Shuffling with a Croupier: Nat-Aware Peer-Sampling

Jim Dowling[†‡], Amir H. Payberah[†‡]
[†]Swedish Institute of Computer Science (SICS)
[‡]KTH - Royal Institute of Technology
Email: {jdowling, amir}@sics.se

***Abstract*—Despite much recent research on peer-to-peer (P2P) protocols for the Internet, there have been relatively few practical protocols designed to explicitly account for Network Address Translation gateways (NATs). Those P2P protocols that do handle NATs circumvent them using relaying and hole-punching techniques to route packets to nodes residing behind NATs.**

In this paper, we present Croupier, a peer sampling service (PSS) that provides uniform random samples of nodes in the presence of NATs in the network. It is the first NAT-aware PSS that works without the use of relaying or hole-punching. By removing the need for relaying and hole-punching, we decrease the complexity and overhead of our protocol as well as increase its robustness to churn and failure. We evaluated Croupier in simulation, and, in comparison with existing NAT-aware PSS', our results show similar randomness properties, but improved robustness in the presence of both high percentages of nodes behind NATs and massive node failures. Croupier also has substantially lower protocol overhead.

***Keywords*-P2P Networks; Gossip Peer Sampling; NAT;**

I. Introduction

Peer sampling services (PSS) are widely used in large scale distributed applications, such as information dissemination [1], aggregation [2], and overlay topology management [3], [4]. A 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 [5]. PSS' can be implemented using gossip protocols [6], [7] or random walks [8], although random walks are only suitable for static networks with low levels of churn [9].

In networks where all nodes can directly communicate with each other, a gossip-based PSS' can ensure that node descriptors are distributed uniformly at random over all partial views [7]. However, in the Internet, where a high percentage of nodes are behind NATs and firewalls, traditional gossip-based PSS' become biased [9]. Nodes cannot establish direct connections to nodes behind NATs or firewalls (*private nodes*), and as a result private nodes become under-represented in partial views. Conversely, nodes that do support direct connectivity, *public nodes*, become over-represented in partial views. The main challenges for a NAT-aware PSS are to generate uniformly random node samples for high percentages of private nodes, to maintain connectivity during high node failure rates, all while minimizing the protocol overhead.

Relaying is a technique used by existing NAT-aware PSS' for communicating with private nodes. In relaying, instead of sending a message directly to a private node, the message is sent via a relay node and the relay node forwards the message either directly to the private node using an existing open connection or via a chain of relay nodes. Relaying has been used to solve the problem of balancing gossip in networks with NATs [9], [10], [11], as it ensures that private nodes receive a balanced number of gossip messages. Relaying, however, introduces complexity into PSS protocols: relaying nodes have to maintain routing tables for private nodes, and private nodes have to maintain open mappings in their NAT to relay nodes. Also, where the system is distributed, nodes have to discover the relay node(s) responsible for the private node they wish to communicate with. Existing techniques for discovering responsible relay nodes include caching the addresses of relay nodes in node descriptors [10], maintaining routing tables to nodes that have recently been communicated with [9], [11] and using a distributed-hash table [12].

Existing gossip-based NAT-aware PSS' [9], [10] are similar to classic PSS [6], [7] in that they maintain a single partial view containing descriptors for a small subset of nodes in the system, and periodically pick a random node to exchange its partial view with. Partial views are randomized in a process called view exchange, where a node selects a neighbour and shuffles its partial view with the neighbour's. If the selected node is a private node, first, the relay node for that private node is discovered, then a view exchange request is sent to the relay node.

In this paper, we present *Croupier*, that introduces a novel mechanism for exchanging partial views to build a PSS without the use of relaying. Our main intuition is to use two partial views, one for public nodes and one for private nodes, and to have public nodes act as Croupiers, exchanging public and private views on behalf of private nodes. View exchanges are initiated by all nodes, but only sent to public nodes (the Croupiers) who shuffle the views. In order to generate a random sample from the two partial views, our protocol requires that we estimate the ratio of public to private nodes in the system. Public nodes collectively estimate the ratio of public to private nodes by sampling the recent rate of view exchange requests from public and private nodes, respectively. As all nodes send a single view exchange request per round to a random public node and the round time is equal at all nodes (subject to clock skew), we estimate the ratio of public to private nodes using a distributed averaging algorithm based

on sampled request rates. Croupier is a different approach than our previous work on a NAT-aware PSS built on relaying, Gozar [10], and a general NAT-traversal middleware based on Distributed Hash Table, Usurp [12]. Croupier's contribution is that produces a more robust, lower overhead NAT-aware PSS with similar randomness properties to existing systems.

We evaluated Croupier in simulation, and in comparison with the best existing NAT-aware PSS', Gozar [10] and Nylon [9], our results show similar to slightly improved randomness properties, and improved robustness in the presence of high percentages of private nodes and high levels of node churn. Croupier also has 50% less overhead than that of Gozar and 80% less compared to Nylon.

II. Background and related work

The ratio of public to private nodes in existing P2P systems varies considerably depending on the geographical distribution of the participating nodes. There are two main trends affecting the public/private ratio: a decreasing number of open IPv4 addresses are being made available to end-users due to the limited size of the IPv4 address space, and, secondly, an increasing number of nodes behind NATs have enabled the UPnP Internet Gateway Device (IGD), allowing them to effectively act as public nodes. D'Acunto et al. [13] showed for a BitTorrent-like system in 2009 that the percentage of nodes with open IP addresses varies from 9% in the USA to 23% in Italy. They did not, however, consider whether nodes support the UPnP IGD. In contrast, the live streaming system NewCoolStreaming [14] included nodes that support UPnP IGD and showed that 20.8% of nodes act as public nodes, with most nodes located in the USA in 2009.

Peer sampling has been widely studied in the area of overlay networks [7]. In gossip-based PSS', protocol execution at each node is divided into periodic rounds. Implementations can vary based on a number of different policies in node selection (*random* selects a random neighbour, *tail* selects the oldest node descriptor), view exchange (push or push-pull) and view merging (*healer* select most recents node descriptor, *swapper* swaps a subset of its local view with its neighbour minimizing loss of information in the system)) [7]. 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 *in-degree distribution*, *average shortest path* and *clustering coefficient*, are close to a random network [6], [7]. The impact of NATs on traditional gossip-based PSS' has been evaluated in both [9] and [15]. 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.

The first PSS' to address the problem of NATs was ARRG [15]. 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. More recently, Kermarrec et al. introduced in Nylon [9] a NAT-aware PSS that uses all existing nodes in the system (both private and public nodes) as rendezvous servers (RVPs). A RVP provides connectivity to private nodes by facilitating hole-punching the private node's NAT. In Nylon, two nodes become the RVP of each other whenever they exchange their views. If a node selects a private node for gossip exchange, it hole-punches a direct connection to the private node using a chain of RVPs until the chain reaches the private node. The chains of RVPs in Nylon are unbounded in length, making Nylon fragile in networks with churn, as well as increasing overhead at intermediary nodes [10]. Their chain of RVPs also performs poorly over high latency links, which are frequently found on the Internet [16].

In our previous work on Gozar [10], we replaced RVP chains with one-hop relaying to all private nodes. Private nodes discover and maintain a redundant set of public nodes that act as relay nodes on their behalf. Nodes shuffled with private nodes by relaying messages via at least one of the private node's relay nodes, where the addresses of the relay nodes are cached in node descriptors. Through redundant relay nodes and quickly expiring node descriptors, connectivity to private nodes is maintained and latency is kept low, even under churn.

In other work on NAT-aware gossiping, Renesse et al. [11] present an approach to fairly distribute relay traffic over public nodes. In their system, each node balances the number of gossip requests it accepts to the number of gossip exchanges it has sent itself. Nodes that have already accepted enough gossip requests, forward them in a manner similar to Nylon, using chains of nodes as relay servers.

Our public/private ratio estimation algorithm is related to existing gossip-based estimation algorithms that estimate the number of nodes in a system [2], [17] and estimate the distribution of attribute values across all nodes [18]. These algorithms require multiple aggregation instances in parallel to improve their estimation accuracy and assume full connectivity between nodes, that is, no NATs. In contrast to these aggregation algorithms, our aggregation algorithm is NAT-friendly and does not need to be run as an independent protocol - estimation in Croupier is done by piggy-backing on view exchange messages.

III. System model

We model a distributed system as a network of autonomous nodes that exchange messages. The goal of Croupier is to provide a PSS, locally at all nodes, where the PSS periodically provides samples of nodes drawn uniformly at random from the set of all nodes in the system. There is no central point of control in the system and all nodes execute the PSS algorithm. Each node knows its own NAT type, which is either public or private, where a public node can be communicated with using an IP address that is globally reachable from any other node, while a private node resides behind at least one NAT or

firewall, and is not reachable from outside its private network unless its is the private node that initiates contact. Each node separately maintains references to both a small, bounded number of randomly selected public nodes in a *public view* and a small, bounded number of randomly selected private nodes in a *private view*. Collectively, a node refers to the nodes in its public and private views as its neighbors.

IV. Problem description

We partition the partial view into two separate bounded-size views: a public view and a private view. This prevents over-representation of public nodes in partial views, but it introduces the problem of how generate a uniform random sample from the two views - we cannot just pick a random neighbor from one of either the public or private views, as we need to know the correct proportion of public to private nodes in the system when generating a sample. That is, we need a distributed algorithm that estimates the ratio of public to private nodes in the system. This ratio may vary both between different systems and over the lifetime of a system, but when a good estimation is available locally at every node, we can use it to sample the correct proportion of nodes from either the public or private view.

V. A distributed NAT type identification protocol

Croupier requires that a node knows its correct NAT type as either public or private. A node's NAT type could be determined by a centralized service, such as a Session Traversal Utilities for NAT (STUN) server [19], but instead we introduce a distributed, minimal NAT type identification protocol that identifies a node as being either public or private. Our protocol can, in principle, be run at any time during system operation, but is typically run once at bootstrap time, as the vast majority of nodes stay either public or private for the duration of their session. When a node's NAT type doesn't change, the protocol does not need to be run for every session, as the NAT type can be cached across sessions.

The protocol is defined in algorithm 1, and is run over UDP. Several instances of the protocol can be run in parallel against different public nodes to improve its robustness and reduce its expected completion time (the protocol finishes when the first public node responds). It identifies a node as a public node if (i) it has a globally reachable IP address and is not behind a NAT or firewall or (ii) if the node's NAT supports the UPnP Internet Gateway Device Protocol (that is, the node can explicitly map a local port to a port on the public interface of its UPnP-enabled NAT, where the NAT has a public IP address). If neither of these two conditions are matched, the node is a private node.

To realise these properties, the protocol executes two tests: firstly, a *MatchingIpTest* compares the node-under-test's local IP address with the IP address seen by a public node, and, secondly, a *ForwardTest* checks to make sure the node-under-test can receive a packet from a public node to which it has not sent a packet in the last 5 minutes, where 5 minutes is assumed higher than the NAT UDP mapping timeout. The tests are executed in parallel over a number of public nodes returned by a *bootstrap server*. The protocol requires only three network messages per run: a *MatchingIpTest* is sent from the node-under-test to a public node returned by the bootstrap server, this node then inserts the public IP address from which it received the event into a *ForwardTest* event that is sent to a different public node (not one of the public nodes returned originally by the bootstrap server - as the node-under-test may be running the protocol in parallel against them). The node that receives the *ForwardTest* event then sends that event back to the node-under-test's public IP address.

The *ForwardTest* event cannot be sent to any of the public nodes returned by the bootstrap server as the node-under-test's NAT may have an entry in its NATs mapping table to that node's IP address, and the *ForwardTest* event would erroneously pass through the NAT. If the client receives the *ForwardTest* event and its local IP address matches the IP address seen in *MatchingIpTest*, then the node's NAT type is public. If the IP addresses do not match, then the node is set to private. This case can happen if the node is behind a NAT that has an Endpoint-Independent filtering policy [20]. If the node's NAT has a more restrictive packet filtering policy or the node is behind a firewall, it will not receive the *ForwardTestResponse* event, and its *Timeout* event handler will return that the node is private. The length of the timeout needs to be long enough to prevent false positives, but it can be adjusted upwards if a late *ForwardTest* event is received after the timeout has expired.

Algorithm 1 Minimal distributed NAT type identification.

```
1:  procedure NatTypeIdentificationClient ⟨this⟩
2:      // Executed at client on joining system
3:      publicNodes ← doBootstrap()
4:      if supportsUpnpIGD() == true then
5:          nodeType ← public
6:      else
7:          for all node_i in publicNodes do
8:              Send MatchingIpTest(publicNodes) to node_i
9:          end for
10:         After timeToWait Send Timeout(publicNodes) to this
11:     end if
12: end procedure

13: // Timeout event triggered if no ForwardResp event is received in time
14: on receive ⟨TIMEOUT | publicNodes⟩ from this do
15:     nodeType ← private
16: end event

17: // Event handler at client node
18: on receive ⟨FORWARDRESP | clientIp⟩ from secondPublicNode do
19:     Send CancelTimeout to this
20:     if this.localIp == clientIp then
21:         nodeType ← public
22:     else
23:         nodeType ← private
24:     end if
25: end event

26: // Event handler at first public node
27: on receive ⟨MATCHINGIPTEST | publicNodes⟩ from client do
28:     secondPublicNode ←last good public node seen not in publicNodes
29:     Send ForwardTest(clientIp) to secondPublicNode
30: end event

31: // Event handler at second public node
32: on receive ⟨FORWARDTEST | clientIp⟩ from firstPublicNode do
33:     Send ForwardResp(clientIp) to clientIp
34: end event
```

VI. The Croupier protocol

Our peer sampling algorithm, Croupier, is based on periodic gossip rounds, executed at roughly the same rate by all nodes (subject to clock skew), where neighbouring nodes exchange local state. Croupier's pseudo-code is given in algorithm 2. Our shuffling algorithm is based on the tail, push-pull and swapper policies for node selection, view exchange and view selection from [7]. The tail policy involves selecting the oldest node descriptor for shuffling, while the swapper policy involves replacing the node descriptors sent to the other node with the received node descriptors.

Each node p maintains a public view, $view_u(p)$, and a private view, $view_v(p)$, both bounded in size, consisting of a set of node descriptors of public and private nodes, respectively. A node descriptor contains the node's address, its NAT type, and a timestamp storing the number of rounds since the descriptor was created. A node p periodically executes the procedure $Round$ to exchange and update both p's views and its ratio estimations in $E_p(\omega)$, see equations 8 and 9. $Round$ firstly updates the age of both the descriptors in p's views and its ratio estimations $E_p(\omega)$. Then, the oldest descriptor q (tail) is selected and removed from the public view, $view_u(p)$, and a *shuffle request* is sent to q. The public node q receives the *shuffle request* containing the following state: a random, bounded subset of the sender p's public view, a random, bounded subset of p's private view, and a random, bounded subset of p's estimations from $E_p(\omega)$.

The public node q's handler for the *shuffle request* takes the following actions. First, depending on whether the sender of the request p is public or private, the public or private shuffle counter C_u or C_v is incremented. Then, it updates its private and public views as well as its estimations using the parameters in the *shuffle request*. The private and public views are updated in $updateView$ procedure, by first checking if the node already exists in its view, and if so, updating it if the received node descriptor is newer. Secondly, if there is free space the received node descriptor is added to its view. Finally, if a view exchange has recently been completed with the node who sent the *shuffle request*, then any node descriptors we sent to that node and are currently in our view and replaced with the node descriptors received.

A *shuffle response* is subsequently sent back to p. Similar to the request, the response includes a bounded, random subset from its public and private views and its ratio estimations. When p receives the *shuffle response*, similar to the *shuffle request* event handler, it updates its private and public views and its estimations using state in the event.

Sampling and ratio estimation

The procedure $generateRandomSample$ in algorithm 3 is called to generate a uniform random sample of nodes from either a public or private node. This procedure needs a good estimation of the ratio of public to private nodes. In the following, we assume both a static ratio of public to private nodes and a fixed number of nodes, although, as shown in our evaluation, our estimation algorithm gives good estimations

Algorithm 2 Croupier shuffling algorithm.

```
1:  // run by each node p in each gossiping round
2:  procedure Round ⟨⟩
3:      update ages of descriptors in view_u and view_v
4:      update ages of estimations in M_p   ▷ estimations received from public nodes
5:      remove estimations older than γ from M_p
6:      if natType is public then
7:          E_p ← CalcHitsRatio()
8:      end if
9:      C_u = C_u ∪ c_u                          ▷ keep a local history of public hits
10:     C_v = C_v ∪ c_v                          ▷ keep a local history of private hits
11:     c_u = 0, c_v = 0                ▷ initialize new estimations for current round
12:     q ← select oldest node from view_u          ▷ oldest node in the public view
13:     remove q from view_u
14:     pPub ← random subset from view_u
15:     pPri ← random subset from view_v
16:     pSubM ← random subset from M_p
17:     if natType is public then
18:         pPub.add(this)
19:     else
20:         pPri.add(this)
21:     end if
22:     Send ShuffleReq(pPub, pPri, pSubM, E_p) to q
23: end procedure

24: // shuffle requests are handled by public nodes q
25: on receive ⟨ShuffleReq | pPub, pPri, pSubM, E_p⟩ from p do
26:     if p.natType is public then
27:         increment c_u
28:     else
29:         increment c_v
30:     end if
31:     qPub ← random subset from view_u
32:     qPri ← random subset from view_v
33:     qSubM ← random subset from M_q
34:     updateView(view_u, qPub, pPub)
35:     updateView(view_v, qPri, pPri)
36:     M_q = M_q ∪ pSubM ∪ E_p              ▷ retain most recent by timestamp.
37:     Send ShuffleRes(qPub, qPri, qSubM, E_q) to p
38: end event

39: // shuffle responses are handled by both public and private nodes
40: on receive ⟨ShuffleRes | qPub, qPri, qSubM, E_q⟩ from q do
41:     updateView(view_u, pPub, qPub)
42:     updateView(view_v, pPri, qPri)
43:     M_p = M_p ∪ qSubM ∪ E_q
44: end event

45: // used to update either the public or the private view
46: procedure updateView ⟨view, sentView, receivedView⟩
47:     for all node_i in receivedView do
48:         if view contains node_i then
49:             view.updateAge(node_i)
50:         else if view has free space then
51:             view.add(node_i)
52:         else
53:             node_j ← sentView.poll()
54:             view.remove(node_j)
55:             view.add(node_i)
56:         end if
57:     end for
58: end procedure

59: // calculates the hits ratio
60: procedure CalcHitsRatio ⟨⟩
61:     pubCnt = 0
62:     priCnt = 0
63:     remove hits older than α from C_u and C_v
64:     for all u in C_u do
65:         pubCnt = pubCnt + u
66:     end for
67:     for all v in C_v do
68:         priCnt = priCnt + v
69:     end for
70:     result = pubCnt / (pubCnt + priCnt)          ▷ calculates the local estimation
71:     return result
72: end procedure
```

for dynamic ratios. Public nodes $\mathcal{U}$ and private nodes $\mathcal{V}$ make up the set of all nodes $\mathcal{N}$ in the system: $\mathcal{N} = \mathcal{U} \cup \mathcal{V}$. The ratio ω of public to private nodes in the system is defined as:

$$\omega = \frac{|\mathcal{U}|}{|\mathcal{U}| + |\mathcal{V}|}. \tag{1}$$

We estimate ω using a decentralized algorithm that is based on three basic assumptions: firstly, there should be no bias between the average gossip round-time of public nodes and private nodes, secondly, the target of shuffle requests should be chosen uniformly at random among public nodes, and thirdly, there should be no bias in message loss between public and private nodes. Our first and third assumptions imply that the rate of shuffle requests coming from public nodes compared to private nodes is roughly the same as ω. Our second assumption is grounded on the equivalence of our node selection algorithm to Cyclon's [6], which has previously shown that nodes are selected almost uniformly at random. Our estimation of ω uses the relative number of shuffle requests received by Croupiers (public nodes) from other public nodes or private nodes, within a small time window α into the past (called the *local history*). If we assume α is equal to the system lifetime, we can define the number of shuffle requests that all Croupiers in the system receive from public nodes as $\mathbb{C}_u$, and the number of shuffle requests all Croupiers receive from private nodes as $\mathbb{C}_v$. For each Croupier i, its local public and private shuffle request counts are defined as C_{ui} and C_{vi}, respectively. That is the system-wide shuffle request counts are defined as the sum of local shuffle counts:

$$\mathbb{C}_u = \sum_{i \in \mathcal{U}} C_{ui} \text{ and } \mathbb{C}_v = \sum_{i \in \mathcal{U}} C_{vi} \tag{2}$$

Our estimation of the ratio of public to private nodes, $E(\omega)$, can now be calculated as the ratio of the number of shuffle requests from public nodes to the number of shuffle requests from all nodes:

$$E(\omega) = \frac{\mathbb{C}_u}{\mathbb{C}_u + \mathbb{C}_v} \tag{3}$$

Assuming our first and third assumptions hold, over all public nodes in the system, ω is roughly equal to $E(\omega)$:

$$\omega \approx E(\omega) \tag{4}$$

As $E(\omega)$ is not available at any individual node, each public node i maintains its local part of the estimation E_i by updating its local counts C_{ui} and C_{vi} within the last time window α:

$$C_{ui} = \sum_{t=0}^{\alpha} c_{ui}(t) \text{ and } C_{vi} = \sum_{t=0}^{\alpha} c_{vi}(t) \tag{5}$$

where c_{ui} and c_{vi} are the number received requests from public and private nodes in each shuffle round, respectively. A node i, then, calculates the local estimation E_i as:

$$E_i = \frac{C_{ui}}{C_{ui} + C_{vi}} \tag{6}$$

Algorithm 3 Sampling and ratio estimation.

```
1:  // generates a random estimation of nodes using the ratio estimation
2:  procedure generateRandomSample ⟨⟩
3:      viewChoice ← random real number between 0 and 1.0
4:      if viewChoice < estimatePublicPrivateRatio() then
5:          return random entry from view_u
6:      else
7:          return random entry from view_v
8:      end if
9:  end procedure

10: // returns the estimation of the ratio of public/private nodes
11: procedure estimatePublicPrivateRatio ⟨⟩
12:     cnt = 0
13:     for all m in M_p do
14:         cnt = cnt + m
15:     end for
16:     if natType is public then
17:         result = (cnt + E_p) / (M_p.size + 1)
18:     else
19:         result = cnt / M_p.size
20:     end if
21:     return result
22: end procedure
```

As α approaches the system lifetime, the average of the local estimations is approximately equivalent to our global estimation:

$$E(\omega) \approx \frac{\sum_{i \in \mathcal{U}} E_i}{|\mathcal{U}|} \tag{7}$$

Each public node i stores its own local estimation E_i, and it also stores a set of local estimations M_i shared by other public nodes. All local estimations by public nodes should be independent of each other as shuffle requests should be uniformly distributed among public nodes. Public nodes can disseminate to their neighbours (public and private neighbours) both their own local estimation E_i as well as a subset of the estimations M_i they received from other public nodes. All estimations can be shared in a simple dissemination protocol to both private and public nodes, but, for efficiency, we piggy-back these estimations on *shuffle request* and *shuffle response* messages.

Estimates contain timestamps that are incremented at every gossip round by. When two estimations for the same node are available, the older estimation is replaced by the newer estimation. Old estimations with a timestamp higher than a configurable parameter γ (*neighbour history*) are removed every gossip round from M_i. Both private and public nodes store a number of estimations that are bounded by the size of α and γ. For every shuffle request and shuffle response, we bound the number of estimations that are shared to a subset of M_i to prevent the size of messages growing for increasing system size. In our experiments, we set this value to 10, and with 5 bytes used per estimation, that resulted in an overhead of 50 bytes per shuffle message. Given local and neighbour estimations, a public node i can estimate ω as the average of both its local estimation E_i and its cached estimations from other public nodes M_i:

$$E_i(\omega) = \frac{\sum_{n \in M_i} E_n + E_i}{|M_i| + 1} \tag{8}$$

In contrast, a private node i has no local estimation E_i (as it does not receive shuffle requests), so it estimates ω as the average of its cached estimations from public nodes M_i:

$$E_i(\omega) = \frac{\sum_{n \in M_i} E_n}{|M_i|} \quad (9)$$

Both equations 8 and 9 are defined in the method $estimatePublicPrivateRatio$ of algorithm 3. In the next section, we will show how the quality of the estimations depends on how stable the public/private ratio is and how well tuned α and γ are to the rate of change of the ratio.

VII. Evaluation

We now evaluate the performance of our public-private estimation algorithm in simulation and compare the performance of the Croupier PSS with Nylon [9] and Gozar [10], the two best performing NAT-friendly gossip-based PSS we found in the literature. We use also 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 [6].

A. Experimental setup

We implemented Croupier, Cyclon, Nylon and Gozar on the Kompics platform [21]. 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 and Nylon are based on the system descriptions in [6] and [9], respectively. For a cleaner comparison with Nylon and Gozar, all protocols use the same tail and swapper policies for node selection and view merging, respectively.

In our experimental setup, for all four systems, the size of a node's partial view is 10 entries, and the size of subset of the partial view sent in each view exchange is 5. The gossiping round 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 [16]. Unless stated otherwise, we use a public-private ratio of 0.2, similar to that seen in existing P2P systems [14], [13]. All experiments results are averaged over 5 runs. The evaluation metrics for new nodes that join the system are not included until they have executed 2 rounds, giving them enough time to initialize their estimates.

B. Evaluation of the Estimation algorithm

We measure the accuracy of our ratio estimation protocol using two error metrics: the maximum approximation error and the average approximation error. Firstly, we define the upper bound on the approximation error of any nodes in the system using the Kolmogorov-Smirnov [22] (or maximum error) metric. For each node n, for all sample points in an experiment run, we measure the maximum distance between ω and $E(\omega_n)$ as:

$$Err_{max}(p) = \arg\max_n \|\omega - E(\omega_n)\| \quad (10)$$

For each node n in the system, we measure the *maximum error* as the maximum error over all n:

$$Err_{max} = \arg\max_n Err_{max}(n) \quad (11)$$

As the maximum error is sensitive to noise, we also measure the average error at each node. The average error is calculated at each node n using:

$$Err_{avg}(n) = \omega - E_n(\omega) \quad (12)$$

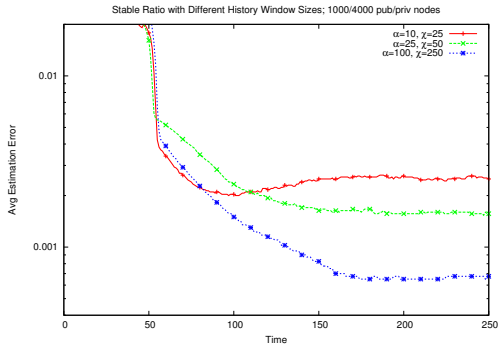
Our total average error is then calculated as the average of these local average errors:

$$Err_{avg} = \frac{\sum_{n \in \mathcal{N}} Err_{avg}(n)}{|\mathcal{N}|} \quad (13)$$

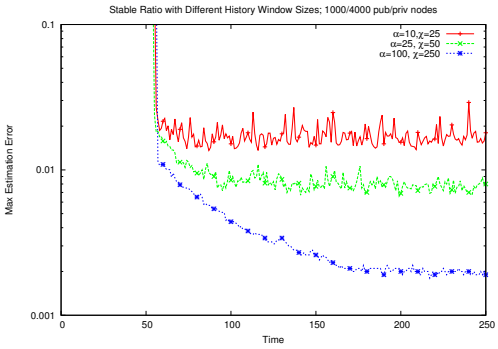
Setting history window sizes for stable and changing ratios

In this experiment, we evaluate the accuracy of our public/private ratio estimation using both a stable ratio and a dynamic ratio (where the ratio of public to private nodes changes over time). Both experiments have 1000 public nodes and 4000 private nodes join the system following a Poisson distribution with an inter-arrival time of 50 and 12.5 milliseconds, respectively. We measure the average error and maximum error while varying the size of the local history (α) and the neighbour history (γ). Our experiments use three pairs of history window sizes: a smaller window with α=10 and γ=25, a medium window with α=25 and γ=50, and a large window with α=100 and γ=250. For the stable ratio, in figures 1(a) and 1(b), we can see clearly that larger values of α and γ have a slower convergence rate, but more accurate estimations. All 5000 nodes have joined the system by time t=51, and it takes roughly 100 rounds longer for the largest history windows ($\alpha = 100$, $\gamma = 250$) to converge on good estimates compared to the smallest history windows ($\alpha = 10$, $\gamma = 25$). The largest history window run converges to an average error of 0.07% with a maximum error of 0.2%, while the smallest window converges to an average error of 0.25% with a maximum error of 1.8%.

In figures 2(a) and 2(b), we observe the convergence rate and estimation accuracy for a public/private ratio that grows slowly in size. We use the same scenario of joining 1000 public nodes and 4000 private nodes over the first 51 rounds, then waited 7 rounds, and then added a new public node every 42 ms. The actual ratio is 0.3 until time t=58, then the ratio rises at a constant rate to t=72 to reach 0.33, whereupon the ratio remains at 0.33 until the end of the experiment run. Again, we show the results for different local history (α) and a neighbour history (γ) sizes. We can see here that for a dynamic public-private ratio the largest history windows take a lot longer to converge on the new ratio, while the smallest history windows converge quicker (but eventually with less accurate estimations when the ratio stabilizes again). From t=58 to t=180, the smallest window has the lowest average error, while from t=180 to t=260 the medium-sized window has the lowest average error, then after t=260, the largest window converges

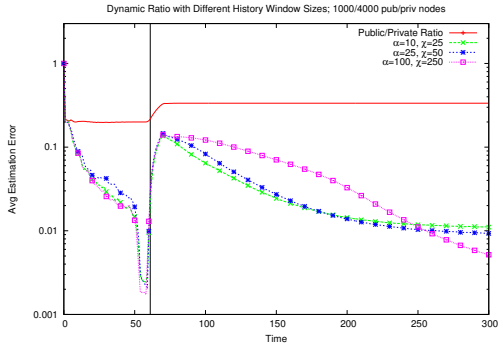


(a) Static public/private ratio: average error when varying the sizes of the local history (α) and neighbour history (γ).

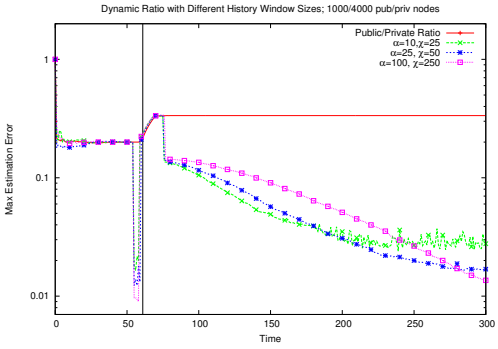


(b) Static public/private ratio: maximum error when varying the sizes of the local history (α) and neighbour history (γ).

Fig. 1. Convergence to a static ratio for different values of α and γ.



(a) Dynamic public/private ratio: average error when varying the sizes of the local history (α) and neighbour history (γ).



(b) Dynamic public/private ratio: maximum error when varying the sizes of the local history (α) and neighbour history (γ).

Fig. 2. Convergence to a dynamic ratio for different values of α and γ.

closer to the real ratio. For a ratio that changes frequently and by a large amount, we would need window sizes closer to our smaller window sizes, but for more stable ratios medium or large-sized windows would have lower average error and lower maximum errors. Unless stated otherwise, further experiments use the medium history window sizes, α=25 and γ=50, as, for a real system, it would provide a reasonable balance of good estimations and adaptability to a dynamic ratio.

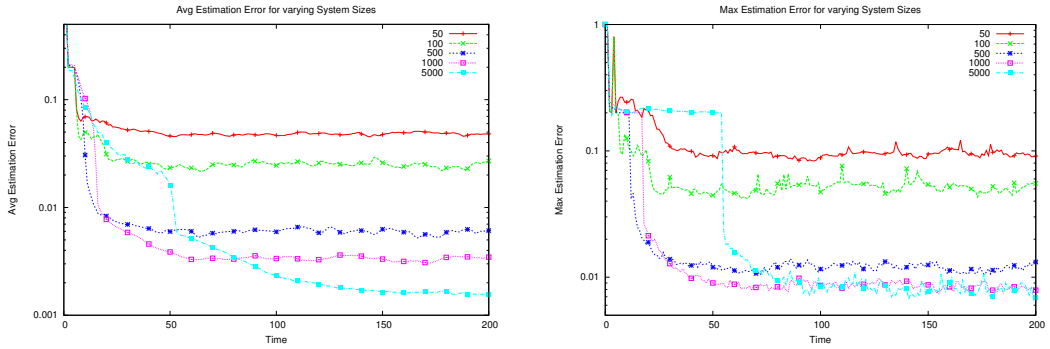
Impact of system size on estimation

In this experiment, we vary the number of nodes in the system to see its effect on the estimation accuracy. We measure systems with 50, 100, 500, 1000, and 5000 nodes. In these experiments, public and private nodes public nodes join the system following a Poisson distribution with an inter-arrival time of 50 and 12.5 milliseconds, respectively.

In figures 3(a) and 3(b), we can see that there is an increase in estimation accuracy with increasing system size. For systems with 5000 nodes, average estimation error is only 0.2%, while for systems with only 100 nodes it rises to 2.5%, rising again to 5% for systems with only 50 nodes. Similarly, the maximum estimation error rises from 0.7% for 5000 nodes to 5.5% for 100 nodes, and to 9% for 50 nodes. In general, we can say that estimation accuracy improves rapidly up to systems with several hundred nodes, and then only becomes gradually better thereafter. For example, the change in estimation accuracy from 1000 to 5000 nodes is negligible - an improvement in average estimation error of only 0.15% and no difference in maximum estimation error. As such, in all subsequent simulations, we set the size of the systems to 1000 nodes, where the nodes join the system following a Poisson distribution with an inter-arrival time of 10 milliseconds, and unless stated otherwise, 20% of nodes are public nodes.

Effect of different ratios on estimations

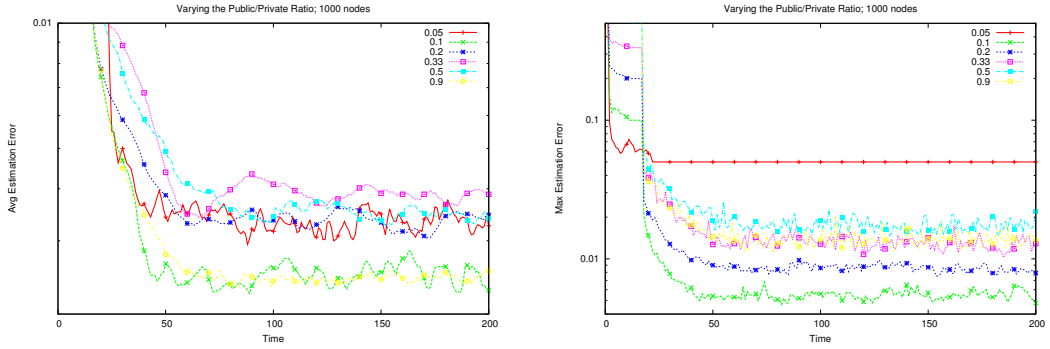
Different P2P systems will have different ratios of public to private nodes, so here we investigate the accuracy of estimations for different stable ratios of public to private nodes, with experiments of 1000 nodes. We measure the average and maximum estimation errors for ratios of 5%, 10%, 20%, 33%, 50%, 80%. We concentrate our measurements more on systems with smaller relative numbers of public nodes, as this is commonly the case in real-world systems. As we can see in figures 4(a) and 4(b), there is no significant difference in the average estimation error for all ratios. We do notice, however, for only 5% public nodes that the maximum error becomes significantly higher (5%) and constant. This is the result of an outlier private node that happens not to receive enough different estimates from public to improve its local estimation. So, for systems with fewer than 5% public nodes, we can



(a) Effect of different system sizes on average estimation errors, with $\alpha = 25, \gamma = 50$.

(b) Effect of different system sizes on maximum estimation errors, with $\alpha = 25, \gamma = 50$.

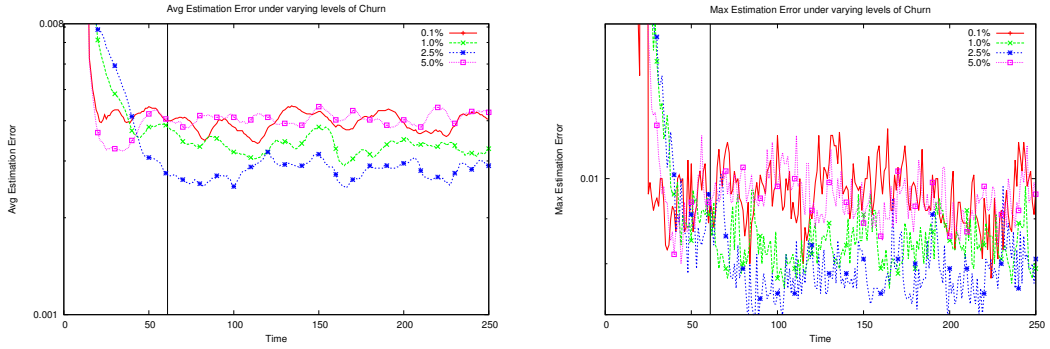
Fig. 3. Evaluating the effect of system size on the estimation algorithm for a stable ratio of 0.2.



(a) Different public/private ratios: average error when varying the ratio of public to private nodes.

(b) Different public/private ratios: maximum error when varying the ratio of public to private nodes.

Fig. 4. Estimation accuracy for different ratios of public to private nodes.



(a) Effect of churn on estimation. (α)=25, (γ)=50. Average estimation error. Churn started at time t=61.

(b) Effect of churn on estimation. (α)=25, (γ)=50. Maximum estimation error. Churn started at time t=61.

Fig. 5. Evaluating the effect of churn on the estimation algorithm for a stable ratio.

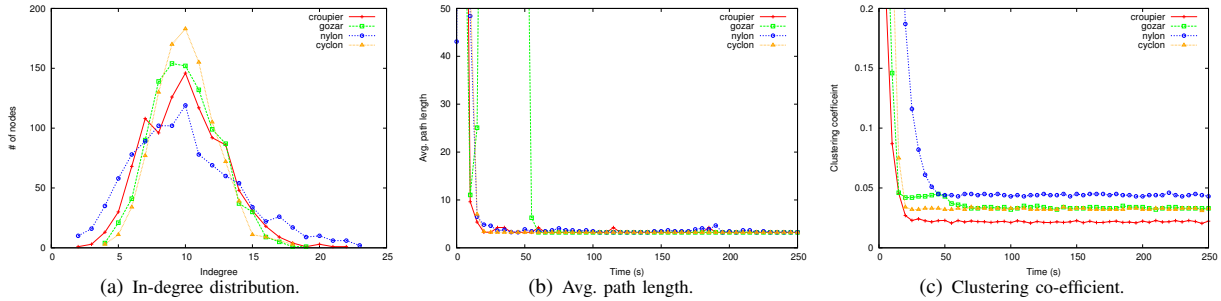
expect that a few private nodes may have poor ratio estimates.

Impact of churn on estimation

Node membership in large-scale distributed systems is typically subject to continuous change, in a process called churn. We model churn by replacing a fixed fraction of randomly selected public and private nodes with new nodes at each gossiping round, but keeping the ratio of public to private nodes stable. The churn rate is set to a level common for P2P systems [23]: assuming a gossip round-time of one second and a mean session duration of 15 minutes, approximately 0.1% of nodes leave the system per second and rejoin immediately as newly initialized nodes. Figures 5(a) and 5(b) show the average error and maximum error, respectively, for ratio estimation under churn. As can be seen, there is no significant effect of churn of up to 5% on the estimation algorithm. This rate of churn is 50 times higher than rates measured in [23].

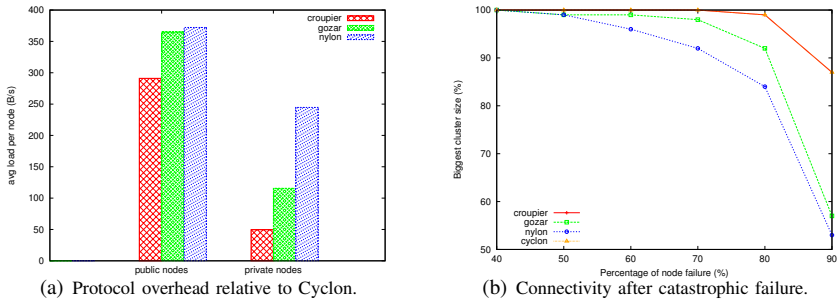
C. Peer sampling evaluation

In this subsection, we evaluate the performance of the PSS, which builds on the estimation protocol for its correct functioning.



(a) In-degree distribution. (b) Avg. path length. (c) Clustering co-efficient.

Fig. 6. Randomness properties.



(a) Protocol overhead relative to Cyclon. (b) Connectivity after catastrophic failure.

Fig. 7. Protocol overhead and connectivity under massive failure.

Measuring PSS Randomness

Here, we compare the randomness of the PSS of Croupier with Gozar and Nylon. Cyclon is used as a baseline for true randomness. In the first experiment we measure the in-degree distribution over the nodes in the all four systems. Figure 6(a) shows the in-degree distribution of nodes after 250 rounds (the out-degree of all nodes is 10). In a uniformly random system, we expect that the in-degree is distributed uniformly among all nodes. Cyclon shows this behaviour as the node in-degree is almost distributed uniformly among nodes. We can see the same distribution in Croupier, as well as, in Gozar and Nylon - their in-degree distributions are very close to Cyclon.

In figure 6(b), 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 6(b) also shows the average path length for the system in different rounds. Here, we can see the average path length of Croupier, Gozar and Nylon track Cyclon very closely. As we can see, in the first few rounds, the path length of Gozar is high, as this is the time that nodes need to find their partners used for relaying.

Finally, we compare the clustering coefficient of the systems. A node's clustering coefficient shows at what level the neighbours of a node are also neighbours of each other. For a complete graph, it is 1, and for a tree, where there is the no connection between any two neighbours of a node, it is 0. We calculate the average clustering coefficient as the average across all nodes in the system. Figure 6(c) shows the evolution of the clustering coefficient of the constructed overlay by each system. We can see that Croupier has smaller clustering coefficient that Gozar, Nylon and Cyclon. Our understanding of why Croupier has a smaller clustering coefficient is as follows. Since a private node in Croupier exchanges its view only with a public node, two private nodes never have a chance to exchange their neighbour list directly. Therefore, the probability that two private node establish a connection with each other's neighbours decreases. Since in our experiments 80% of nodes are private nodes, the average clustering coefficient in the overlay also decreases.

Protocol overhead

An important objective for any PSS is to minimize communication costs and to bound the extra overhead on public nodes (and achieve fairness). The network traffic exchanged by a node in Croupier is proportional to the rate of gossiping, as message sizes are bounded. Every node, both public and private, send one message per round. Private nodes receive one message per round (the response to the message they sent). On average, every public node receives one message from a public node per round, one response to a message they sent per round, and n messages from private nodes per round (where n is the ratio of private nodes to public nodes).

In this experiment, we set the local history α to 25, and the neighbour history length γ to 100. As in the other experiments, we bounded the number of estimations piggybacked on *shuffle requests* to 10. Each estimation required 5 bytes: two bytes for the node identifier, one byte each for the public and private counts, and one for the timestamp. The steady-state overhead is shown in figure 7(a). As we can see in figure 7(a), the public node overhead in Croupier is less than that of Gozar and Nylon. Interestingly, the overhead of private nodes, which are 80% of the nodes, is less than half compared to Gozar,

and less than one fourth compared to Nylon. As such, we conclude that the overhead on public nodes is not excessive, and our goal of fairness to public nodes has been achieved.

Connectivity after catastrophic failure

We finally evaluate the behaviour of Croupier if high numbers of nodes leave the system or crash at a single instant in time. We measure the size of biggest cluster after a catastrophic failure. Figure 7(b) shows the size of biggest cluster for Croupier, Gozar and Nylon for varying percentages of private nodes, when varying numbers of nodes fail. We can see that Croupier is more resilient to node failure than both Gozar and Nylon. For example, in the case of 80% private nodes, when 90% of the nodes fail, the biggest cluster still covers more than 85% of the nodes, while it covers 57% and 53% of nodes in Gozar and Nylon, respectively.

VIII. Conclusion

In this paper, we presented *Croupier*, the first NAT-friendly gossip-based peer sampling service that is built without relaying. Public nodes act as Croupiers, shuffling views amongst one another as well as on behalf on private nodes. Our main insight was to partition a node's view into two parts: a public view and a private view. This decision, however, necessitated that we could identify a node as being either public or private, and that nodes have a local estimation of the ratio of public to private nodes in the system. To solve these problems, we presented a minimal, distributed algorithm for the identification of a node's NAT type, as well a protocol to estimation the public/private ratio that piggybacks on existing Croupier shuffle messages. We showed in simulation that Croupier preserves the randomness properties of a gossip-based peer sampling service. We also showed that the protocol overhead in our system is less than that of existing NAT-aware PSS' and that it is more robust to large-scale failure than existing PSS'. We also showed that the extra overhead incurred by public nodes is acceptable. In future work, we will integrate our existing P2P video-streaming and video-on-demand applications with Croupier, and evaluate their behaviour on the open Internet.

Acknowledgements

The authors would like to acknowledge the contributions of Salman Niazi to early work on the protocol. The research leading to these results has received funding from KTH's TNG initiative.

References

[1] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec, "Lightweight probabilistic broadcast," in *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 443–452.

[2] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-based aggregation in large dynamic networks," *ACM Trans. Comput. Syst.*, vol. 23, no. 3, pp. 219–252, 2005.

[3] M. Jelasity, A. Montresor, and O. a. Babaoglu, "T-Man: Gossip-based fast overlay topology construction," *Computer Networks*, vol. 53, no. 13, pp. 2321–2339, 2009.

[4] J. Sacha, J. Dowling, R. Cunningham, and R. Meier, "Discovery of stable peers in a self-organising peer-to-peer gradient topology." in *6th IFIP WG 6.1 International Conference Distributed Applications and Interoperable Systems (DAIS)*, F. Eliassen and A. Montresor, Eds., vol. 4025, Bologna, Jun. 2006, pp. 70–83.

[5] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. 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*. New York, NY, USA: Springer-Verlag New York, Inc., 2004, pp. 79–98.

[6] S. Voulgaris, D. Gavidia, and M. V. Steen, "Cyclon: Inexpensive membership management for unstructured p2p overlays," *Journal of Network and Systems Management*, vol. 13, p. 2005, 2005.

[7] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "Gossip-based peer sampling," *ACM Trans. Comput. Syst.*, vol. 25, no. 3, p. 8, 2007.

[8] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulie, "Peer-to-peer membership management for gossip-based protocols," *IEEE Transactions on Computers*, vol. 52, p. 2003, 2003.

[9] A.-M. Kermarrec, A. Pace, V. Quema, and V. Schiavoni, "Nat-resilient gossip peer sampling," in *ICDCS '09: Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 360–367.

[10] A. H. Payberah, J. Dowling, and S. Haridi, "Gozar: Nat-friendly peer sampling with one-hop distributed nat traversal." in *DAIS*, ser. Lecture Notes in Computer Science, P. Felber and R. Rouvoy, Eds., vol. 6723, 2011, pp. 1–14.

[11] 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)*, San Jose, CA, U.S.A., 2010, p. (to appear).

[12] S. Niazi and J. Dowling, "Usurp: Distributed nat traversal for overlay networks." in *DAIS*, ser. Lecture Notes in Computer Science, P. Felber and R. Rouvoy, Eds., vol. 6723, 2011, pp. 29–42.

[13] L. D'Acunto, J. Pouwelse, and H. Sips, "A measurement of nat and firewall characteristics in peer-to-peer systems," in *Proc. 15-th ASCI Conference*, L. W. Theo Gevers, Herbert Bos, Ed. Advanced School for Computing and Imaging (ASCI), June 2009, pp. 1–5.

[14] B. Li, Y. Qu, Y. Keung, S. Xie, C. Lin, J. Liu, and X. Zhang, "Inside the new coolstreaming: Principles, measurements and performance implications," in *INFOCOM*, 2008, pp. 1031–1039.

[15] N. Drost, E. Ogston, R. V. van Nieuwpoort, and H. E. Bal, "Arrg: real-world gossiping," in *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*. New York, NY, USA: ACM, 2007, pp. 147–158.

[16] K. P. Gummadi, S. Saroiu, and S. D. Gribble, "King: Estimating latency between arbitrary internet end hosts," in *SIGCOMM Internet Measurement Workshop*, 2002.

[17] C. C. Moallemi and B. V. Roy, "Consensus propagation," *IEEE Transactions on Information Theory*, vol. 52, pp. 4753–4766, 2006.

[18] J. Sacha, J. Napper, C. Stratan, and G. Pierre, "Adam2: Reliable distribution estimation in decentralised environments," in *ICDCS*, 2010, pp. 697–707.

[19] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy, "Rfc 3489: Stun - simple traversal of user datagram protocol (udp) through network address translators (nats)," in *IETF RFC*, 2003.

[20] R. Roverso, S. El-Ansary, and S. Haridi, "Natcracker: Nat combinations matter," in *ICCCN '09: Proceedings of the 2009 Proceedings of 18th International Conference on Computer Communications and Networks*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–7.

[21] C. Arad, J. Dowling, and S. 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*. New York, NY, USA: ACM, 2009, pp. 1–9.

[22] G. Schay, *Introduction to probability with statistical applications*. Birkhäuser, 2007.

[23] D. Stutzbach and R. Rejaie, "Understanding churn in peer-to-peer networks," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, ser. IMC '06. New York, NY, USA: ACM, 2006, pp. 189–202. [Online]. Available: http://doi.acm.org/10.1145/1177080.1177105