Through the Wormhole:
Low Cost, Fresh Peer Sampling for the Internet

Roberto Roverso
Peerialism AB
Email: roberto@peerialism.com

Jim Dowling
KTH - Royal Institute of Technology, and
Swedish Institute of Computer Science
Email: jdowling@kth.se

Mark Jelasity
University of Szeged, and
Hungarian Academy of Sciences
Email: jelasity@inf.u-szeged.hu

Abstract—State of the art gossip protocols for the Internet are based on the assumption that connection establishment between peers comes at negligible cost. Our experience with commercially deployed P2P systems has shown that this cost is much higher than generally assumed. As such, peer sampling services often cannot provide fresh samples because the service would require too high a connection establishment rate. In this paper, we present the wormhole-based peer sampling service (WPSS). WPSS overcomes the limitations of existing protocols by executing short random walks over a stable topology and by using shortcuts (wormholes), thus limiting the rate of connection establishments and guaranteeing freshness of samples, respectively. We show that our approach can decrease the connection establishment rate by one order of magnitude compared to the state of the art while providing the same levels of freshness of samples. This, without sacrificing the desirable properties of a PSS for the Internet, such as robustness to churn and NAT-friendliness. We support our claims with a thorough measurement study in our deployed commercial system as well as in simulation.

Keywords-P2P Networks; Peer Sampling; NAT-resilient gossip protocols;

I. INTRODUCTION

A peer sampling service (PSS) provides nodes in a distributed system with a uniform random sample of live nodes from all nodes in the system, where the sample size is typically much smaller than the system size. PSSes are widely used by peer-to-peer (P2P) applications to periodically discover new peers in a system and to calculate system statistics. A PSS can be implemented as a centralized service [1], using gossip protocols [2] or random walks [3]. Gossip-based PSSes have been the most widely adopted solution, as centralized PSSes are expensive to run reliably, and random walks are only suitable for stable networks, i.e. with very low levels of churn [3].

Classical gossip-based PSSes [2] assume that all nodes can communicate directly with one another, but these protocols break down on the open Internet [4], where a large majority of nodes do not support direct connectivity as they reside behind Network Address Translation Gateways (NATs) and firewalls. To overcome this problem, a new class of NAT-aware gossip-based PSSes have appeared that are able to generate uniformly random node samples even for systems with a high percentage of *private nodes*, that is, nodes that reside behind a NAT and/or firewall [4]–[6].

State of the art NAT-aware gossip protocols, such as Gozar [4] and Croupier [5], require peers to frequently establish network connections and exchange messages with *public nodes*, nodes that support direct connectivity, in order to build a view of the overlay network. These designs are based on two assumptions: i) connection establishment from a private to a public peer comes at negligible cost, and ii) the connection setup time is short and predictable. However, these assumptions do not hold for many classes of P2P applications. In particular, in commercial P2P applications such as Spotify [1], P2P-Skype [7], and Google's WebRTC [8] establishing a connection is a relatively complex and costly procedure. This is primarily because security is a concern. All new connections require peers to authenticate the other party with a trusted source, typically a secure server, and to setup an encrypted channel. Another reason is that establishing a new connection may involve coordination by a helper service, for instance, to work around connectivity limitations that are not captured by NAT detection algorithms or that are caused by faulty network configurations. To put this in the perspective of our P2P live streaming system [9], which we believe is representative of many commercial P2P systems, all these factors combined produce connection setup times which can range from few tenths of a second up to a few seconds, depending on network latencies, congestion, and the complexity of the connection establishment procedure. In addition to these factors, public nodes are vulnerable to denial-of-service attacks, as there exists an upper bound on the rate of new connections that peers are able to establish in a certain period of time.

It is, therefore, preferable in our application to build a PSS over a more stable topology than the constantly changing topologies built by continuous gossiping exchanges. An instance of such a stable topology might be an overlay network where connections between peers are maintained over time and node degree is kept constant by replacing failed connections. Random walks (RWs) over such a stable overlay are potentially an alternative to NAT-resilient gossip protocols. However, we are not aware of any work which addresses the problem of random walks over the Internet in presence of NATs. That said, RW methods over other stable networks that are not the Internet [10] are not practical in our case because of the high level of churn experienced in P2P systems, which causes the PSS' quality of service to degrade by interrupting or delaying the RWs. Furthermore, RWs are not able to provide fresh-enough samples because a random walk has to complete a large number of hops (depending on the topology of the network) to collect or deposit a sample.

In this paper, we present an alternative approach where the PSS creates enough new connections to ensure the PSS' quality of service in the face of peer churn, but not too many as to exceed peers' upper bounds on connection establishment rate. We call our system a *wormhole*-based PSS (WPSS).

WPSS can tune the number of new stable network connections established per sample to match application-level requirements and constraints. We show that our system can provide the same level of freshness of samples as the state of the art in NAT-aware PSSes [5] but with a connection establishment rate that is one order of magnitude lower. This, without sacrificing the desirable properties of a PSS for the Internet, such as robustness to churn, NAT-friendliness and local randomness of samples.

The main idea behind WPSS is that we separate the service into two layers. The bottom layer consists of a stable base overlay network that should be NAT-friendly, with private nodes connecting to public nodes, while public nodes connect to one another. On top of this overlay, every node periodically connects to a random public node selected from the base overlay (not necessarily a neighbor in the base overlay). Using the random public node, each node systematically places samples of itself on nodes in the neighborhood of this random public node. We call these links to random public nodes *wormholes*. That is, a wormhole is a link to a public node that is selected uniformly and independently at random. We do not require hole-punching or relaying to private nodes in this paper, although those techniques can be used as well if there are not enough public nodes.

In addition to explaining the WPSS algorithm (Section IV), our contributions also include a analytical comparison between WPSS and related work (Section V), and a thorough evaluation of the protocol in both simulation and our deployed system (Section VI). The latter experiments include a comparison with the state of the art NAT-resilient PSS, Croupier.

II. System Model

We model a distributed system as a network of autonomous nodes that exchange messages and execute the same protocol. Nodes join and leave the system continuously. We consider large-scale networked systems with limited connectivity, for example, where a large majority of nodes reside behind NAT devices or firewalls. A public node has a globally reachable IP address; this includes nodes that use IP addresses allocated by UPnP Internet Gateway Devices. A private node does not support direct connectivity, typically, because it is behind a firewall or a NAT.

Each node discovers its NAT type (public or private) at bootstrap-time and also when its IP address changes using a NAT-type identification protocol. We also assume that a bootstrap service provides newly joined nodes with a small number of node descriptors for live nodes in the system. Each node separately maintains open network connections to a small, bounded number of randomly selected public nodes in a *stable* base overlay. The node degree on the base overlay is kept constant over time by replacing connections to failed nodes with new ones.

III. Background and Related Work

The first generation of peer sampling services were not designed to account for NATs [2], [11], [12]. In recent years, researchers have worked on NAT-aware gossip protocols [13], and the first NAT-aware PSSes were Nylon [6] and Gozar [4]. They enabled gossiping with a private node by relaying a message via an existing node in the system that had already successfully communicated with the private node. Nylon routes

packets to a private node using routing tables maintained at all nodes in the system. In contrast, Gozar routes packets to a private node using an existing public node in the system. Gozar does not require routing tables as the address of a private node includes the addresses of j public nodes that can act as relays to it. To improve Gozar's reliability, gossip messages can be sent to a private node in parallel via its j relays, where j is a system parameter. Parallelizing relay messages also reduces the latency of gossip messages, but at the cost of an increase in protocol overhead. Both Nylon and Gozar require that private nodes refresh their NAT bindings by periodically pinging their neighbors. Croupier [5] provided an alternative NAT-aware PSS that removed the need for relaying gossip messages to private nodes. Instead of routing gossip messages to private nodes, gossip requests are only sent to public nodes that act as croupiers, shuffling node descriptors on behalf of both public and private nodes. Public nodes are able to send response messages through a private node's NAT, as the shuffle request from the private node created a NAT binding rule that is subsequently used to forward the response to the private node. Two similarities between Croupier and WPSS are that nodes need to know whether they are public or private, and that the stable topology created by WPSS is similar to the dynamic topology maintained by Croupier, where both private and public nodes only connect to public nodes. In contrast to WPSS, Croupier provides a decentralized NAT-type identification protocol to discover a node's NAT type (public or private), while our system provides an infrastructure-supported service based on [14].

One general difference between gossip-based approaches and WPSS is that a gossip message combines both the advertisement of the source node's descriptor as well as a set of other node descriptors for dissemination [2], [11], while WPSS messages only contain an advertisement of the initiating node's descriptor, making the message size somewhat smaller.

PSSes have also been implemented using independent RWs on stable, and even very slowly changing, dynamic graphs [3]. Intuitively, RWs work by repeatedly injecting randomness at each step until the initiator node is forgotten. RW sampling can be classified as either *push-based* or *pull-based*. In push-based RWs, the initiating node advertises its descriptor as a random sample at the terminating node, where the RW completes. In pull-based RWs, the final node's descriptor is advertised as a random sample at the initiating node. An extension to these methods allows a new direct link (DL) to be created by the final node to fetch (pull) or send (push) a fresher sample. Depending on the network topology, RWs may require a large number of hops over unreliable links with highly varying latencies before they reach good mixing and complete, thus samples can be relatively old on arrival at final nodes. Direct links provide more recent samples at the cost of creating a new network connection. However, on the open Internet, DLs will not work if the initiator is a private node and there is no support for NAT traversal. To the best of our knowledge, there has been no previous work on making RW-based PSSes NAT-aware.

IV. Wormhole Peer Sampling Service

The core of our idea is that nodes disseminate (push) advertisements of themselves over a stable base overlay using
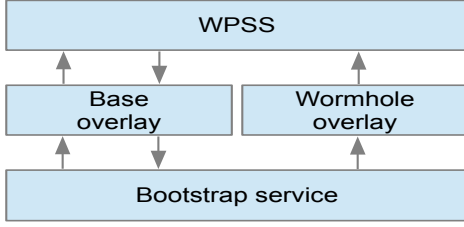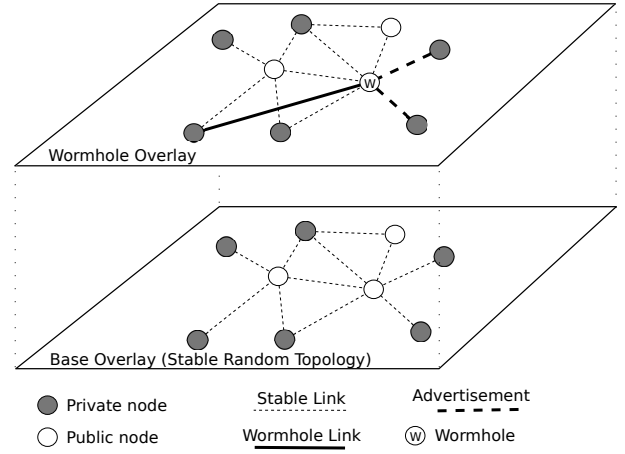
Fig. 1: WPSS layered architecture.

Fig. 2: The Base Overlay contains stable links between nodes (bottom layer). Wormholes (the thick line on the upper layer) are created to public nodes. The upper layer also illustrates a private node placing two advertisements at neighboring nodes to the wormhole.

short walks, that also utilize *wormhole* links. That is, the node that initiates the advertisement sends its own descriptor over a small number of hops in the base overlay and places it at the node where this (typically short) walk terminates. The short length of the walks guarantees the freshness of the samples. However, to be able to provide good random samples, despite completing only a relatively few hops, our advertisements always traverse a wormhole link.

A wormhole link (or wormhole, for short) points to a public node in the system that is selected independently at random. In our WPSS protocol, every node discovers such a random public node to act as its wormhole, and every node only has one wormhole active at any given point in time. The distribution of the wormholes does not need to be uniform in the system, for example, we restrict the wormholes to be public nodes. However, all the nodes must sample their wormhole from the same distribution independently. This guarantees that for each node it is true that any other node will pick it as a wormhole with the same probability.

A new network connection will be created only when a wormhole is traversed for the first time by an advertisement. The wormhole is then reused for a few subsequent advertisements from the same initiator node, in which case no new network connection needs to be established. This makes it possible to decrease the number of new links we establish.

The very first time an advertisement traverses the wormhole, it can be considered to have reached a random public node. Thus, the advertisement can now be placed at the public node as a new sample. However, if the wormhole has already been used, or the public node already has a sample from the initiator node, then the advertisement will start a random walk over the base overlay until it either (1) reaches a node that does not already have a sample from the initiator node or (2) it reaches a given time-to-live (TTL) that guarantees good quality sampling. Clearly, as a wormhole creates a link to a public node, advertisements through it will place a sample first at that public node unless it already has an advertisement from the initiator. However, the reuse of the wormhole causes advertisements to continue, allowing them to also finish at private nodes, as private nodes are connected to public nodes over the base overlay.

When a wormhole is reused by an advertisement, the expected number of hops the advertisement will have to take increases, as it needs to reach a node that does not already have a sample from the initiator node. To counteract this, new wormholes are created. WPSS defines a wormhole renewal period as a parameter for creating new wormholes enabling users to control how frequently new network connections will be created. If a new wormhole is created for each

advertisement, then we get a protocol very similar to RW-push-DL (see Section V). If wormholes are never updated, the behavior converges to that of RW-push, eventually. In between, there is a range of interesting protocols some of which—as we will argue—achieve the goals we set.

We illustrate the interaction between the wormholes and the base overlay in Figure 2. In the figure, the bottom layer is the base overlay, while the upper layer shows the wormhole overlay, containing a single wormhole. The upper layer also shows two advertisements over the wormhole that follow links in the base overlay to place samples in the vicinity of the wormhole.

A. WPSS Architecture

We implemented WPSS in a modular, layered architecture, illustrated in Figure 1. WPSS requires a stable base overlay and a set of wormhole links. It is very important that the wormhole links are drawn independently from an identical (but not necessarily uniform) distribution. Since the random samples generated using wormholes do not guarantee inter-node independence, these samples cannot be used to generate new wormhole links. So, another PSS is required that provides independent samples of public nodes. We call this the bootstrap PSS. The bootstrap PSS will have relatively low load, since wormholes are refreshed rarely, given that WPSS is designed to reduce the number of new network connections created. For this reason, the bootstrap PSS can be implemented even as a central server. Alternatively, public nodes can generate independent samples by periodically starting random walks that continue until they reach their TTL.

In our architecture, the base overlay can be any connected overlay, as long as the TTL of the advertisements is set to make sure the random walks are long enough to provide high quality samples. It is beneficial, however, to maintain a random base overlay due to its low mixing time [3], [15]. Therefore, we construct a stable undirected overlay that is random, but where private nodes are connected to a random set of public nodes, thus avoiding the need for NAT traversal. This results in

Algorithm 1 Wormhole peer sampling

```
 1: procedure onWormholeFailure ⟨⟩
 2:     wormhole ← getNewWormhole()
 3: end procedure
 4:
 5: procedure onWormholeTimeout ⟨⟩          ▷ Every Δ_wh time units
 6:     wormhole ← getNewWormhole()
 7: end procedure
 8:
 9: procedure onAdTimeout ⟨⟩                ▷ Every Δ time units
10:     ad ← createAd()
11:     hops ← 1
12:     sendAd(wormhole, ad, hops)
13: end procedure
14:
15: procedure onReceiveAd ⟨ad, hops⟩
16:     if hops == getTTL() || acceptAd(ad) then
17:         view.addAd(ad)
18:     else
19:         j ← getMetropolisHastingsNeighbor(baseOverlay)
20:         sendAd(j, ad, hops+1)
21: end procedure
```

a stable topology with a low clustering coefficient, a constant degree for private nodes, and a narrow (roughly binomial) degree distribution for public nodes. This is similar to the topology maintained by Croupier [5], although Croupier's topology is dynamic.

The base overlay also needs a PSS in order to replace broken links with new random neighbors. The samples used to replace random links can be obtained either from the bootstrap PSS or from WPSS itself, with the choice depending on application requirements. For example, with relatively low churn rates the bootstrap PSS might be a better choice. However, under high churn, when many samples are needed, the cheaper WPSS samples are preferable despite the lack of inter-node independence that might result in higher clustering. This potentially higher clustering is not a problem as long as the network maintains good enough mixing properties.

The base overlay service strives to keep the overlay connected by keeping and repairing in case of failures a fixed number of outgoing links. In order to detect failures of base topology links, we implement a simple failure detector based on timeouts.

B. Wormhole Peer Sampling Skeleton

Algorithm 1 contains the pseudocode of WPSS that implements the ideas described above. The algorithm contains a number of abstract methods, implementations of which will be discussed in the subsequent sections. The algorithm is formulated as a set of event-handlers that are in place on every node in the system. The events in the system are classified into three types: failures, timeouts and advertisements.

Failure events are generated by detecting failing neighboring nodes in the two topology layers in Figure 1. We deal with these failure events by picking a new neighbor using the appropriate PSS through the abstract methods GetNewLink and GetNewWormhole. Timeout events are generated by two local timers with a tunable period. The periods of these timers are protocol parameters that are the same at all nodes. One of the timers, the wormhole timer, has a period of Δ_{wh}. This timer triggers the generation of new wormholes.

The other timer defines the rate at which advertisements are published by all the nodes. Clearly, this rate is the same as the average rate of receiving new random samples, so the period of this timer must be Δ at all nodes. This timer triggers the sending of one advertisement over the local wormhole. Finally, the event of receiving an advertisement is handled by first checking whether the node is willing to add the given sample to its set of samples or View (that is, whether it will consume the advertisement).

This check is performed by the AcceptAd method. The AcceptAd method consumes an advertisement only if its sample is not already contained in the node's view, thus promoting diversity of samples. On top of that, AcceptAd makes sure that every node consumes advertisements at the same rate, namely one advertisement in each Δ time period. We implement this rate control mechanism only at the public nodes; if the acceptance rate on the public nodes is $1/\Delta$, then private nodes are guaranteed to observe the same rate on average due to the advertisement generation rate being $1/\Delta$ at all the nodes. To control the rate, we need to approximate the period of receiving advertisements. To do this, we calculate the running average and the average deviation of the delays between consecutive acceptance events. If the approximated period, increased by mean deviation, is higher than Δ then the advertisement is accepted and the approximation of the period is updated. We show later in Section VI that the AcceptAd method successfully balances advertisements over public and private nodes.

If the node does not consume the advertisement, it sends it on to another node using the Metropolis-Hastings transition probabilities over the (random, stable) base network which results in a uniform stationary distribution [16]. Let d_i denote the degree of node i, that is, the number of neighbors of node i. Note that the graph is undirected. The implementation of GetMetropolisHastingsNeighbor works as follows. First, we select a neighbor j with uniform probability, that is, with probability $1/d_i$. Then, we return j with probability $\min(d_i/d_j, 1)$, otherwise we return node i itself, that is, the advertisement will visit i again.

Note that the node will consume the advertisement also if the TTL of the advertisement is reached, since at that point it can be considered a uniform random sample. The TTL is returned by GetTTL, that can be implemented in many different ways depending on the system properties. We simply set a constant TTL in our experiments.

V. ANALYTICAL COMPARISON

Here, we provide an analytical comparison between related work and WPSS. In order to do that, we develop a number of metrics and criteria. As a general assumption, we assume that the considered PSSes provide a continuous stream of random node descriptors at every node in the network. To be more precise, we require each node to receive one sample every Δ time units on average, where Δ is the *sampling period*.

A. Metrics of Quality of Service and Cost

The minimum that we assume about any PSS is that the stream of samples at a given fixed node is *unbiased* (that is, the samples received by a node are independent of the node) and that samples are *uniform* (that is, any fixed individual

sample has a uniform distribution). These properties hold for WPSS. This is almost trivial and can be formally proven using the facts that WPSS is blind to the content of node descriptors and wormholes are random. The technical proof is omitted due to lack of space.

Apart from this, we characterize the quality of service using the properties of *freshness* and *independence*. Freshness is the age of the sample, that is, the delay between the recording of the sample at the source node and its delivery at the terminating node. Obviously, fresher samples are always preferable. Independence is a property of the random samples that states whether the stream of samples generated at any two nodes are statistically independent of one another or not. Independence is not a hard requirement for many applications. Having unbiased and uniform streams (that are not necessarily independent) is often sufficient.

We focus now on two costs for the PSSes: *bandwidth* and *link creation rate*. Bandwidth is the amount of data transferred per received sample. The link creation rate is the number of new network connections (links) that are created (or in other words, the number of new network connections established) per sample. This new measure is motivated by our experience with deployed commercial P2P streaming.

B. Analysis of Existing PSS Algorithms

In Table I, we summarize the quality of service and costs of the PSS algorithm space for existing RW and gossiping approaches, where δ_{hop} is the average latency for a hop in a RW. The example given here results in a connection establishment that is tolerable for our application and good enough level of freshness for samples.

In the case of RW-push, the initiating node's descriptor will be added to the final node's sample set (the node where the RW terminates). In the case of RW-pull, the final node's descriptor is added to the sample set of the initiating node. In addition, if the physical network supports efficient routing then after the RW terminates a new direct link (DL) can be created by the final node to fetch (push) or send (pull) a fresh sample as well. Clearly, using direct links one can achieve a higher freshness at the cost of creating new links.

For the gossip-based protocols, we consider an idealized gossip protocol that has a view size of $2k$, and a gossip period of $k\Delta$. This setting makes sure that a node gets one new sample per sampling period (Δ) on average, since, in a typical gossip-based PSS, nodes refresh half of their views in each round. This is an optimistic assumption, because k is an upper bound on the independent samples.

We consider the two classical gossip protocols: *healer* and *swapper* [2]. Let us provide a quick intuitive analysis of the freshness for these protocols. In the case of healer, fresher descriptors always have preference when spreading. This results in an exponential initial spreading of newly inserted descriptors, until their age reaches about the logarithm of the view size. At that point the fresher descriptors take over. This results in an average age that is equal to $O(\log k)$, where age is measured in gossip rounds. So, freshness equals $O(k \log k)\Delta$. In the case of swapper, descriptors perform batched RWs, during which no descriptor gets replicated. In every round, every node removes one random descriptor and replaces it with its own new descriptor. For this reason, the age of a descriptor follows a geometric distribution, where the expected age (measured in number of rounds) is proportional to the view size: $O(k)$. So the average freshness equals $O(k^2)\Delta$.

Let us now discuss the NAT-aware gossip protocols. Croupier has the same costs and freshness as *swapper* (although node-level load is higher at public than private nodes, as they provide shuffling services). In Gozar, public nodes create the same number of new links and generate the same bandwidth as *healer*. Private nodes, on the other hand, generate $j*2$ times more bandwidth, as messages are sent over 2 links: first to the j relays and then to the private nodes. Gozar also creates and maintains an extra constant number of links to relays ($j*N$, where N is system size). Nylon has unbounded costs, as routing paths to private nodes can be arbitrarily long, and for that reason it is not considered here.

For WPSS, the most important parameter is Δ_{wh}, the wormhole renewal period. One important value that is determined by Δ_{wh}/Δ is the average number of hops that an advertisement makes before being accepted by a node. Let us denote this value by $h(\Delta_{wh}/\Delta)$. Now, the freshness of samples can be calculated using $h(\Delta_{wh}/\Delta)\delta_{hops}$. The number of newly created links per sample is Δ/Δ_{wh}, since the only new links are the wormholes. Finally, the bandwidth utilization per sample is given by $h(\Delta_{wh}/\Delta) + \Delta TTL/\Delta_{wh}$ where the second addend accounts for the cost of the bootstrap PSS, assuming it is implemented as a RW over the stable random topology.

Clearly, since Δ is a constant given by the application requirements, and Δ_{wh} (and thus Δ/Δ_{wh}) is a parameter of the protocol, the most important component is $h(\Delta_{wh}/\Delta)$. We know from the definition of the protocol that $h(1) = 1$ (one hop through the wormhole), and $\lim_{x\to\infty} h(x) = TTL$. They key question is whether $h(\Delta_{wh}/\Delta)$ grows slowly. In Section VI, we show that this is indeed the case.

We calculate the properties of the different protocols using a typical practical parameter setting, with the results summarized in Table I. For this example, we set $k = 10$, and $\delta_{hop} = 0.1$ seconds. We assume that $TTL = 100$. During the experimental evaluation in Section VI we apply the same setting, and we present a justification for it as well.

For WPSS, we set $\Delta_{wh} = 10\Delta$, which is shown in Section VI to provide a good trade-off between reducing the number of new connections created per round and keeping samples fresh. For $\Delta_{wh} = 10\Delta$, the freshness is evaluated in our experiments in Section VI as $h(10) \approx 3$, and, thus, this value is given in the table.

Note that freshness depends on the sampling period Δ only in the case of the gossip protocols. Typical settings for Δ range from 1 to 10 seconds. Besides, it is not meaningful to set $\Delta < \delta_{hop}$, so we can conclude that WPSS provides samples that are fresher by an order of magnitude than those provided by the fastest possible gossip protocol.

We would like to point out that gossip protocols will use a lot less bandwidth compared to the other protocols under most parameter settings, assuming our invariant (which fixes the same rate of receiving new samples). If we allow for the same bandwidth for gossip protocols and WPSS by speeding up gossip then gossip samples will get proportionally fresher, besides, proportionally more samples will arrive in a unit time as well. However, freshness will still depend on the length of

TABLE I: Comparison of PSS implementations. The example uses $k = 10$, $\delta_{hop} = 0.1$, TTL=100, and $\Delta_{wh} = 10\Delta$. These values result in a connection establishment time that is tolerable for our application and provides a good enough level of freshness for samples.

Algorithm	Quality of service		Cost per sample		Example		
	Freshness	Indep.	# New links	Bandwidth	Freshness	# New links	Bandwidth
RW-push	$TTL \cdot \delta_{hop}$	yes	0	TTL	10	0	100
RW-push-DL	δ_{hop}	yes	1	TTL+1	0.1	1	101
RW-pull	$2 \cdot TTL \cdot \delta_{hop}$	yes	0	$2 \cdot$TTL	20	0	200
RW-pull-DL	δ_{hop}	yes	1	TTL+1	0.1	1	101
gossip healer	$O(k \log k)\Delta$	no	$1/k$	1	33Δ	0.1	1
gossip swapper	$O(k^2)\Delta$	no	$1/k$	1	100Δ	0.1	1
WPSS	$h(\Delta_{wh}/\Delta)\delta_{hop}$	no	Δ/Δ_{wh}	$h(\Delta_{wh}/\Delta) + \Delta TTL/\Delta_{wh}$	**0.3**	**0.1**	**13**

the (now shorter) gossip cycle. In addition, most importantly, the number of new links will increase proportionally since all the connections in gossip protocols are new connections. Reducing the number of new links is one of our main motivations.

Gossip protocols have a further advantage due to batching $2k$ advertisements in a single message. This could additionally reduce bandwidth costs, as the relative amount of meta-data per packet sent is lower if the network packets from gossip messages are closer in size to the network's maximum transmission unit (MTU) than WPSS messages. However, in our target application the size of an advertisement can be large, which justifies our focus on bandwidth as a measure of cost as opposed to the number of messages sent.

Based on the above cost analysis of existing PSS algorithms, our main observation is that no single method other than WPSS offers a combination of three desirable properties for a PSS: fresh samples, a low number of new links, and low bandwidth overhead. Apart from being vulnerable to failures, RW methods can offer fresh samples with an extremely high number of new links (one per each sample) or relatively old samples without creating new links; in both cases with a relatively high bandwidth. Gossip based methods provide even older samples than RW methods.

VI. EVALUATION

We now evaluate the performance of WPSS in simulation as well as in our deployed system. The experiments on the deployed system show that our protocol provides the desirable PSS properties of fresh samples, randomness, and low cost in a real environment. The simulation experiments, in contrast, test robustness in scenarios that are difficult or impossible to reproduce in deployment, such as different churn levels.

Our implementation follows the structure outlined in Figure 1. The bootstrap service component provides addresses of random public nodes that are used by the upper layer to build the base overlay and to create wormholes. In our implementation, the bootstrap service initiates RWs from the public nodes in the base overlay using the same transition probabilities as used by WPSS. The rate of starting these walks is double the rate of wormhole renewal to account for failure events in the base overlay and the wormhole overlay. If the bootstrap service runs out of local random samples generated by RWs (because of massive join or failure events), a central bootstrap server is contacted for random samples.

We set $TTL = 100$. After a RW of this length, the distribution is very close to uniform in the base overlays we apply here. More precisely, its total variational distance [15] from the uniform distribution is less than 10^{-6} in all base overlays in this section. In a fully public network—but with the same number of nodes and random links—the same quality can be reached with $TTL = 16$. This means that in our environment, independent RW methods have a further disadvantage due to the constrained topology. By increasing the network size, this difference becomes larger. As we will see, WPSS walks will always terminate much sooner in most practical settings, so the TTL is not a critical parameter from that point of view.

The base overlay service strives to keep the overlay connected by identifying and repairing broken links, thus, maintaining a fixed number of outgoing links at each node. In order to detect failures of base topology links and the wormhole, we implement a simple failure detector based on timeouts. Every node maintains 20 links to random public nodes in the base overlay. However, these links are bidirectional, so the effective average degree is 40 as a result.

A. Experimental Setup: Simulation

For simulations, we implemented WPSS on the Kompics platform [17]. Kompics provides a framework for building P2P systems and a discrete event simulator for evaluating those systems under different churn, latency and bandwidth scenarios. All experiments are averaged over 6 runs. Unless stated otherwise, we applied the following settings. The view size (the number of freshest random samples a node remembers) was 50 and we set $\Delta = 1$ second. For all simulation experiments, we use a scenario of $N = 1000$ nodes that join following a Poisson distribution with a mean inter-arrival time of 100 milliseconds. In all simulations 20% of the nodes were public and 80% were private, which reflects the distribution observed in the commercial deployments of our P2P application.

B. Experimental Setup: Deployment

In order to test WPSS in a real environment, we implemented the protocol using [18], a production-quality framework for building event-based distributed applications. The framework utilizes a UDP-based transport library that implements the same reliability and flow control mechanisms as TCP [19].

We tested WPSS in our test network, where volunteers give us permission to conduct experiments on their machines using a remotely-controlled test agent. The test network contains around 12000 installations. The network included nodes mostly from Sweden (89%) but also some from Europe (6%) and USA (4%). For connectivity, 76% of the nodes were behind NATs or firewalls and 19.2% were public nodes, while
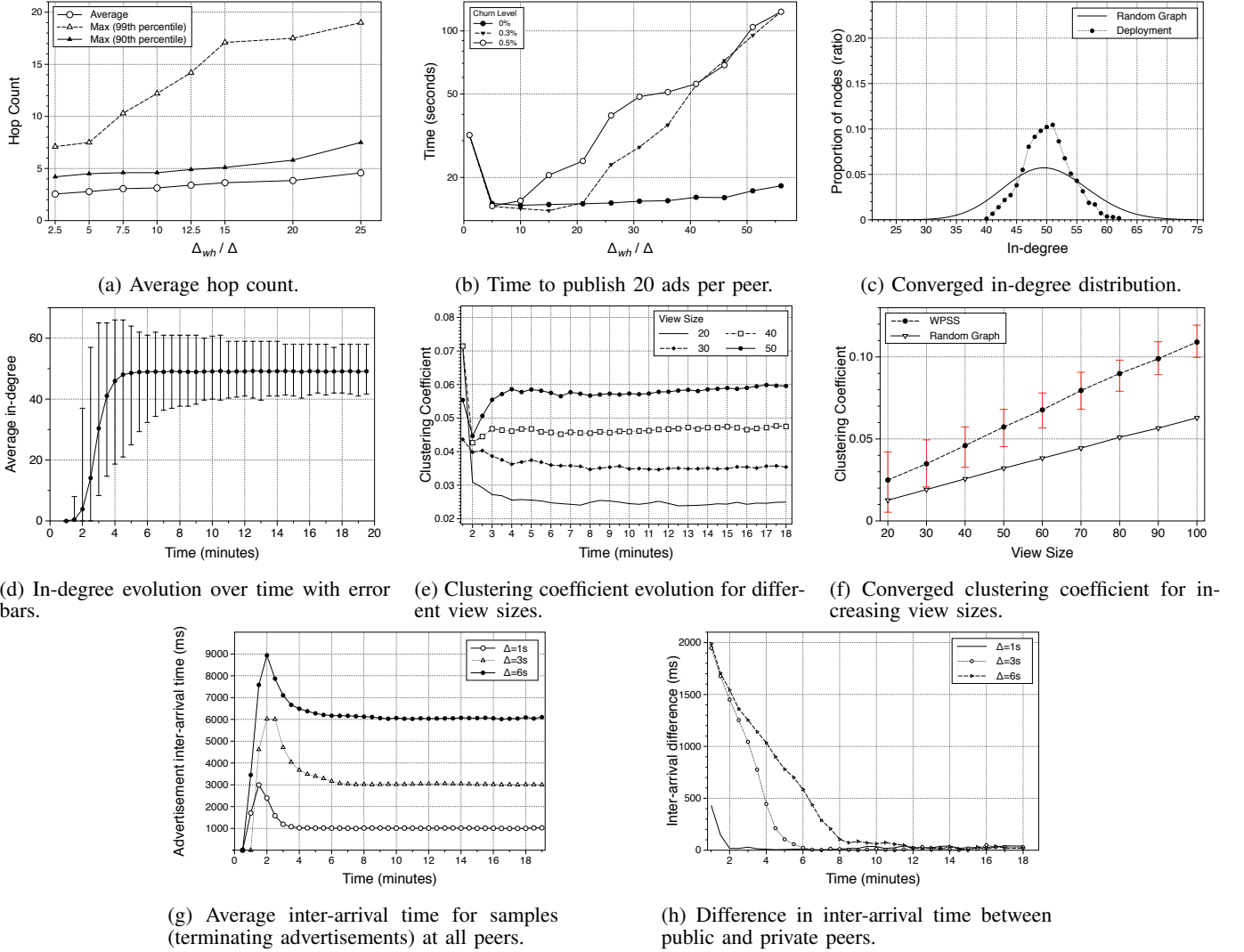
(a) Average hop count.

(b) Time to publish 20 ads per peer.

(c) Converged in-degree distribution.

(d) In-degree evolution over time with error bars.

(e) Clustering coefficient evolution for different view sizes.

(f) Converged clustering coefficient for increasing view sizes.

(g) Average inter-arrival time for samples (terminating advertisements) at all peers.

(h) Difference in inter-arrival time between public and private peers.

Fig. 3: *Experiments in our deployed system (except Fig. 3b, which is in simulation).*

the rest (4.8%) could not be determined by our NAT-type identification protocol.

Each experiment is run in wall-clock time and thus it is subject to fluctuations in network conditions and in the number of nodes involved. In order to keep a good level of reproducibility, we selected a subset of 2000 of the more stable nodes, out of an average of 6200 online, which are representative of both our test network and the Internet in Sweden, in general. For each data point, the deployment experiments were repeated a number of times varying from 3 to 10 runs, depending on the variance of the results, and every run lasted 20 minutes. In all deployment experiments, the nodes join at a uniform random point in time within the first 2 minutes from the start of the test. Unless stated otherwise, we set the a view size to 50, and $\Delta = 2$ seconds.

C. Freshness

In this set of experiments, we measure the freshness of samples on our deployed system using the average hop count as well as the $90th$ and $99th$ percentiles. As we can see in Figure 3a, both the average hop count ($h(\Delta_{wh}/\Delta)$) and the

90th percentile grow slowly with increasing Δ_{wh} (wormhole renewal period), while the TTL is never reached by any advertisement. The $99th$ percentile, however, grows more quickly when Δ_{wh}/Δ exceeds 5 seconds as a small number of public nodes have under-performing or too few connections, meaning that advertisements that arrive at them via wormholes have to take more hops to complete.

In simulation, we now measure the average time required for a node to publish 20 consecutive advertisements. This characterizes the average freshness of the advertisements also taking into account all possible delays, not only hop count. We examine scenarios with and without churn. The total time we measure includes the time required for an advertisement to be accepted by a node ($h(\Delta_{wh}/\Delta)\delta_{hops}$), and the amount of time devoted to opening new wormhole links, if applicable. It also includes the retransmission time in case of failures.

We note that the average time required to establish a network connection to a public node in our system and was measured to be $1250\,ms$ in deployment. In Figure 3b, we can observe the freshness of advertisements under three different scenarios: with no churn, and with a churn level of 0.3% or
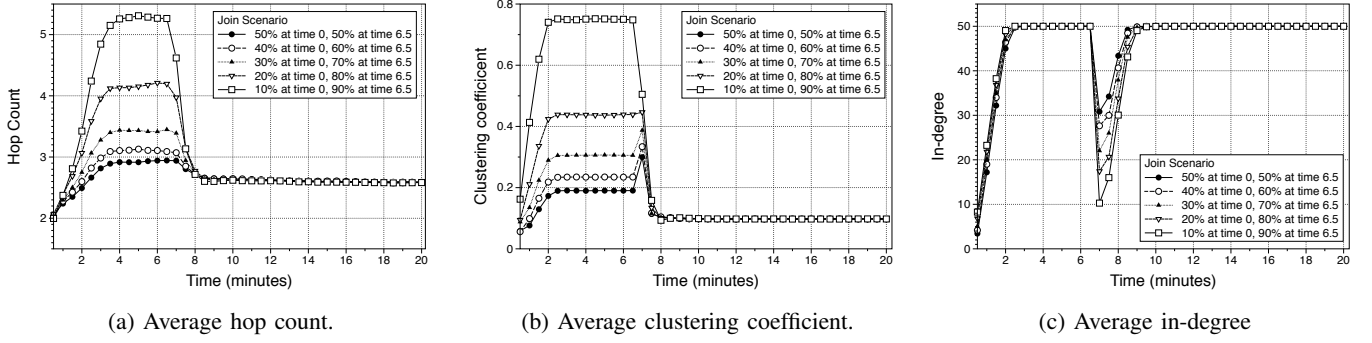
(a) Average hop count.

(b) Average clustering coefficient.

(c) Average in-degree

Fig. 4: *Flash crowd scenario for different sizes of flash crowd.*

(a) Average hop count.

(b) Average clustering coefficient.
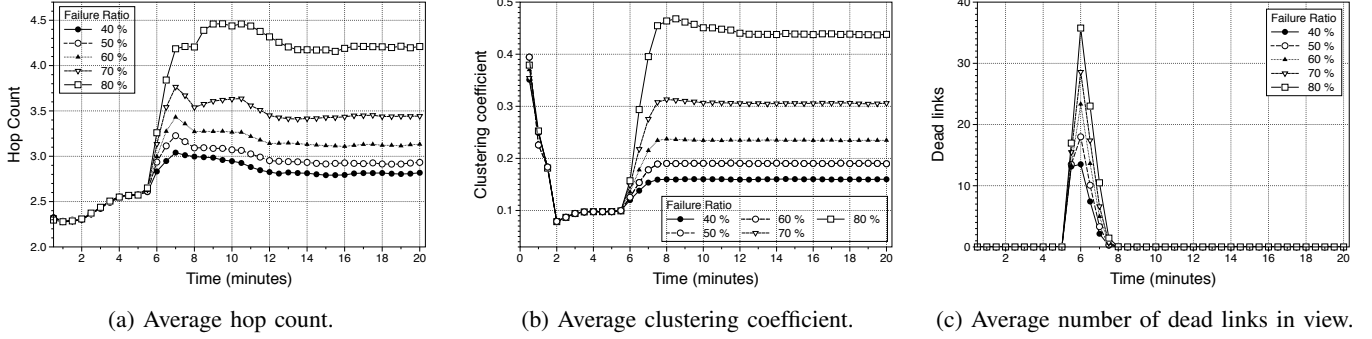

(c) Average number of dead links in view.

Fig. 5: *Catastrophic failure scenarios in WPSS for different ratios of failed nodes.*

0.5% of nodes failing and joining every 10 seconds. Recall that in simulation we had $\Delta = 1$ second. We consider these churn figures to be representative of the deployments of our commercial P2P application. The required time to publish an advertisement increases with Δ_{wh}/Δ, especially in the scenarios with churn. This is due to both a higher average hop count, as shown later in the evaluation, and to retransmissions caused by failures along the advertisement path.

The most interesting result here is that the performance is optimal for Δ_{wh}/Δ between about 5 and 10 even under high churn. Given this finding, we set $\Delta_{wh} = 10\Delta$ for the remaining experiments, as this value has good freshness (even considering the $99th$ percentile), while the number of new links required is relatively low.

D. Randomness

Similar to [2], [4], [5], we evaluate here the global randomness properties of our deployed system by measuring properties of the WPSS overlay topology (that is, not the base overlay, but the overlay that is defined by the samples stored at nodes). In this set of experiments, we measure the indegree distribution of the WPSS overlay network, its convergence time for different view sizes, and finally its clustering coefficient for different view sizes.

With samples drawn uniformly at random, we expect that the in-degree to follow the binomial distribution. In Figure 3c, we can see that WPSS actually results in a distribution that is even narrower than what is predicted by the uniform random case, which suggests good load balancing. In Figure 3d we can also see that the average indegree converges after around 4 minutes and the variance after around 8 minutes, respectively; an acceptable convergence time for our system. Since in

deployment we had $\Delta = 2$ seconds, this translates to 120Δ and 240Δ, respectively.

In Figure 3e, we can see that the clustering coefficient converges at roughly the same rate as the indegree distribution. Figure 3f indicates that the clustering coefficient is higher than that of the random graph by a constant factor, which is due to the fact that WPSS does not guarantee independent samples at nodes that are close in the stable base overlay, as we explained previously. As we increase the view size, the clustering coefficient increases simply due to the larger chance of triangles; this is true for the random graph as well.

E. Inter-arrival Time of Advertisements

These experiments were again conducted in the deployed system. Figure 3g shows that the average inter-arrival times for samples (advertisements) converges to the respective advertisement period after roughly 120Δ seconds. As public nodes can be wormhole exits, they receive samples at a higher rate than private nodes. But, as Figure 3h shows, the method $acceptAd$ (see Section IV) successfully balances samples (advertisements) across public and private nodes.

F. Robustness to Different Churn Patterns

We experiment with several churn patterns in simulation. Figure 4 shows our results with flash crowd scenarios, where we progressively decrease the number of peers that join at the beginning of the simulation while increasing the number of those that join at a later point in time. For instance, for a flash crowd scenario of 70% of the nodes, 300 nodes start to join at time 0 and the other 700 nodes start to join the system at minute 6.5. The number of nodes involved in all experiments stays constant at 1000. As we can observe in Figure 4a, the
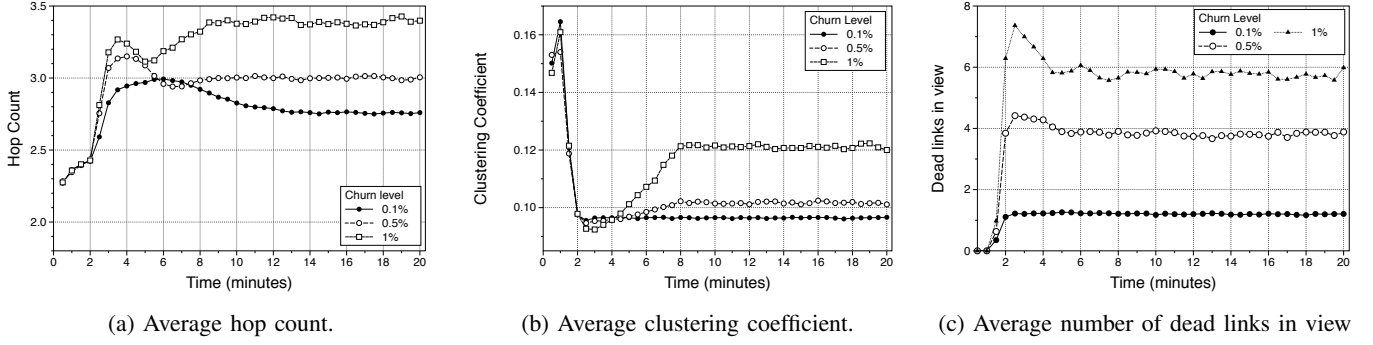
(a) Average hop count.

(b) Average clustering coefficient.

(c) Average number of dead links in view

Fig. 6: *WPSS under churn scenarios with varying levels of churn.*

(a) Average hop count.

(b) Average clustering coefficient.
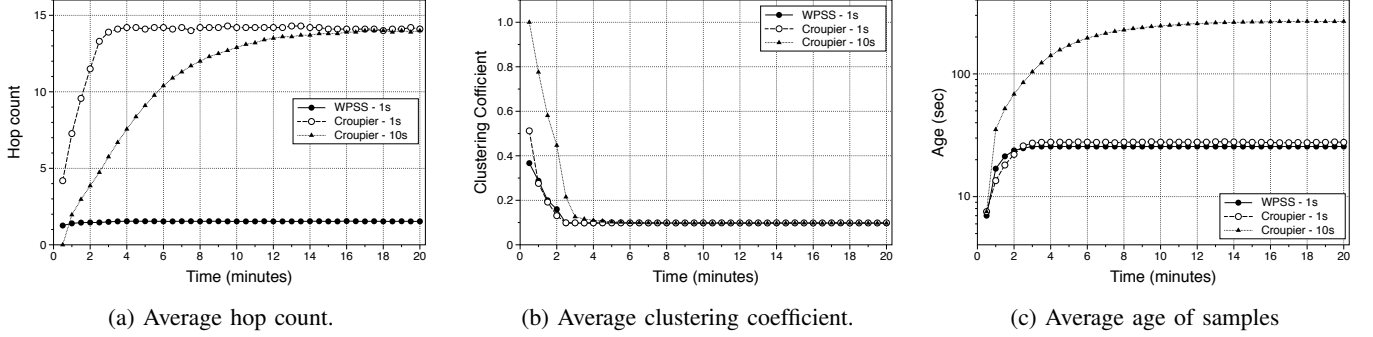
(c) Average age of samples

Fig. 7: *Comparison between WPSS and Croupier.*

hop count not only stabilizes quickly after the flash crowd terminates, but it also converges to the same value for all flash crowd scenarios.

The clustering coefficient exhibits a similar behavior, that is, it stabilizes quickly after the flash crowd to the same value for all flash crowd sizes. The converged clustering coefficient value (0.0975) is almost identical to the converged value of *gossip healer* (0.0960) [2] in a scenario with no churn. Figure 4c shows that the average indegree drops significantly during the flash crowd but it recovers quickly after that.

In Figure 5, we also show that the protocol is robust to catastrophic failures, that is, when a large number of peers leaves the system at a single instant in time. In this scenario, we wait for the overlay to stabilize after all the nodes have joined, then we fail a percentage of peers drawn uniformly from the set of all peers at time 5 minutes. It is important to note that the overlay remains connected even after the failure of 80% of the nodes.

As we can observe in Figure 5a, the average hop count stabilizes more slowly for higher percentages of failed nodes. This is expected given the large number of broken links to detect and repair for both the base and wormhole overlays. The clustering coefficient converges somewhat quicker. Note that the clustering coefficient stabilizes on higher values, because the number of remaining nodes is lower after the failure. More precisely, for both clustering coefficient and hop count, the converged values after the mass failure are the same as they would have been in a network that hadn't experienced mass failure but had the same size as our system had size after the mass failure. Two minutes after the mass failures, in all scenarios, dead links have been expelled from the view, as shown in Figure 6c.

In Figure 6, we also show that the protocol is robust to different levels of steady churn, with up to one percent of the nodes failing and joining every wormhole refresh period (10 seconds). Figure 6a shows that for a level of churn of 0.1%, the hop count takes a longer time to stabilize compared to a scenario without churn, but it converges nevertheless to the same value as in the scenario without churn. For levels of churn of 0.5% and 1%, the average hop count increases by 7% and 21%, respectively, which is still acceptable in terms of freshness. Another effect of churn is the steady increase of the clustering coefficient, as shown in Figure 6b.

In order to measure the extent of damage of continuous churn, we also show in Figure 6c the average number of dead links in a node's view. From the figure, it is clear that it is proportional to the level of churn.

G. Comparison with Croupier

We conclude our evaluation with a comparison with the state of the art NAT-aware PSS, Croupier. We use the same experiment setup in Croupier as in WPSS (the same number of nodes, ratio of public/private nodes and join distribution) with a view size of 50 in Croupier. We compare WPSS and Croupier using well-established properties for good PSSes, and we show that WPSS has better results for global randomness (through the network properties of the clustering coefficient, and the in-degree distribution of graph of samples), the freshness of samples, and how many hops node descriptors have to traverse before being placed as a sample.

For a fair comparison with Croupier, we include two settings, one that creates new network connections at the same rate as WPSS (Croupier-10s) with the Croupier gossip round time is set to 10 seconds, and another setting that has the
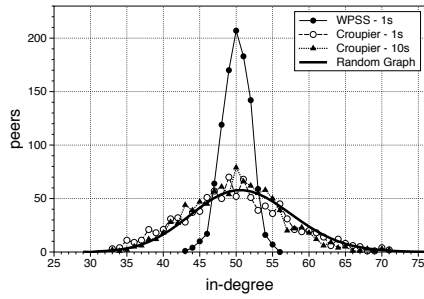
Fig. 8: *Comparison of in-degree distribution between WPSS and Croupier.*

same round time as WPSS (Croupier-1s), but, for this setting, the network connection establishment rate is 10 times higher than WPSS. In Figure 7b, we can see that the clustering coefficient of WPSS converges more quickly than in Croupier-10s, but at only a slightly faster rate than Croupier-1s. Both protocols have low clustering coefficients, close to random graphs. In Figure 8 we can see that WPSS has a much narrower in-degree distribution than Cropuier around the expected in-degree, indicating better load balancing of samples around all the nodes.

In Figure 7a, we can see that average hop count in WPSS is stable and low over time, while in Croupier the average hop count increases as node descriptors spread throughout the system until they finally are expired.

In Figure 7c, we can see the average age (freshness) of samples generated by WPSS is roughly the same as Croupier-1s at 11.5 seconds, but much better than Croupier-10s. As Croupier is based on the swapper policy, from our earlier Table I, we can see that the average freshness of its samples are close to Swapper's expected value of 10s, with the extra second resulting from the private nodes having to use an extra hop to public nodes who shuffle descriptors on behalf of private nodes.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we presented WPSS, a peer sampling service to meet the requirements of commercially deployed P2P systems on the Internet. WPSS executes short random walks over a stable topology and by using shortcuts (wormholes). WPSS provides the same level of sample freshness as other NAT-aware PSSes, but achieves that with a connection establishment rate that is one order of magnitude lower.

In addition, the connection establishment rate of WPSS can be tuned from zero per sample to up one per sample, according to the application requirements on freshness and cost. We showed in our deployed live-streaming P2P system that the lowest cost connection creation rate lies between these two extremes. On top top of that, we experimentally demonstrated that our system has the randomness properties required of a peer sampling service, while it is robust to churn and large-scale failure scenarios.

While we have designed WPSS for the Internet, we believe it is general enough to work over any type of stable base overlay (given a high enough TTL for RWs) and any subset of nodes can act as wormholes. As part of our future work, we

will consider applying WPSS to mobile and sensor networks, and overlay networks without NATs or firewalls.

REFERENCES

[1] G. Kreitz and F. Niemela, "Spotify – large scale, low latency, P2P Music-on-Demand streaming," in *Tenth IEEE Intl. Conf. on Peer-to-Peer Computing (P2P'10)*. IEEE, August 2010, pp. 1–10.
[2] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "Gossip-based peer sampling," *ACM Transactions on Computer Systems*, vol. 25, no. 3, p. 8, August 2007.
[3] D. Stutzbach, R. Rejaie, N. Duffield, S. Sen, and W. Willinger, "On unbiased sampling for unstructured peer-to-peer networks," *IEEE/ACM Transactions on Networking*, vol. 17, no. 2, pp. 377–390, April 2009.
[4] A. Payberah, J. Dowling, and S. Haridi, "Gozar: NAT-friendly peer sampling with one-hop distributed NAT traversal," in *Distributed Applications and Interoperable Systems*, ser. LNCS, vol. 6723. Springer, 2011, pp. 1–14.
[5] J. Dowling and A. H. Payberah, "Shuffling with a croupier: Nat-aware peer-sampling," in *Proceedings of The 32nd Intl. Conf. on Distributed Computing Systems (ICDCS 2012)*. Los Alamitos, CA, USA: IEEE Comp. Soc., 2012, pp. 102–111.
[6] A.-M. Kermarrec, A. Pace, V. Quema, and V. Schiavoni, "NAT-resilient gossip peer sampling," in *Proc. 29th IEEE Intl. Conf. on Distributed Computing Systems*. IEEE Comp. Soc., 2009, pp. 360–367.
[7] S. Guha, N. Daswani, and R. Jain, "An Experimental Study of the Skype Peer-to-Peer VoIP System," in *Proceedings of IPTPS*, Santa Barbara, CA, February 2006, pp. 1–6.
[8] A. Bergkvist, D. Burnett, C. Jennings, and A. Narayanan, "Webrtc 1.0: Real-time communication between browsers," in *W3C Working Draft 21 August 2012*, 2012. [Online]. Available: http://www.w3.org/TR/webrtc/
[9] R. Roverso, S. El-Ansary, and S. Haridi, "Smoothcache: Http-live streaming goes peer-to-peer," in *NETWORKING 2012*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 29–43.
[10] I. Eyal, I. Keidar, and R. Rom, "Limosense – live monitoring in dynamic sensor networks," in *Algorithms for Sensor Systems*, ser. LNCS. Springer, 2012, vol. 7111, pp. 72–85.
[11] S. Voulgaris, D. Gavidia, and M. van Steen, "CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays," *Journal of Network and Systems Management*, vol. 13, no. 2, pp. 197–217, 2005.
[12] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié, "Peer-to-peer membership management for gossip-based protocols," *IEEE Transactions on Computers*, vol. 52, no. 2, February 2003.
[13] J. Leitão, R. van Renesse, and L. Rodrigues, "Balancing gossip exchanges in networks with firewalls." in *Proc. 9th Intl. Workshop on Peer-to-Peer Systems (IPTPS '10)*. USENIX, 2010, p. 7.
[14] R. Roverso, S. El-Ansary, and S. Haridi, "NATCracker: NAT combinations matter," in *Proceedings of 18th Internatonal Conf. on Computer Communications and Networks (ICCCN)*, August 2009, pp. 1–7.
[15] L. Lovász and P. Winkler, "Mixing of random walks and other diffusions on a graph," in *Surveys in Combinatorics*, ser. London Math. Soc. Lecture Notes Series, P. Rowlinson, Ed. Cambridge University Press, 1995, vol. 218, pp. 119–154.
[16] S. Chib and E. Greenberg, "Understanding the metropolis-hastings algorithm," *The American Statistician*, vol. 49, no. 4, pp. 327–335.
[17] C. Arad, J. Dowling, and S. Haridi, "Message-passing concurrency for scalable, stateful, reconfigurable middleware," in *Middleware 2012*, ser. LNCS. Springer, 2012, vol. 7662, pp. 208–228.
[18] R. Roverso, S. El-Ansary, A. Gkogkas, and S. Haridi, "Mesmerizer: a effective tool for a complete peer-to-peer software development life-cycle," in *SIMUTools '11*, 2011.
[19] R. Reale, R. Roverso, S. El-Ansary, and S. Haridi, "DTL: Dynamic Transport Library for Peer-To-Peer Applications," in *In Proc. of the 12th Intl. Conf. on Distributed Computing and Networking*, ser. ICDCN, Jan 2012.