NAT-resilient Gossip Peer Sampling

Anne-Marie Kermarrec, Alessio Pace, Vivien Quema, Valerio Schiavoni

NAT-resilient Gossip Peer Sampling

Anne-Marie Kermarrec
INRIA Rennes - Bretagne Atlantique

Alessio Pace
INRIA Grenoble - Rhône-Alpes

Vivien Quéma
CNRS

Valerio Schiavoni
INRIA Grenoble - Rhône-Alpes

Abstract

Gossip peer sampling protocols now represent a solid basis to build and maintain peer to peer (p2p) overlay networks. They typically provide peers with a random sample of the network and maintain connectivity in highly dynamic settings. They rely on the assumption that, at any time, each peer is able to establish a communication with any of the peers of the sample provided by the protocol. Yet, this ignores the fact that there is a significant proportion of peers that now sit behind NAT devices (70% is a fair ratio in the current Internet), preventing direct communication without specific mechanisms. This has been largely ignored so far in the community. Our experiments demonstrate that the presence of NATs, introducing some restrictions on the communication between peers, significantly hurts both the randomness of the provided samples and the connectivity of the p2p overlay network, in particular in the presence of high rate of peers arrivals, departures and failures (aka churn). In this paper we propose a NAT-resilient gossip peer sampling protocol, called $\mathcal{N}ylon$, that accounts for the presence of NATs. $\mathcal{N}ylon$ is fully decentralized and spreads evenly among peers the extra load caused by the presence of NATs. $\mathcal{N}ylon$ ensures that a peer can always establish a communication, and therefore initiates a gossip, with any peer in its sample. This is achieved through a simple, yet efficient mechanism, establishing a path of relays between peers. Our results show that the randomness of the generated samples is preserved, that the connectivity is not impacted even in the presence of high churn and a high ratio of peers sitting behind NAT devices.

1 Introduction

Gossip protocols have received an increasing attention in distributed computing over the past decade as they are robust, simple and highly resilient to churn. Gossip random peer sampling protocols are extensively used in that area to build and maintain unstructured networks.

In gossip peer sampling, each peer typically maintains a set of neighbors (called its view) which it periodically exchanges with another peer in the system, picked from its view. This view is expected to be a sample of peers picked uniformly at random among all peers. Such protocols rely on the implicit assumption that a peer is able to communicate with any peer of its view. Yet, it is a well known fact that today, a large number of peers sit behind NATs [4] (such peers are called *natted* in the sequel, while other peers are called *public*). NAT devices allow several peers with a private IP address to share a single public IP address. NATs implement firewall-like mechanisms that drop unsolicited incoming messages. Consequently, the presence of NATs between peers may prevent them to communicate directly.

While this issue has been addressed in the context of structured p2p networks [4, 9], it has been mostly ignored in the area of gossip protocols so far. To the best of our knowledge, the only work that deals with NATs in gossip protocol is [6]. In this solution, a peer p stores in a cache the peers with which it successfully communicated in the past. The presence of this cache is expected to ensure that at any time p has a high probability to know a peer with which it can communicate. Needless to say, such a simple mechanism cannot ensure that the network will remain connected. As we show in the sequel, the presence of natted peers significantly impacts the properties of the peer sampling protocol with respect to both the randomness of the provided samples and the connectivity. A straightforward cope out is to associate every natted peer to a public one. Provided the natted peer accepts incoming messages from its associated public peer, the latter can act as a relay between this natted peer and any other peer. Obviously, this imposes a significant overhead on public peers which is not acceptable.

In this paper we present $\mathcal{N}ylon$, a fully decentralized NAT-resilient gossip peer sampling protocol where the relay load is evenly spread among peers be they natted or public. This protocol ensures that the communication between a peer and its neighbors is always possible. As soon as a peer picks a neighbor n in its view to initiate a gossip, it

uses as relay the peer which gave it this specific entry to set up a communication with n, and becomes itself a relay to n. Note that the peer might rely on more than one relay to set up a communication with n. Typically, in our experiments, the chain of relays contains on average less that 4 peers in a system comprising 10.000 peers, 90% of which are natted. We show through a simulation study that $\mathcal{N}ylon$ *(i)* ensures that the properties of the peer sampling are preserved in the presence of NATs; *(ii)* evenly balances the relay load between peers; and *(iii)* is highly resilient to churn.

The rest of this paper is organized as follows. We provide a background on NAT in Section 2, we study the impact of the presence of NAT on existing peer sampling protocols in Section 3. Section 4 provides a description of our NAT resilient protocol. We report experimental results in Section 5. We discuss related works before concluding in Section 6.

2 Background on NATs

This section presents the various NAT devices and describes NAT traversal techniques allowing UDP message exchanges between natted peers. More details can be found in [19]. Note that in this section and in the rest of the paper, we do not consider nested NAT topologies.

2.1 NAT devices behavior

A NAT device typically orchestrates the communication between peers sitting behind it and the rest of the network (external peers). When a natted peer opens an outgoing TCP or UDP session through a NAT, the NAT assigns the session a public IP address and port number to allow subsequent messages from an external peer to be received. In addition, the NAT assigns the session a filtering rule, which specifies whether messages received from external peers on the assigned public IP address and port should be forwarded or not to the natted peer's private IP address and port. The public IP address and port mapping, as well as the filtering rule, only remain valid a limited time after the last message was sent (or received) in a session.

Existing NATs differ in the way they assign public IP addresses and ports, as well as in the filtering rules they implement. We briefly describe the four main NAT types.

Full Cone (FC). This is the most permissive type of NAT. The NAT assigns the same public IP address and port to all sessions started from a given natted peer's IP address and port. These sessions all share the same filtering rule, which states that the NAT must forward all incoming messages.

Restricted Cone (RC). This type of NAT imposes restrictions on the IP addresses of external peers that can send messages to natted peers. As for FC NATs, the RC NAT assigns the same public IP address and port to all sessions started from a given natted peer's IP address and port. All the sessions started from a given natted peer's IP address and port, and involving the same target IP address, share the same filtering rule: the NAT only forwards messages coming from this IP address.

Port Restricted Cone (PRC). This type of NAT imposes restrictions on the IP addresses and ports of external peers that can send messages to natted peers. As for the previous NAT types, the NAT assigns the same public IP address and port to all sessions started from a given natted peer's IP address and port. Nevertheless, each session started from a given natted peer's IP address and port towards a target IP address and port, has its own filtering rule. This rule states that the NAT only forwards messages coming from the target IP address and port to which the session has been opened.

Symmetric (SYM). This is the most restrictive type of NAT. For every session started from a given natted peer's IP address and port, the NAT always assigns the same public IP address but a different port. Note that contrarily to other NAT types, the mapping is destination-dependent. The filtering rule is similar to the one used in PRC NATs: the NAT device only forwards messages coming from the target IP address and port to which the session has been opened.

2.2 NAT traversal techniques

The public IP address and port mapping and the filtering rules determine how peers can communicate. As long as a peer behind a FC NAT regularly sends or receives messages through the public address and port the NAT device assigned to it, it will have a valid filtering rule forcing the NAT device to forward it all incoming messages. Rather, if the target peer is behind a RC, PRC, or SYM NAT, the source peer willing to communicate with it has to apply a so-called NAT traversal technique. NAT traversal techniques rely on the use of *rendez-vous* peers (*RVP*) able to exchange messages with both the source *and* the destination peers[1]. There exist two different techniques depending on the combination of source's and target's NAT type. The two techniques are described below. The table summarizes which one should be used in various configurations. Source peer's NAT type is given in the most-left column, whereas target peer's NAT type is given in the heading row.

	public	RC	PRC	SYM
public	direct	hole punching	hole punching	relay
RC	direct	hole punching	hole punching	hole punching
PRC	direct	hole punching	hole punching	relaying
SYM	direct	mod. hole punching	relaying	relaying

[1] *RVP* is usually a public node to which the source and destination peers periodically send PING messages.

Hole punching. In the hole punching technique, the source peer sends a PING message to the destination peer. Consequently, the source peer's NAT device creates a filtering rule forcing it to forward incoming messages from the destination peer. The source peer then sends an OPEN_HOLE message to an *RVP*, indicating that it wants to communicate with the destination peer. The *RVP* forwards the OPEN_HOLE message to the destination peer. As soon as it receives the OPEN_HOLE message, the destination peer sends a PONG message to the source peer. Thereafter, the NAT device of the destination peer has a valid filtering rule allowing incoming messages from the source peer (we say that there is a *hole* in the NAT). The source peer can start sending messages to the destination peer as soon as it receives the PONG message[2]. Note that for most combinations (i.e. those not involving SYM NATs), after the hole punching technique has been applied, the destination peer can also send messages directly to the source peer.

Relaying. In some cases, the hole punching mechanism cannot be used: when the destination peer is behind a SYM NAT and the source peer is either behind a PRC NAT or a SYM NAT, or when the destination peer is behind a PRC NAT and the source peer is behind a SYM NAT. This is due to the fact that the SYM NAT device assigns a different port to every new session, and this port is not known by the source peer. The only possibility for sending messages to the destination peer is then to use the *RVP* as a relay.

3 Impact of NATs on existing protocols

Various peer sampling protocols have been proposed [12, 17, 23]. The protocols described in [12, 17] rely on random walks. These protocols assume a fairly static peer interconnection topology and are not specifically designed to sustain high levels of churn. Conversely, gossip protocols have been designed to handle peers joining and leaving the system at a high rate. We focus on such protocols in the sequel.

A generic gossip peer sampling protocol is described in Figure 1. The system is composed of a set of uniquely identified peers, each one storing references to few other peers into a view. Typically, the view size is in the order of $log(n)$, where n is the number of peers in the network. The generic protocol works as follows: each peer periodically initiates a communication (i.e. gossips) with one target peer selected from its view. The source and/or the target peer exchange their views. When a peer receives a view, it merges it with its view, and truncates the result to a constant maximum view size. This is typically called a view *shuffling*.

A peer sampling protocol is expected to provide the following properties: (i) the graph formed by peer views remains connected, and (ii) every peer in the network has the same probability to be selected by other peers (the provided sample is *random*).

```
1  every shuffling_period units do
2    target ← select_gossip_destination(view)
3    send ⟨REQUEST, view⟩ to target
4    if push_pull then
5      receive ⟨RESPONSE, view_t⟩ from target
6      view ← merge_and_truncate(view, view_t)
7    increase_view_age()

8  on receive ⟨REQUEST, view_s⟩ from source do
9    if push_pull then
10     send ⟨RESPONSE, view⟩ to source
11   view ← merge_and_truncate(view, view_s)
12   increase_view_age()
```

Figure 1. Generic peer sampling protocol.

The generic gossip-based peer sampling protocol described in Figure 1 can be configured along the following three dimensions [11]: *(i)* Gossip target selection: can either be done randomly (rand), or by picking the oldest peer in the view (tail); *(ii)* View propagation: either only the source peer sends its view to the target peer (push), or both source and target peers exchange their view (push/pull); *(iii)* View merging: when truncating a view, randomly chosen peers are kept (rand), or the youngest ones (healer), or the ones received from the other peer (swapper).

We evaluated six different configurations of the generic protocol described in the previous section. The view propagation strategy is the same in all the configurations and is set to push/pull, which is most used in the literature as a push mode consistently exhibits significantly worse performances than push/pull. The gossip target selection and view merging strategies that we evaluated are those described above.

The experiments have been obtained through simulations. The network size is 10,000 peers, and the bootstrapping procedure is such that at the beginning of the simulation all peers' views are filled with randomly chosen public peers. The initial graph is thus always connected. No churn was considered. A more detailed description of the experimental setup is done in Section 5. Moreover, for the sake of simplicity, only PRC NATs are considered in the experiments presented in this section. We evaluated the protocols along the following metrics: *(i)* the resilience of the protocol with respect to network partitioning; *(ii)* the ratio of stale entries in the views and; *(iii)* the randomness of the resulting views.

Network partitions. Figure 2 shows the size of the biggest cluster as a function of the percentage of natted peers for

[2]When the source peer is behind a SYM NAT, the hole punching technique needs to be slightly modified. Indeed, as the destination peer does not know the public IP address and port that has been assigned to the source peer, it uses the *RVP* to send the PONG message to the source peer.

two view sizes (15 and 27). The biggest cluster size is expressed as the percentage of peers that belong to it. We clearly see that the graph partitions when the percentage of natted peers reaches a certain threshold (50% and 70% for the considered view sizes). We observe that, as expected, increasing the view size has a positive impact on the biggest cluster size for all protocols. This result is not surprising as it is well known that a graph remains connected with only a few neighbors. One can legitimately consider that increasing the view sizes is enough to prevent partitions in the presence of NATs. We show in the reminder of this section that increasing the view size is not a satisfactory solution with respect to the two other metrics, the randomness and ratio of stale entries.

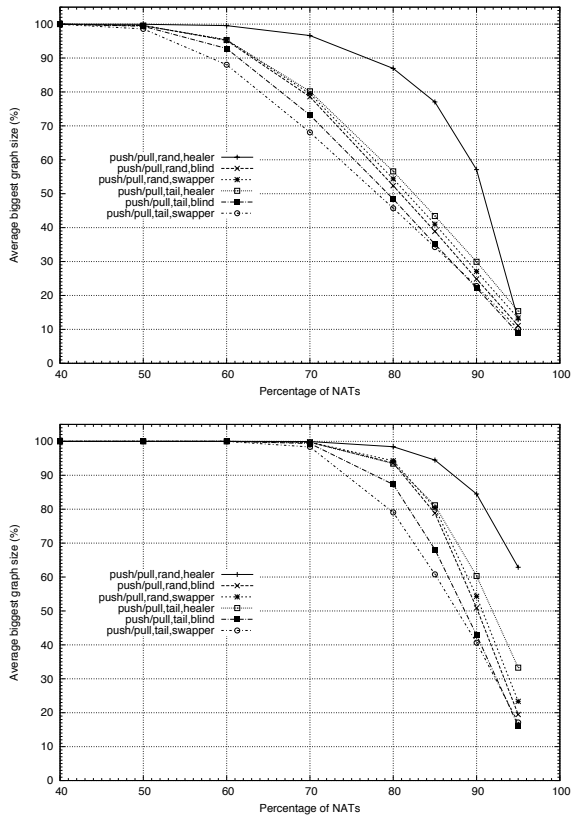

Figure 2. Size of the biggest cluster for view sizes equal to 15 (top), 27 (bottom).

Stale references. Figure 3 shows the average percentage of stale references in peer views for two different view sizes (15 and 27). A reference to a peer is said to be stale when it is not possible to communicate with this peer (due to the presence of NATs). We observe that a small proportion of natted peers suffices to cause peers to maintain stale references in their view. This percentage of stall references almost linearly grows with the percentage of natted peers. Moreover, we observe that the percentage of stale references increases when the view size increases, and that the percentage of stale references decreases for view size 15 when the percentage of NATs reaches a certain threshold (85%). These two observations can be easily explained by two facts. First, increasing the view size decreases the probability that two peers shuffle with each other twice during the lifetime of a NAT filtering rule. Second, with a large percentage of NATs and view size 15, the network starts to significantly partition in many small clusters. Consequently, two peers within a cluster have a very high probability to shuffle with each other twice during the lifetime of a NAT filtering rule.

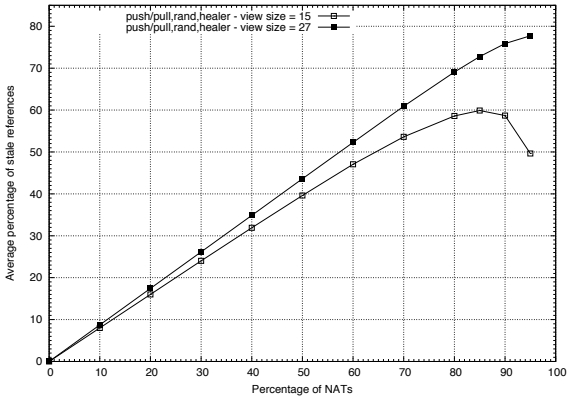

Figure 3. Percentage of stale references.

Randomness. Figure 4 shows the average ratio of non-stale references that correspond to natted peers. Again, we consider two different view sizes (15 and 27). For instance, the plot shows that with 40% of natted peers and a view of size 15, peers have on average only 10% of their non-stale references that correspond to natted peers. This typically means that 40% of the peers are sampled only 10% which is obviously a non uniform random sampling. As in Figure 3, we observe that increasing the view size negatively impacts the protocol. We also observe that when the percentage of NATs reaches a certain threshold (70%), the average ratio of non-stale references increases. The explanation is similar to the one given for Figure 3.

4 The $\mathcal{N}ylon$ protocol

In this section, we present $\mathcal{N}ylon$, a NAT-resilient gossip peer sampling protocol. A commonly used technique for traversing NATs is to use public RVPs [13, 24]. This technique could be used to build a NAT-resilient peer sampling protocol as follows: a source peer needing to communicate with a natted peer, would contact first the natted peer's public RVP to forward an OPEN_HOLE message to the target peer. This simple scheme suffers however from several

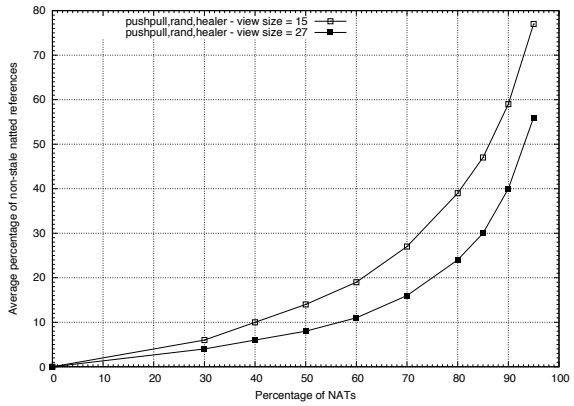

Figure 4. Ratio of non-stale references towards natted peers.

drawbacks. First, the extra load induced by the presence of NATs is supported by the public peers. This creates an uneven distribution of the load where public peers contribute much more to the protocol than natted peers. Another issue is the non uniform impact of failures of natted and public peers. A public peer's failure invalidates all references to natted peers bound to it. A possible solution is to use several RVPs for each natted peer. Nevertheless, this solution increases the bandwidth consumption.

In order to overcome the limitations imposed by using only public RVPs, we design a fully decentralized protocol that uses both natted and public peers as RVPs. Relying on natted peers for implementing RVPs is challenging: effectively, an RVP must be reachable by all peers willing to communicate with peers for which it acts as RVP. This is obviously impossible to ensure that a natted RVP will have valid filtering rules for every peer in the system. Instead, peers may rely on a routing infrastructure to send messages to any RVP in multiple hops. This is for instance what is applied in Distributed Hash Tables (DHT), where each peer in the DHT maintains valid filtering rules for the natted peers that are in its routing table. When a peer needs to communicate with an RVP, its message is routed using the DHT. Unfortunately, it is not possible to use such a routing infrastructure in the large-scale, highly dynamic, environments that we consider in this paper.

The design of $\mathcal{N}ylon$ relies on two observations:

1. A gossip protocol does not require all peers to be reachable at any time by all peers. Effectively, at a given time, the only peers a given peer might want to communicate with are those that are in its view.

2. In gossip protocols, although a peer *should* be able to communicate with any peer in its view at any time, it does not. Instead, a single peer of its view is picked upon each gossip operation. It may be the case that a peer p in the view of a peer q is removed from q's view without p and q effectively gossip with each other.

$\mathcal{N}ylon$ leverages these two observations to build NAT-resilient gossip-based peer sampling protocols in which all peers can act as RVPs. The first observation is used to implement a hole punching protocol for only a subset of the system. The second observation is used to implement a *reactive* hole punching protocol which consists in performing the actual hole punching protocol between two peers only when needed, namely when a gossip between the two peers is initiated. This avoids to systematically send an OPEN_HOLE message to all peers that p adds in its view.

The $\mathcal{N}ylon$ protocol. The main idea of $\mathcal{N}ylon$ is to implement *reactive* hole punching. Intuitively, this works as follows: a peer only performs hole punching towards peers it gossip with. Hole punching is implemented using a chain of RVPs that forward the OPEN_HOLE message until it reaches the gossip target.

The chain of RVPs is built as follows. Consider the case of a peer $n1$ shuffling with a peer $n2$. After having performed hole punching towards $n2$ (using a chain of RVPs), peer $n1$ and $n2$ can directly communicate with each other. Thus, they both become RVP for each other. Consider now that later, one of them, say $n2$, shuffles with a peer $n3$ and gives it a reference to $n1$. Before shuffling, peers $n2$ performs hole punching towards $n3$. Consequently, as between $n1$ and $n2$, peers $n2$ and $n3$ both become RVP for each other. Finally, consider that $n3$ shuffles with a peer $n4$ and gives it a reference to $n1$. A chain of RVPs has thus been created, as shown in Figure 5. This chain allows $n4$ to shuffle with peer $n1$. For this purpose, it performs hole punching towards peer $n1$ by sending an OPEN_HOLE message to $n3$ that will forward it to $n2$, that will forward it to $n1$.

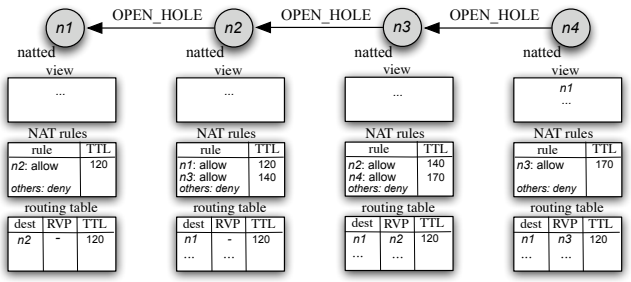

Figure 5. $\mathcal{N}ylon$ operating principle.

As illustrated in Figure 5, in addition to its view, each peer maintains a routing table. This routing table maintains the mapping between a natted peer in its view and its associated RVP. For each peer n in the routing table, the RVP is the peer it shuffled with to obtain the reference to n. RVPs

in $\mathcal{N}ylon$ are constantly changing and following the reactive flavour of $\mathcal{N}ylon$, RVPs do not proactively refresh holes. Therefore, a time to live (TTL) is associated to each RVP entry in the routing tables. TTLs are exchanged by peers together with their views and are updated every shuffling period, and every time a message from one RVP stored in the routing table is received. Note that the TTL mechanism assumes that there is a known upper bound on the latency between each pair of peers[3].

Pseudocode. The pseudocode of the $\mathcal{N}ylon$ protocol is presented in Figure 6. The basis of the protocol is the (push/pull, rand, healer) protocol presented in Section 3. The only additions to the protocol are for handling NAT traversal techniques and implementing the RVP chaining mechanism presented in the previous paragraph. The routing table code is not presented in the figure. It is abstracted in four methods. The `next_RVP()` method returns the next RVP to be used for a given destination. Note that if the destination is directly reachable (because either the destination is public or the peer acts as an RVP for the destination), the method returns the destination itself. The `update_next_RVP()` method is used to update (or create) an entry in the routing table. It is called whenever a message is received. The `update_routing_table()` method is called to update the routing table. It takes as parameter a view that has been received during a shuffle. This method adds an entry in the routing table for each entry in the view and specifies that the RVP for these entries is the peer with which the shuffle was performed. The `decrease_routing_table_ttls()` method is used to decrease the TTL of routing table entries, and purge the expired ones.

```
1  every shuffling_period units do
2    target ← select_gossip_destination(view)
3    if (target is public
       or next_RVP(target) = target) then
4      send ⟨REQUEST, view, self, target⟩ to target
5    elif ((target is SYM and self is PRC)
       or self is SYM) then
6      // Use relaying
7      send ⟨REQUEST, view, self, target⟩
         to next_RVP(target)
8    else
9      // Hole punching
10     send ⟨OPEN_HOLE, self, target⟩ to next_RVP(target)
11     if self is not public then
12       send ⟨PING⟩ to target
13   increase_view_age()
14   decrease_routing_table_ttls()

15 on receive ⟨REQUEST, view_s, src, dest⟩ from p do
16   update_next_RVP(p,p,HOLE_TIMEOUT)
17   if dest ≠ self then
18     // Forwarding
19     send ⟨REQUEST, view_s, src, dest⟩ to next_RVP(dest)
20   elif (src is SYM and self ≠ public)
       or (self is SYM and src ≠ public) then
21     // Use relaying
22     send ⟨RESPONSE, view, src⟩ to next_RVP(src)
23   else
24     send ⟨RESPONSE, view, src⟩ to src
25     view ← merge_and_truncate(view, view_s)
26     update_routing_table(view)

27 on receive ⟨RESPONSE, view_t, dest⟩ from p do
28   update_next_RVP(p,p,HOLE_TIMEOUT)
29   if dest ≠ self then
30     // Forwarding
31     send ⟨RESPONSE, view, dest⟩ to next_RVP(dest)
32   else
33     view ← merge_and_truncate(view, view_t)
34     update_routing_table(view)

35 on receive ⟨OPEN_HOLE, src, dest⟩ from p do
36   update_next_RVP(p,p,HOLE_TIMEOUT)
37   if dest = self then
38     send ⟨PONG⟩ to src
39   else
40     send ⟨OPEN_HOLE, src, dest⟩ to next_RVP(dest)

41 on receive ⟨PING⟩ from p do
42   update_next_RVP(p,p,HOLE_TIMEOUT)
43   send ⟨PONG⟩ to src

44 on receive ⟨PONG⟩ from p do
45   update_next_RVP(p,p,HOLE_TIMEOUT)
46   send ⟨REQUEST, view, self, p⟩ to p
```

Figure 6. The $\mathcal{N}ylon$ protocol.

5 Evaluation

In this section, we report the results of the evaluation of the $\mathcal{N}ylon$ protocol. We simulated a system of 10,000 peers and varied the percentage of peers sitting behind NATs. In short, we show that (i) it achieves uniform random peer sampling, (ii) it induces a reasonable overhead and homogeneously balances the load among natted and public peers, (iii) it achieves reasonable latency, and (iv) it is highly resilient to churn. Before describing these results in more detail, we first present the experimental setup.

Experimental settings. To the best of our knowledge, existing p2p simulators do not take into account NATs. We thus developed a Java-based, event-driven simulator that

[3] If the upper bound is not met, this could cause an entry in the routing table to be stale. We show in Section 5 that the protocol resists to the simultaneous departure of 50% of the nodes. This shows that the protocol would resist to half of the message exchanges simultaneously exceeding the upper bound.

takes into account the four kinds of NATs described in Section 2. Message latency was set to $50ms$, the hole timeout was set to $90s$ (a typical vendor value), and the shuffling period was set to $5s$. Experiments were conducted on a 10,000 peers system. Although we experimented with all four kinds of NATs, experiments with FC NAT are not reported. In practice, as explained in Section 2, peers behind FC NATs behave similarly to public peers as long as they frequently send or receive messages. The distribution we used is the following: 50% of RC NATs, 40% of PRC NATs, and 10% of SYM NATs. Note that we evaluated other distributions and got comparable results. Peers were initialized with a view composed of a random set of public peers to ensure connectivity at the start of each experiment. Unless explicitly mentioned otherwise, the view size is set to 15. All experiments were run with 30 different seeds, the results reported are the average of those 30 runs. Finally, experiments lasted a long enough time to observe, most of the time, a negligible variance. However, any non negligible observed variance is indicated in the graphs.

Correctness. We assessed the correctness of $\mathcal{N}ylon$ with different experiments. Due to space limitation, we do not show graphs for these experiments. First, we checked that there were no network partitions and no stale references in peer views. Moreover, we assessed randomness using the *diehard* test suite for random number generators [16].

Network bandwidth usage. We made experiments to assess the bandwidth usage of $\mathcal{N}ylon$. We computed the average number of bytes per second that each peer sends and receives as a function of the percentage of NATs. Results are depicted in Figure 7. $\mathcal{N}ylon$ consumes less than 350B/s. For comparison, we plotted the average number of bytes per second consumed by the (push/pull, rand, healer) configuration (line "reference"). We also observe that the bandwidth usage does not evolve linearly with the number of NATs. This comes from the fact that the length of RVP chains do not evolve linearly with the number of NATs (see next section).

As explained in Section 4, one of the objectives of $\mathcal{N}ylon$ is to ensure that all peers contribute almost equally to the protocol[4]. This is reflected in Figure 8 which shows the average number of bytes per second sent and received by public and natted peers. We observe that public peers send and receive between 10% and 20% less bytes per second than natted peers. This comes from the fact that (i) all peers can act as RVP, and (ii) public peers do not receive OPEN_HOLE messages for themselves and do not send PONG messages.

Latency. The latency is expressed in the number of hops required for a peer to establish a message exchange with the peer it selected for shuffling. The latency towards public

[4]The only exception being that messages sent and received by peers sitting behind SYM NATs must be relayed by public peers.

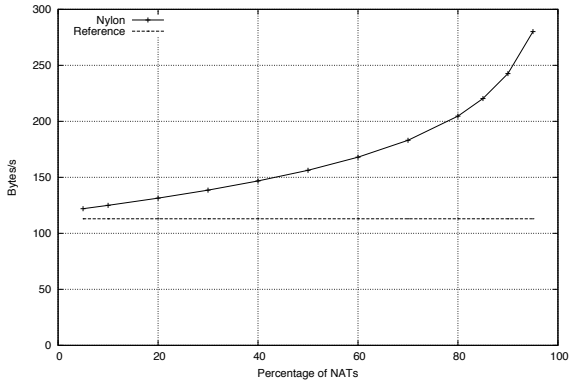

Figure 7. Average number of Bytes/s sent and received by a peer.

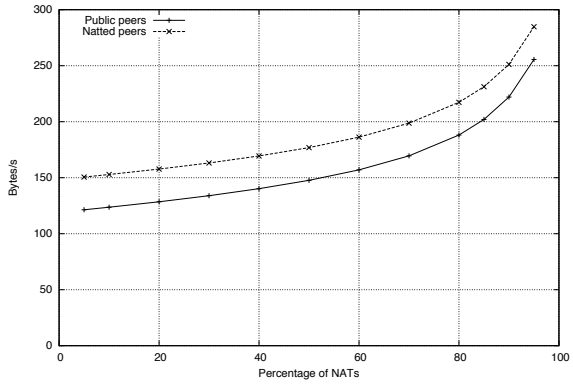

Figure 8. Average number of Bytes/s sent and received by public and natted peers.

peers is obviously equal to one hop. Regarding natted destinations, the protocol requires sending one PING and one PONG message. The main factor impacting latency is the length of the RVP chain used to send the OPEN_HOLE message. Figure 9 shows the average length of RVP chains with two different view sizes (15 and 27). Not surprisingly, we observe that the number of RVPs increases with the percentage of NATs. Note that this increase is not linear, which explains the non-linear bandwidth usage observed for $\mathcal{N}ylon$ in Figure 7. With a view size of 15, the RVP chain length ranges from 1 (with 10% of NAT) to 3. The average relaying latency of $\mathcal{N}ylon$ is thus smaller than 4 hops, which is very reasonable. The fact that the length of RVP chain is small limits the TTL expiration. Finally, an interesting observation is that the average RVP chain length decreases when the view size increases. This result is consistent with random graph theory results on the average distance between peers in a graph as a function of their in and out degree [5].

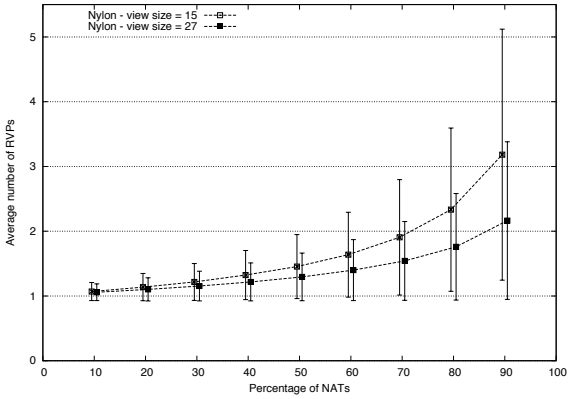

Figure 9. Average number of RVPs towards a natted destination.

Churn resiliency. We conclude this section by an analysis of the behavior of $\mathcal{N}ylon$ under massive churn. The experiments consisted in removing a varying fraction of peers after each of them had performed 500 shuffles. Public and natted peers were removed proportionally to their number in the system. We present results in Figure 10. The different bar types correspond to different percentages of NATs. On the X axis is represented the percentage of peers that are leaving the system. The Y axis represent the size of the biggest cluster 1500 shuffles after the start of the massive churn. We observe that $\mathcal{N}ylon$ is highly resilient to churn. It tolerates the departure of 50% of the peers without partitioning. Even with higher percentage, it exhibits very good performance. This result can be explained by the fact that each peer can be reached by different chains of RVPs at the same time.

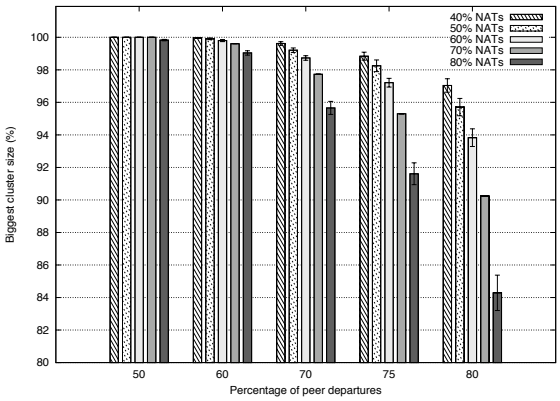

Figure 10. Impact of massive churn on the size of the biggest cluster.

6 Related works and conclusion

Several systems have tried to overcome the problem of limited connectivity [15, 20, 14]. All these systems rely on an explicit structure to route messages on top of a gossip protocol. They use proactive mechanisms to ensure that communication between natted peers is possible under the implicit assumption that the network is fairly static. Some works have also been done in the context of Distributed Hash Tables (DHTs) [13, 22]. Traversing NATs in such systems can be achieved provided that each peer has a relatively static set of neighbors. In addition the structure of DHTs can be used as a natural vector to assign public peers to natted peers. Let us also note that there exist protocols allowing the creation of permanent NAT filtering rules: NAT-PMP [**?**] and UPnP [**?**]. These protocols could be used in gossip protocols to avoid the problems caused by the presence of NAT devices. Unfortunately these protocols have limitations. First, they are not supported by all NAT devices. Second, they pose security issues since any application running on a peer can open ports on the NAT device without any approval or notification to the node's user. Finally, some works have also been done at the network level. For instance, [8] proposes an extension to the routing process of IPv4 in order to take into account NAT devices. Nevertheless, the proposed architecture requires modifications to NAT devices and to end hosts.

While taking into account NATs can be achieved in fairly static systems, this is challenging in the context of highly dynamic systems. In this paper, we have proposed $\mathcal{N}ylon$, a fully decentralized NAT-resilient gossip peer sampling protocol. $\mathcal{N}ylon$ leverages the fact that in a gossip protocol each peer only needs to communicate with a subset of peers contained in its view and does actually communicate with an even smaller subset of the peers. It uses a *reac-*

tive hole punching protocol, which creates a path of relay peers to setup communications. Experiments have shown that $\mathcal{N}ylon$ accommodates a large proportion of NATs without impacting the properties of the peer sampling. Moreover, $\mathcal{N}ylon$ evenly spreads the overhead induced by NATs between public and natted peers and is highly resilient to churn.

References

[1] Peersim simulator. http://peersim.sf.net, 2008.

[2] Stunt. http://nutss.gforge.cis.cornell.edu/stunt.php,2008.

[3] 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, 1999.

[4] M. Casado and M. J. Freedman. Peering through the shroud: The effect of edge opacity on ip-based client identification. In *NSDI*, 2007.

[5] F. Chung, F. Chung, F. Chung, L. Lu, and L. Lu. The average distances in random graphs with given expected degrees. *Internet Mathematics*, 99:15879–15882, 2002.

[6] N. Drost, E. Ogston, R. van Nieuwpoort, and H. E. Bal. Arrg: real-world gossiping. In C. Kesselman, J. Dongarra, and D. W. Walker, editors, *HPDC*, pages 147–158. ACM, 2007.

[7] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-peer communication across network address translators. *CoRR*, abs/cs/0603074, 2006.

[8] P. Francis and R. Gummadi. Ipnl: A nat-extended internet architecture. *SIGCOMM Comput. Commun. Rev.*, 31(4):69–80, 2001.

[9] M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-transitive connectivity and DHTs. San Francisco, CA, Dec. 2005.

[10] M. Jelasity, A. Montresor, and Ö. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, 2005.

[11] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3), 2007.

[12] V. King and J. Saia. Choosing a random peer. In *PODC*, pages 125–130, 2004.

[13] M. Lee, H. Choi, and S. Park. Donet-p: A streaming overlay network protocol with private network support. *TENCON 2007 - 2007 IEEE Region 10 Conference*, pages 1–4, 30 2007-Nov. 2 2007.

[14] M.-J. Lin and K. Marzullo. Directional gossip: Gossip in a wide area network. In *EDCC-3: Proceedings of the Third European Dependable Computing Conference on Dependable Computing*, pages 364–379, London, UK, 1999. Springer-Verlag.

[15] J. Maassen and H. E. Bal. Smartsockets: solving the connectivity problems in grid computing. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 1–10, New York, NY, USA, 2007. ACM.

[16] G. Marsaglia and W. W. Tsang. Some difficult-to-pass tests of randomness. *Journal of Statistical Software*, 7(3):1–9, 1 2002.

[17] L. Massoulié, E. L. Merrer, A.-M. Kermarrec, and A. J. Ganesh. Peer counting and sampling in overlay networks: random walk methods. In *PODC*, pages 123–132, 2006.

[18] S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, and D. Chalmers. The state of peer-to-peer simulators and simulations. *SIGCOMM Comput. Commun. Rev.*, 37(2):95–98, April 2007.

[19] D. K. P. Srisuresh, B. Ford. State of peer-to-peer (p2p) communication across network address translators (nats). RFC 5128, 2008.

[20] R. V. Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.

[21] R. V. Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. Technical report, Ithaca, NY, USA, 1998.

[22] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 73–86, New York, NY, USA, 2002. ACM.

[23] 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.

[24] H. Yoshimi, N. Enomoto, Z. Cui, K. Takagi, and A. Iwata. Nat traversal technology of reducing load on relaying server for p2p connections. *Consumer Communications and Networking Conference, 2007. CCNC 2007. 4th IEEE*, pages 100–104, Jan. 2007.