Gozar: NAT-friendly Peer Sampling with One-Hop Distributed NAT Traversal

Amir H. Payberah[1,2], Jim Dowling[1], and Seif Haridi[1,2]

[1] Swedish Institute of Computer Science (SICS)
[2] KTH - Royal Institute of Technology

Abstract. Gossip-based peer sampling protocols have been widely used as a building block for many large-scale distributed applications. However, Network Address Translation gateways (NATs) cause most existing gossiping protocols to break down, as nodes cannot establish direct connections to nodes behind NATs (private nodes). In addition, most of the existing NAT traversal algorithms for establishing connectivity to private nodes rely on third party servers running at a well-known, public IP addresses. In this paper, we present *Gozar*, a gossip-based peer sampling service that: (i) provides uniform random samples in the presence of NATs, and (ii) enables direct connectivity to sampled nodes using a fully distributed NAT traversal service, where connection messages require only a single hop to connect to private nodes. We show in simulation that Gozar preserves the randomness properties of a gossip-based peer sampling service. We show the robustness of Gozar when a large fraction of nodes reside behind NATs and also in catastrophic failure scenarios. For example, if 80% of nodes are behind NATs, and 80% of the nodes fail, more than 92% of the remaining nodes stay connected. In addition, we compare Gozar with existing NAT-friendly gossip-based peer sampling services, Nylon and ARRG. We show that Gozar is the only system that supports one-hop NAT traversal, and its overhead is roughly half of Nylon's.

1 Introduction

Peer sampling services have been widely used in large scale distributed applications, such as information dissemination [7], aggregation [17], and overlay topology management [14, 28]. A peer sampling service (PSS) periodically provides a node 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 [15]. The sampled nodes are stored in a *partial view* that consists of a set of node descriptors, which are updated periodically by the PSS.

Gossiping algorithms are the most common approach to implementing a PSS [29, 9, 16]. Gossip-based PSS' can ensure that node descriptors are distributed uniformly at random over all partial views [18]. However, in the Internet, where a high percentage of nodes are behind NATs, these traditional gossip-based PSS' become biased. Nodes cannot establish direct connections to nodes behind NATs (*private nodes*), and private nodes become under-represented in partial views, while nodes that do support direct connectivity, *public nodes*, become over-represented in partial views [19].

The ability to establish direct connectivity with private nodes, using NAT traversal algorithms, has traditionally not been considered by gossip-based PSS'. However, as

nodes are typically sampled from a PSS in order to connect to them, there are natural benefits to including NAT traversal as part of a PSS. Nylon [19] was the first system to present a distributed solution to NAT traversal that uses existing nodes in the PSS to help in NAT traversal. Nylon uses nodes that have successfully established a connection to a private node as partners who will both route messages to the private node (through its NAT) and coordinate NAT *hole punching* algorithms [8, 19]. As node descriptors spread in the system through gossiping, this creates routing table entries for paths that forward packets to private nodes. However, long routing paths increase both network traffic at intermediary nodes and the routing latency to private nodes. Also, routing paths become fragile when nodes frequently join and leave the system (*churn*). Finally, hole punching is slow and can take up to a few seconds over the Internet [27].

This paper introduces *Gozar*, a gossip-based peer sampling service that (i) provides uniform random samples in the presence of NATs, and (ii) enables direct connectivity to sampled nodes by providing a distributed NAT traversal service that requires only a single intermediary hop to connect to a private node. Gozar uses public nodes as both *relay servers* [13] (to forward messages to private nodes) and *rendezvous servers* [8] (to establish direct connections with private nodes using hole punching algorithms).

Relaying and hole punching is enabled by private nodes finding public nodes who will act as both relay and rendezvous *partners* for them. For load balancing and fairness, public nodes accept only a small bounded number of private nodes as partners. When references to private nodes are gossiped in the PSS or sampled using the PSS, they include the addresses of their partner nodes. A node, then, can use these partners to either (i) gossip with a private node by relaying or (ii) establish a direct connection with the private node by using the partner for hole punching. We favour relaying over hole punching when gossiping with private nodes due to the low connection setup time compared to hole punching and also because the messages involved are small and introduce negligible overhead to public nodes. However, the hole punching service can be used by clients of the PSS to establish a direct connection with a sampled private node. NAT hole punching is typically required by applications such as video-on-demand [2] and live streaming [22, 23], where relaying would introduce too much overhead on public nodes.

A private node may have several redundant partners. Although redundancy introduces some extra overhead on public nodes, it also reduces latency when performing NAT traversal, as parallel connection requests can be sent to several partners, with the end-to-end connection latency being the fastest of the partners to complete NAT traversal. In this way, a more reliable NAT traversal service can be built over more unreliable connection latencies, such as those widely seen on the Internet.

We evaluate Gozar in simulation and show how its PSS maintains its randomness property even in networks containing large fractions of NATs. We validate its behaviour through comparison with the widely used Cyclon protocol [29] (which does not support networks containing NATs). We also compare the performance of Gozar with the only other NAT-friendly PSS' we found in the literature, Nylon [19] and ARRG [4], and show how Gozar has less protocol overhead compared to Nylon and ARRG, and is the only NAT-friendly peer sampling system that supports one hop NAT traversal.

2 Related work

Dan Kegel explored STUN [26] as a UDP hole punching solution for NAT traversal, and Guha et al. extended it to TCP by introducing STUNT [10]. However, studies [8, 10] show that NAT hole punching fails 10-15% of the time for UDP and 30-40% of the time for TCP traffic. TURN [13] was an alternative solution for NAT traversal using relay nodes that works for all nodes that can establish an outbound connection. Interactive connectivity establishment (ICE) [25] has been introduced as a more general technique for NAT traversal for media streams that makes use of both STUN [26] and TURN [13]. All these techniques rely on third party servers running at well-known addresses.

Kermarrec et al. introduce in Nylon [19] a distributed NAT traversal technique that uses all existing nodes in the system (both private and public nodes) as rendezvous servers (RVPs). In Nylon, two nodes become the RVP of each other whenever they exchange their views. Later, if a node selects a private node for gossip exchange, it opens a direct connection to the private node using a chain of RVPs for hole punching. The chains of RVPs in Nylon are unbounded in length, making Nylon fragile in dynamic networks, and increasing traffic at intermediary nodes.

ARRG [4] supports gossip-based peer sampling in the presence of NATs without an explicit solution for traversing NATs. In ARRG, each node maintains an open list of nodes with whom it has had a successful gossip exchange in the past. When a node view exchange fails, it selects a different node from this open list. The open list, however, biases the PSS, since the nodes in the open list are selected more frequently for gossiping.

Renesse et. al [20] present an approach to fairly distribute relay traffic over public nodes in a NAT-friendly gossiping system. In their system, which is not a PSS, each node accepts exchange requests as much as it initiates view exchanges. Similar to Nylon, they use chains of nodes as relay servers.

In [5], D'Acunto et. al introduce an analytical model to show the impact of NATs on P2P swarming systems, and in [21] Liu and Pan analyse the performance of bittorrent-like systems in private networks. They show how the fraction of private nodes affects the download speed and download time of a P2P file-sharing system. Moreover, authors in [6] and [27] study the characteristics of existing NAT devices on the Internet, and show the success rate, on the Internet, of NAT traversal algorithms for different NAT types. In addition, the distribution of NAT rule timeouts for NAT devices on the Internet is described in [6], and in [24] an algorithm is presented, based on binary search, to adapt the time required to refresh NAT rules to prevent timeouts.

3 Background

In gossip-based PSS', protocol execution at each node is divided into periodic cycles [18]. In each cycle, every node selects a node from its partial view to exchange a subset of its partial view with the selected node. Both nodes subsequently update their partial views using the received node descriptors. Implementations vary based on a number of different policies in node selection (rand, tail), view exchange (push, push-pull) and view selection (blind, heale, swapper) [18].

In a PSS, the sampled nodes should follow a uniform random distribution. To ensure randomness of a partial view in an overlay network, the overlay constructed by a peer sampling protocol should ensure that *indegree distribution*, *average shortest path* and *clustering coefficient*, are close to a random network [18, 29]. Kermarrec et al. evaluated the impact of NATs on traditional gossip-based PSS' in [19]. They showed that the network becomes partitioned when the number of private nodes exceeds a certain threshold. The larger the view size is, the higher the threshold for partitioning is. However, increasing the nodes' view size increases the number of stale node descriptors in views, which, in turn, biases the peer sampling.

There are two general techniques that are used to communicate with private nodes: (i) *hole punching* [8, 12] can be used to establish direct connections that traverse the private node's NAT, and (ii) *relaying* [13] can be used to send a message to a private node via a third party relay node that already has an established connection with the private node. In general, hole punching is preferable when large amounts of traffic will be sent between the two nodes and when slow connection setup times are not a problem. Relaying is preferable when the connection setup time should be short (typically less than one second) and small amounts of data will be sent over the connection.

In principle, existing PSS' could be adapted to work over NATs. This can be done by having all nodes run a protocol to identify their NAT type, such as STUN [26]. Then, nodes identified as private keep open a connection to a third party rendezvous server. When a node wishes to gossip with a private node, it can request a connection to the private node via the rendezvous server. The rendezvous server then executes a hole punching technique to establish a direct connection between the two nodes. Aside from the inherently centralized nature of this approach, other problems include the success rate of NAT hole punching for UDP is only 85-90% [8, 10], and the time taken to establish a direct connection using hole punching protocols is high and has high variance (averaging between 700ms and 1100ms on the open Internet for the company Peerialism within Sweden [27]). This high and unpredictable NAT traversal time of hole punching is the main reason why Gozar uses relaying when gossiping.

4 Problem description

The problem Gozar addresses is how to design a gossip-based NAT-friendly PSS that also supports distributed NAT traversal using a system composed of both public and private nodes. The challenge with gossiping is that it assumes a node can communicate with any node selected from its partial view. To communicate with a private node, there are three existing options:

1. Relay communications to the private node using a public relay node,
2. Use a NAT hole-punching algorithm to establish a direct connection to the private node using a public rendezvous node,
3. Route the request to the private node using chains of existing open connections.

For the first two options, we assume that private nodes are assigned to different public nodes that act as relay or rendezvous servers. This leads to the problem of discovering which public nodes act as partners for the private nodes. A similar problem arises for

the third option - if we are to route a request to a private node along a chain of open connections, how do we maintain routing tables with entries for all reachable private nodes. When designing a gossiping system, we have to decide on which option(s) to support for communicating with private nodes. There are several factors to consider. How much data will be sent over the connection? How long lived will the connection be? How sensitive is the system to high and variable latencies in establishing connections? How fairly should the gossiping load be distributed over public versus private nodes?

For large amounts of data traffic, the second option of hole-punching is the only really viable option, if one is to preserve fairness. However, if a system is sensitive to long connection establishment times, then hole-punching may not be suitable. If the amount of data being sent is small, and fast connection setup times are important, then relaying is considered an acceptable solution. If it is important to distribute load as fairly as possible between public and private nodes, then option 3 is attractive. In existing systems, it appears that Skype supports both options 1 and 2, and can considered to have a solution to the fairness problem that, by virtue of its widespread adoption, can be considered acceptable to their user community [3].

5 The Gozar protocol

Gozar is a NAT-friendly gossip-based peer sampling protocol with support for distributed NAT traversal. Our implementation of Gozar is based on the *tail*, *push-pull* and *swapper* policies for node selection, view exchange and view selection, respectively [18] (although we also run experiments, ommitted here for brevity, showing that Gozar also works with different policies introduced in [18]).

In Gozar, node descriptors are augmented with the node's NAT type (private or public) and the mapping, assignment and filtering policies determined for the NAT [27]. A STUN-like protocol is run on a bootstrap server when a node joins the system to determine its NAT type and policies. We consider running STUN once at bootstrap time acceptable, as, although some corporate NAT devices can change their NAT policies dynamically, the vast majority of consumer NAT devices have a fixed NAT type and fixed policies.

In Gozar, each private node connects to one or more public nodes, called *partners*. Private nodes discover potential partners using the PSS, that is, private nodes select public nodes from their partial view and send $partnering$ requests to them. When a private node successfully partners with a public node, it adds its partner address to its own node descriptor. As node descriptors spread in the system through gossiping, a node that subsequently selects the private node from its partial view communicates with the private node using one of its partners as a relay server. Relaying enables faster connection establishment than hole punching, allowing for shorter periodic cycles for gossiping. Short gossiping cycles are necessary in dynamic networks, as they improve convergence time, helping keep partial views updated in a timely manner.

However, for distributed applications that use a PSS, such as online gaming, video streaming, and P2P file sharing, relaying is not acceptable due to the extra load on public nodes. To support these applications, the private nodes' partners also provide a

rendezvous service to enable applications that sample nodes using the PSS to connect to them using a hole punching algorithm (if hole punching is possible).

5.1 Partnering

Whenever a new node joins the system, it contacts the *bootstrap server* and asks for a list of nodes from the system and also runs the modified STUN protocol to determine its NAT type and policies. If the node is public, it can immediately add the returned nodes to its partial view and start gossiping with the returned nodes. If the node is private, it needs to find a partner before it can start gossiping. It selects m public nodes from the returned nodes and sends each of them a $partnering$ request. Public nodes only partner a bounded number of private nodes to ensure the partnering load is balanced over the public nodes. Therefore, if a public node cannot act as a partner, it returns a NACK. The private node continues sending $partnering$ requests to public nodes until it finds a partner, upon which the private node can now start gossiping. Private nodes proactively keep their connections to their partners open by sending *ping* messages to them periodically. Authors in [6] showed that unused NAT mapping rules remain valid for more than 120 seconds for 70% of connections. In our implementation, the private nodes send the ping messages every 50 seconds to refresh a higher percentage of mapping rules. Moreover, private nodes use the ping replies to detect the failure of their partners. If a private node detects a failed partner, it restarts the partner discovery process.

5.2 Peer sampling service

Each node in Gozar maintains a partial view of the nodes in the system. A node descriptor, stored in a partial view, contains the address of the node, NAT type, and the addresses of the node's partners, which are initially empty. When a node descriptor is gossiped or sampled, other nodes learn about the node's NAT type and any partners. Later on, a node can gossip with a private node by relaying messages through the private node's partners.

Each node p periodically executes algorithm 1 to exchange and update its view. The algorithm shows that in each iteration, p first updates the age of all nodes in its view, and then chooses a node to exchange its view with. After selecting a node q, p removes that node from its view. Node p, then, selects a subset of random nodes from its view, and appends to the subset its own node descriptor (the node, its NAT type, and its partners). If the selected node q is a public node, then p sends the *shuffle request* message directly to q, otherwise it sends the *shuffle request* as a *relay message* to one of q's partners, selected uniformly at random.

Algorithm 2 shows how a node p selects another node to exchange its view with. Node p selects the oldest node in its view (the tail policy), which is either a public node, or a private node that has at least one partner.

Algorithm 3 is triggered whenever a node receives a shuffle request message. Once node q receives the shuffle request, it selects a random subset of node descriptors from its view and sends the subset back to the requester node p. If p is a public node, q sends the *shuffle response* back directly to it, otherwise it uses one of p's partners to relay

Algorithm 1 Shuffle view.

```
1:  procedure ShuffleView ⟨this⟩
2:      this.view.updateAge()
3:      q ← SelectANodeToShuffleWith(this.view)                ▷ See algorithm 2
4:      this.view.remove(q)
5:      pView ← this.view.subset()                             ▷ a random subset from p's view
6:      pView.add(p, p.natType, p.partners)
7:      if q.natType is public then
8:          Send ShuffleRequest(pView, p) to q
9:      else
10:         qPartner ← random partner from q.partners
11:         Send Relay(shuffleRequest, pView, q) to qPartner
12:     end if
13: end procedure
```

Algorithm 2 Select a node to shuffle with.

```
1:  procedure SelectANodeToShuffleWith ⟨this.view⟩
2:      for all node_i in this.view do
3:          if node_i.natType = public OR (node_i.natType = private AND node_i.partners ≠ ∅) then
4:              candidates ← node_i
5:          end if
6:      end for
7:      q ← oldest node from candidates
8:      Return q
9:  end procedure
```

Algorithm 3 Handling the shuffle request.

```
1:  upon event ⟨ShuffleRequest | pView, p⟩ from m              ▷ m can be p or q.partner
2:      qView ← this.view.subset()                             ▷ a random subset from q's view
3:      if p.natType is public then
4:          Send ShuffleResponse(qView, q) to p
5:      else
6:          pPartner ← random partner from p.partners
7:          Send Relay(shuffleResponse, qView, p) to pPartner
8:      end if
9:      UpdateView(qView, pView)
10: end event
```

Algorithm 4 Handling the shuffle response.

```
1:  upon event ⟨ShuffleResponse | qView, q⟩ from n             ▷ n can be q or p.partner
2:      UpdateView(pView, qView)
3:  end event
```

Algorithm 5 Updating the view.

```
1:  procedure UpdateView ⟨sentView, receivedView⟩
2:      for all node_i in receivedView do
3:          if this.view.contains(node_i) then
4:              this.view.updateAge(node_i)
5:          else if this.view has free entries then
6:              this.view.add(node_i)
7:          else
8:              node_j ← sentView.poll()
9:              this.view.remove(node_j)
10:             this.view.add(node_i)
11:         end if
12:     end for
13: end procedure
```

Algorithm 6 Handling the relay message.

```
1: upon event ⟨RELAY | natType, view, y⟩ from x
2:     if natType is shuffleRequest then
3:         Send ShuffleRequest(view, x) to y
4:     else
5:         Send ShuffleResponse(view, x) to y
6:     end if
7: end event
```

Algorithm 7 NAT Traversal to private nodes.

```
1: procedure SendData ⟨q, data⟩
2:     if q.natType is public then
3:         Send data to q
4:     else
5:         RVP ← random partner from q.partners
6:                                  ▷ Determine hole punching algorithm for the combination of NAT types
7:         hp ← hpAlgorithm(p.natType, q.natType)
8:                                  ▷ Start hole punching at RVP using the hole punching algorithm hp.
9:         holePunch(hp, p, q, RVP)
10:        Send data to q
11:    end if
12: end procedure
```

the response. Again, node q selects p's relaying node uniformly at random from the list of p's partners. Finally, node q updates its view. A node updates its view whenever it receives a shuffle response (algorithm 4).

Algorithm 5 shows how a node updates its view using the received list of node descriptors. Node p merges the node descriptors received from q with its current view by iterating through the received list, and adding the descriptors to its own view. If its view is not full, it adds the node, and if a node descriptor to be merged already exists in p's view, p updates its age (if more recent). If the view is full, p replaces one of the nodes it had sent to q with the node in received list (the swapper policy).

Algorithm 6 is triggered whenever a partner node receives a relay message from another node. The node extracts the embedded message that can be a shuffle request or shuffle response, and forwards it to the destination private node.

If a client of the PSS, node p, wants to establish a direct connection to a node q, it uses algorithm 7 that implements the hole punching service. Algorithm 7 shows that if q is a public node, then p sends data directly to q. Otherwise, p selects uniformly at random one of q's partners as a rendezvous node (RVP), and determines the hole punching algorithm (hp) using the combination of its own NAT type and q's NAT type RVP [27]. Then, p starts the hole punching process through the RVP [27]. After successfully establishing a direct connection, node p sends data directly to q.

6 Evaluation

In this section, we compare in simulation the behavior of Gozar with Nylon [19] and ARRG [4], the only two other NAT-friendly gossip-based PSS' we found in the literature. In our experiments, we use Cyclon as a baseline for comparison, where Cyclon

experiments are executed using only public nodes. Cyclon has shown in simulation that it passes classical tests for randomness [29].

6.1 Experiment setup

We implemented Gozar, Cyclon, Nylon and ARRG on the Kompics platform [1]. Kompics provides a framework for building P2P protocols and a discrete event simulator for simulating them using different bandwidth, latency and churn models. Our implementations of Cyclon, Nylon and ARRG are based on the system descriptions in [29], [19] and [4], respectively. Nylon differs from Gozar in its node selection and view merging policies: Gozar uses tail and swapper policies, while Nylon uses rand and healer policies [19]. For a cleaner comparison with the NAT-friendly features of Nylon, we use the tail and swapper policies in our implementation of Nylon.

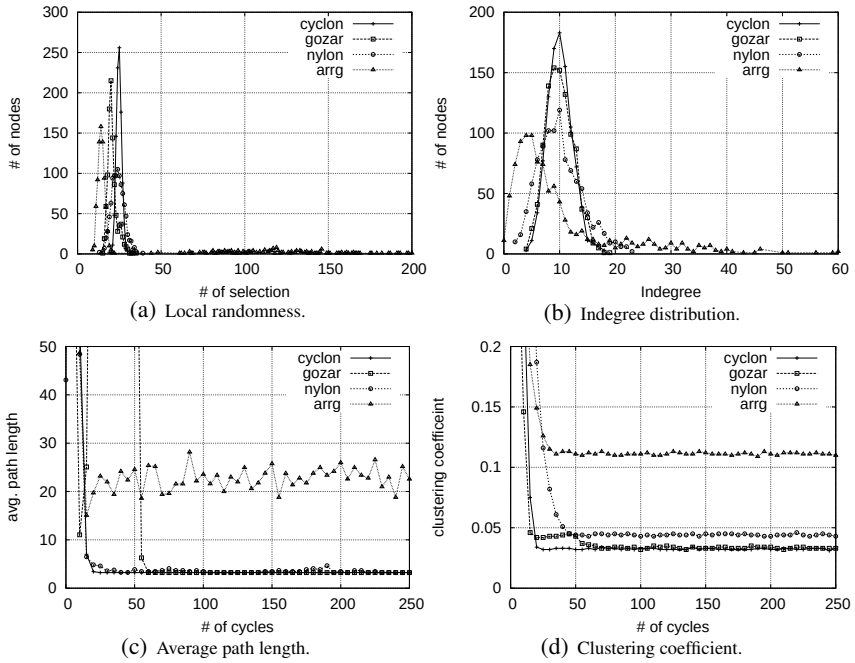
In our experimental setup, for all four systems, the size of a node's partial view is 10, and the size of subset of the partial view sent in each view exchange is 5. The iteration period for view exchange is set to one second. Latencies between nodes are modelled on Internet latencies, using a latency map based on the King data-set [11]. In all simulations, 1000 nodes join the system following a Poisson distribution with an inter-arrival time of 10 milliseconds, and unless stated otherwise, 80% of nodes are behind NATs. In Gozar, each private node has 3 public nodes as partners, and they keep a connection to their partners open by sending ping messages every 50 seconds.

The experiment scenarios presented here are a comparison of the randomness of Gozar with Cyclon, Nylon and ARRG; a comparison of the protocol overhead of Gozar and Nylon for different percentages of private nodes, and finally, we evaluate the behaviour of Gozar in dynamic networks.

6.2 Randomness

Here, we compare the randomness of the PSS' of Gozar with Nylon and ARRG. Cyclon is used as a baseline for true randomness. In the first experiment, we measure the *local randomness* property [18] of these systems. Local randomness shows the number of times that each node in the system is returned by the PSS for each node in the system. For a truly random PSS, we expect that the returned nodes follow a uniform random distribution. In figure 1(a), we measure the local randomness of all nodes in the system, after 250 cycles. For a uniform random distribution, the expected number of selections for each node is 25. As we can see, Cyclon has an almost uniform random distribution, while Nylon's distribution is slightly closer to uniform random than Gozar's distribution. ARRG, on the other hand, has a long-tailed distribution, where there are a few nodes that are sampled many times (the public nodes stored in private nodes' caches [4]). For Gozar, we can see two spikes: one representing the private nodes, that is roughly four times higher than the other consisting of the public nodes. This slight skew in the distribution results from the fact that public nodes are more likely to be selected during the first few cycles when private nodes have no partners.

In addition to the local randomness property, we use the *global randomness* metrics, defined in [18], to capture important global correlations of the system as a whole.



(a) Local randomness. (b) Indegree distribution.

(c) Average path length. (d) Clustering coefficient.

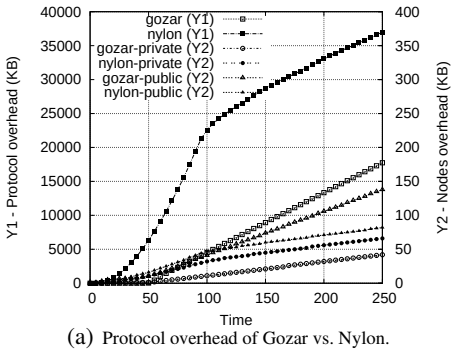
Fig. 1. Randomness properties.

The global randomness metrics are based on graph theoretical properties of the system, including the *indegree distribution*, *average path length* and *clustering coefficient*.

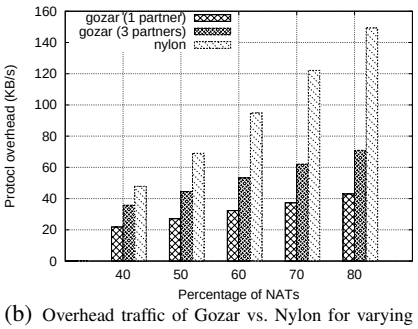
Figure 1(b) shows the indegree distribution of nodes after 250 cycles (the out-degree of all nodes is 10). In a uniformly random system, we expect that the indegree is distributed uniformly among all nodes. Cyclon shows this behaviour as the node indegree is almost distributed uniformly among nodes. We can see the same distribution in Gozar and Nylon - their indegree distributions are very close to Cyclon. Again, due to high number of unsuccessful view exchanges in ARRG, we see that the node indegree is highly skewed.

In figure 1(c), we compare the average path length of the three systems, with Cyclon as a baseline. The path length for two nodes is measured as the minimum number of hops between two nodes, and the average path length is the average of all path lengths between all nodes in the system. Figure 1(c) also shows the average path length for the system in different cycles. Here, we can see the average path length of Gozar and Nylon track Cyclon very closely, but ARRG has higher average path length. As we can see, in the first few cycles, the path length of Gozar is high but after passing 50 cycles (50 seconds), the path length decreases. That is because of the time that private nodes need to find their partners and add them to their node descriptors.

Finally, we compare the clustering coefficient of the systems. The clustering coefficient of a node is the number of links between the neighbors of the node divided by



(a) Protocol overhead of Gozar vs. Nylon.



(b) Overhead traffic of Gozar vs. Nylon for varying percentages of private nodes.

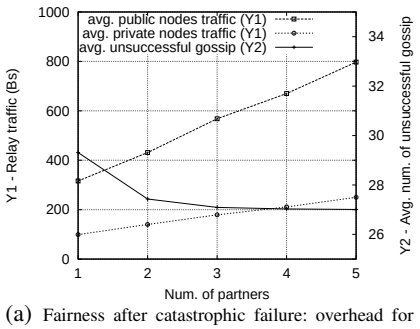
Fig. 2. Protocols overhead.

all possible links. Figure 1(d) shows the evolution of the clustering coefficient of the constructed overlay by each system. We can see that Gozar and Nylon almost have the same clustering coefficient as Cyclon, while the value for ARRG is higher.

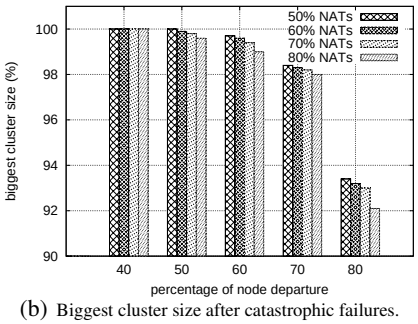
6.3 Protocol overhead

In this section, we compare the protocol overhead of Gozar and Nylon in different settings, where the protocol overhead traffic is the extra messages required to route messages through NATs. Protocol overhead traffic in Gozar consists of relay traffic and partner management, while in Nylon it consists of routing traffic. Figure 2(a) shows the protocol overhead when 80% of nodes are behind NAT. The Y1-axis shows the total overhead, and the Y2-axis shows the average overhead of each public and private node. In this experiment, each private node in Gozar has three public nodes as partners, but only one partner is used to relay a message to a private node. Nylon, however, routes messages through more than two intermediate nodes on average (see [19] for comparable results). Figure 2(a) shows that after 250 cycles the relay traffic and partner management overhead in Gozar is $20000KB$, while the routing traffic overhead in Nylon is roughly $37000KB$.

Now, we compare the protocol overhead for Gozar and Nylon for different percentages of private nodes. To show the overhead in adding more partners, we consider two settings for Gozar: private nodes have one partner, and private nodes have three partners. In figure 2(b), we can see that when 80% of nodes are behind NAT, the protocol overhead for all nodes in Nylon is around $150KBs$ after 250 cycles. The corresponding overhead in Gozar, when the private nodes have three and one partners, are around $70KBs$ and $40KBs$, respectively. The main contributory difference between the protocol overhead in the two different partner settings is that *shuffle request* and *shuffle response* messages become larger for more partners, as all partners addresses are included in private nodes' descriptors. The increase in traffic is a function of the percentage of private nodes (as only their descriptors include partner addresses), but is independent of the size of the partial view.



(a) Fairness after catastrophic failure: overhead for public and private nodes for varying numbers of parents.



(b) Biggest cluster size after catastrophic failures.

Fig. 3. Behaviour of the system after catastrophic failure.

6.4 Fairness and connectivity after catastrophic failure

We evaluate the behaviour of Gozar if high numbers of nodes leave the system or crash. Our experiment models a catastrophic failure scenario: 20 cycles after 1000 nodes have joined, 50% of nodes fail following a Poisson distribution with inter-arrival time of 10 milliseconds.

Our first failure experiment shows the level of fairness between public and private nodes after the catastrophic failure. In figure 3(a), the Y1-axis shows the average traffic on each public node and private node for different number of partners, and the Y2-axis shows the average number of unsuccessful view exchanges for each node. Here, 80% of nodes are private nodes and we capture the results 80 cycles after 50% of the nodes fail. As we can see in figure 3(a), the higher the number of partners the private nodes have, the more overhead traffic generated, again, due to the increasing the size of messages exchanged among nodes. The Y2-axis shows that when the private nodes have only one partner, the average number of unsuccessful view exchanges is higher than when the private nodes have more than one partner. If a private node has more than one partner, then in case of failure of any of them, there are still other partners that can be used to communicate with the private node. An interesting observation here is that we cannot see a big decrease in the number of unsuccessful view exchanges when the private nodes has more than two partners. This observation, however, is dependent on our catastrophic failure model, and high churn rates might benefit more from more than two partners.

Finally, we measure the size of biggest cluster after a catastrophic failure. Here, we assume that each private node has three partners. Figure 3(b) shows the size of biggest cluster for varying percentages of private nodes, when varying numbers of nodes fail. We can see that Gozar is resilient to node failure. For example, in the case of 80% private nodes, when 80% of the nodes fail, the biggest cluster still covers more than 92% of the nodes.

7 Conclusion

In this paper, we presented *Gozar*, a NAT-friendly gossip-based peer sampling service that also provides a distributed NAT traversal service to clients of the PSS. Public nodes are leveraged to provide both the relaying and hole punching services. Relaying is only used for gossiping to private nodes, and is preferred to hole punching or routing through existing open connections (as done in Nylon), as relaying has lower connection latency, enabling a faster gossiping cycle, and the messages relayed are small, thus, adding only low overhead to public nodes. Relaying and hole punching services provided by public nodes are enabled by every private node partnering with a small number of (redundant) public nodes and keeping a connection open to them. We extended node descriptors for private nodes to include the addresses of their partners, so when a node wishes to send a message to a private node (through relaying) or establish a direct connection with the private node through hole punching, it sends a relay or connection message to one (or more) of the private node's partners.

We showed in simulation that Gozar preserves the randomness properties of a gossip-based peer sampling service. We also show that the protocol overhead in our system is less than that of Nylon in different network settings and different percentages of private nodes. We also showed that the extra overhead incurred by public nodes is acceptable. Finally, we show that if 80% of the nodes are private, and when 50% of the nodes suddenly fail, more than 92% of nodes stay connected.

In future work, we will integrate our existing P2P applications with Gozar, such as our work on video streaming [22, 23], and evaluate their behaviour on the open Internet.

References

1. Cosmin Arad, Jim Dowling, and Seif Haridi. Developing, simulating, and deploying peer-to-peer systems using the kompics component model. In *COMSWARE '09: Proceedings of the Fourth International ICST Conference on COMmunication System softWAre and middlewaRE*, pages 1–9, New York, NY, USA, 2009. ACM.
2. Gautier Berthou and Jim Dowling. P2p vod using the self-organizing gradient overlay network. In *SOAR '10: Proceeding of the second international workshop on Self-organizing architectures*, pages 29–34, New York, NY, USA, 2010. ACM.
3. Dario Bonfiglio, Marco Mellia, Michela Meo, Dario Rossi, and Paolo Tofanelli. Revealing skype traffic: when randomness plays with you. *SIGCOMM Comput. Commun. Rev.*, 37(4):37–48, 2007.
4. Niels Drost, Elth Ogston, Rob V. van Nieuwpoort, and Henri E. Bal. Arrg: real-world gossiping. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 147–158, New York, NY, USA, 2007. ACM.
5. L. DAcunto, M. Meulpolder, R. Rahman, J.A. Pouwelse, and H.J. Sips. Modeling and analyzing the effects of firewalls and nats in p2p swarming systems. In *Proceedings IPDPS 2010 (HotP2P 2010)*. IEEE, April 2010.
6. L. DAcunto, J.A. Pouwelse, and H.J. Sips. A measurement of nat and firewall characteristics in peer-to-peer systems. In Lex Wolters Theo Gevers, Herbert Bos, editor, *Proc. 15-th ASCI Conference*, pages 1–5, P.O. Box 5031, 2600 GA Delft, The Netherlands, June 2009. Advanced School for Computing and Imaging (ASCI).
7. Patrick Th. Eugster, Rachid Guerraoui, S. B. Handurukande, Petr Kouznetsov, and Anne-Marie Kermarrec. Lightweight probabilistic broadcast. In *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, pages 443–452, Washington, DC, USA, 2001. IEEE Computer Society.
8. Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-peer communication across network address translators. *CoRR*, abs/cs/0603074, 2006.

9. Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulie. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52:2003, 2003.
10. Saikat Guha and Paul Francis. Characterization and measurement of tcp traversal through nats and firewalls. In *IMC '05: Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.
11. Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *SIGCOMM Internet Measurement Workshop*, 2002.
12. Ray Hunt Huynh Cong Phuoc and Andrew McKenzie. Nat traversal techniques in peer-to-peer networks, 2008.
13. R. Mahy J. Rosenberg and C. Huitema. Turn - traversal using relay nat. In *[Online]. Available: http://tools.ietf.org/id/draft-rosenberg-midcom-turn-08.txt, Sep. 2005*.
14. M. Jelasity, A. Montresor, and O. and Babaoglu. T-Man: Gossip-based fast overlay topology construction. *Computer Networks*, 53(13):2321–2339, 2009.
15. Márk Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 79–98, New York, NY, USA, 2004. Springer-Verlag New York, Inc.
16. Márk Jelasity and Alberto Montresor. Epidemic-style proactive aggregation in large overlay networks. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 102–109, Washington, DC, USA, 2004. IEEE Computer Society.
17. Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, 2005.
18. Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3):8, 2007.
19. Anne-Marie Kermarrec, Alessio Pace, Vivien Quema, and Valerio Schiavoni. Nat-resilient gossip peer sampling. In *ICDCS '09: Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, pages 360–367, Washington, DC, USA, 2009. IEEE Computer Society.
20. J. Leitão, R. van Renesse, and L. Rodrigues. Balancing gossip exchanges in networks with firewalls. In *Proceedings of the 9th International Workshop on Peer-to-Peer Systems (IPTPS '10)*, page (to appear), San Jose, CA, U.S.A., 2010.
21. Y. Liu and J. and Pan. The impact of NAT on BitTorrent-like P2P systems. In *IEEE Ninth International Conference on Peer-to-Peer Computing, 2009. P2P'09*, pages 242–251, 2009.
22. Amir H. Payberah, Jim Dowling, Fatemeh Rahimian, and Seif Haridi. gradienTv: Market-based P2P Live Media Streaming on the Gradient Overlay. In *Lecture Notes in Computer Science (DAIS 2010)*, pages 212–225. Springer Berlin / Heidelberg, Jan 2010.
23. Amir H. Payberah, Jim Dowling, Fatemeh Rahimian, and Seif Haridi. Sepidar: Incentivized market-based p2p live-streaming on the gradient overlay network. *International Symposium on Multimedia*, 0:1–8, 2010.
24. R. Price and P. Tino. Adapting to NAT timeout values in P2P overlay networks. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–6. IEEE, 2010.
25. J. Rosenberg. Interactive connectivity establishment (ice): A methodology for network address translator (nat) traversal for offer/answer protocols. In *[Online]. Available: http://tools.ietf.org/html/draft-ietf-mmusic-ice-13, Jan. 2007*.
26. J. Rosenberg, R. Mahy, P. Mathews, and D. Wing. Rfc 5389: Session traversal utilities for nat (stun), 2008.
27. Roberto Roverso, Sameh El-Ansary, and Seif Haridi. Natcracker: Nat combinations matter. In *ICCCN '09: Proceedings of the 2009 Proceedings of 18th International Conference on Computer Communications and Networks*, pages 1–7, Washington, DC, USA, 2009. IEEE Computer Society.
28. Jan Sacha, Jim Dowling, Raymond Cunningham, and René Meier. Discovery of stable peers in a self-organising peer-to-peer gradient topology. In Frank Eliassen and Alberto Montresor, editors, *6th IFIP WG 6.1 International Conference Distributed Applications and Interoperable Systems (DAIS)*, volume 4025, pages 70–83, Bologna, June 2006.
29. Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13:2005, 2005.