus. Extrema propagation: Fast distributed estimation of sums and network sizes. *IEEE Transactions on Parallel and Distributed Systems*, 23(4):668–675, April 2012.

[6] M. Bertier, F. Bonnet, A. M. Kermarrec, V. Leroy, S. Peri, and M. Raynal. D2ht: The best of both worlds, integrating rps and dht. In *Dependable Computing Conference (EDCC), 2010 European*, pages 135–144, April 2010.

[7] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

[8] F. Bonnet, F. Tronel, and S. Voulgaris. *Brief Announcement: Performance Analysis of Cyclon, an Inexpensive Membership Management for Unstructured P2P Overlays*, pages 560–562. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[9] G. Camarillo and J. Maenpaa. Self-tuning distributed hash table (dht) for resource location and discovery (reload). RFC 7363, Ericsson, September 2014.

[10] R. Carvajal-Gómez, D. Frey, M. Simonin, and A.-M. Kermarrec. *Web Information Systems Engineering – WISE 2015: 16th International Conference, Miami, FL, USA, November 1-3, 2015, Proceedings, Part II*, chapter WebGC Gossiping on Browsers Without a Server, pages 332–336. Springer International Publishing, Cham, 2015.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

[12] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987.

[13] P. Erdős and A. Rényi. On random graphs i. *Publ. Math. Debrecen*, 6:290–297, 1959.

[14] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems (TOCS)*, 21(4):341–374, Nov. 2003.

[15] D. Frey, R. Guerraoui, A. Kermarrec, M. Monod, and V. Quéma. Stretching Gossip with Live Streaming. In *DSN*, 2009.

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

[17] D. Frey, R. Guerraoui, A.-M. Kermarrec, and M. Monod. Live Streaming with Gossip. Research report, Inria Rennes Bretagne Atlantique ; RR-9039, Mar. 2017.

[18] A. Ganesh, A.-M. Kermarrec, E. Le Merrer, and L. Massoulié. Peer counting and sampling in overlay networks based on random walks. *Distributed Computing*, 20(4):267–278, 2007.

[19] A. Ganesh, A.-M. Kermarrec, and L. Massoulié. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In J. Crowcroft and M. Hofmann, editors, *Networked Group Communication*, volume 2233 of *Lecture Notes in Computer Science*, pages 44–55. Springer Berlin Heidelberg, 2001.

[20] A. Ganesh, A.-M. Kermarrec, and L. Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2):139–149, Feb 2003.

[21] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In H.-A. Jacobsen, editor, *Middleware 2004*, volume 3231 of *Lecture Notes in Computer Science*, pages 79–98. Springer-Verlag, 2004.

[22] M. Jelasity and A. Montresor. Epidemic-style proactive aggregation in large overlay networks. In *Proceedings. 24th International Conference on Distributed Computing Systems.*, pages 102–109, 2004.

[23] M. Jelasity, A. Montresor, and O. Babaoglu. T-man: Gossip-based fast overlay topology construction. *Computer Networks*, 53(13):2321 – 2339, 2009. Gossiping in Distributed Systems.

[24] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8, 2007.

[25] A.-M. Kermarrec, L. Massoulié, and A. Ganesh. Probabilistic Reliable Dissemination in Large-Scale Systems. *TPDS*, 14(3):248–258, 2003.

[26] J. Kleinberg. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 163–170, New York, NY, USA, 2000. ACM.

[27] J. Leitão, J. Pereira, and L. Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP Intern ational Conference on*, pages 419–429, June 2007.

[28] H. Li, A. Clement, M. Marchetti, M. Kapritsos, L. Robinson, L. Alvisi, and M. Dahlin. FlightPath: Obedience vs. Choice in Cooperative Services. In *OSDI*, 2008.

[29] M. Monod. *Live Streaming with Gossip*. PhD thesis, IC, Lausanne, 2010.

[30] A. Montresor and M. Jelasity. Peersim: A scalable P2P simulator. In *Proceedings of the 9th International Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA, Sept. 2009.

[31] A. Montresor, M. Jelasity, and O. Babaoglu. Robust aggregation protocols for large-scale overlay networks. In *International Conference on Dependable Systems and Networks*, pages 19–28, June 2004.

[32] B. Nédelec, P. Molli, and A. Mostefaoui. Crate: Writing stories together with our browsers. In *Proceedings of the 25th International Conference Companion on World Wide Web*, WWW '16 Companion, pages 231–234, Republic and Canton of Geneva, Switzerland, 2016. International World Wide Web Conferences Steering Committee.

[33] B. Nédelec, P. Molli, A. Mostéfaoui, and E. Desmontils. LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing. In ACM, editor, *13th ACM Symposium on Document Engineering*, Sept. 2013.

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

[35] R. Roverso, R. Reale, S. El-Ansary, and S. 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.

[36] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, Mar. 2005.

[37] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types.

Stabilization, Safety, and Security of Distributed Systems, pages 386–400, 2011.

[38] N. Tölgyesi and M. Jelasity. Adaptive peer sampling with newscast. In H. Sips, D. Epema, and H.-X. Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 523–534. Springer Berlin Heidelberg, 2009.

[39] S. Voulgaris, D. Gavidia, and M. van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005.

[40] S. Voulgaris, E. Rivière, A.-M. Kermarrec, and M. Van Steen. Sub-2-Sub: Self-Organizing Content-Based Publish and Subscribe for Dynamic and Large Scale Collaborative Networks. Research Report RR-5772, INRIA, 2005.

[41] S. Voulgaris and M. van Steen. Epidemic-style management of semantic overlays for content-based searching. In J. Cunha and P. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 1143–1152. Springer Berlin Heidelberg, 2005.

[42] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 06 1998.

[43] F. Wuhib, M. Dam, R. Stadler, and A. Clem. Robust monitoring of network-wide aggregates through gossiping. *Network and Service Management, IEEE Transactions on*, 6(2):95–109, 2009.

[44] L. Zhang, F. Zhou, A. Mislove, and R. Sundaram. Maygh: Building a cdn from client web browsers. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 281–294, New York, NY, USA, 2013. ACM.

[45] M. Zhang, Q. Zhang, L. Sun, and S. Yang. Understanding the Power of Pull-Based Streaming Protocol: Can We Do Better? *JSAC*, 25(9):1678–1694, 2007.