Modeling the Performance of Ring Based DHTs in the Presence of Network Address Translators

John Ardelius[1] and Boris Mejías[2]

[1] Swedish Institute of Computer Science
john@sics.se
[2] Université catholique de Louvain
boris.mejias@uclouvain.be

Abstract. Dealing with Network Address Translators (NATs) is a central problem in many peer-to-peer applications on the Internet today. However, most analytical models of overlay networks assume the underlying network to be a complete graph, an assumption that might hold in evaluation environments such as PlanetLab but turns out to be simplistic in practice. In this work we introduce an analytical network model where a fraction of the communication links are unavailable due to NATs. We investigate how the topology induced by the model affects the performance of ring based DHTs. We quantify two main performance issues induced by NATs namely large lookup inconsistencies and increased break-up probability, and suggest how theses issues can be addressed. The model is evaluated using discrete based simulation for a wide range of parameters.

1 Introduction

Peer-to-peer systems are widely regarded as being more scalable and robust than systems with classical centralised client-server architecture. They provide no single point of failure or obvious bottlenecks and since peers are given the responsibility to maintain and recover the system in case of departure or failure they are also in best case self-stabilising.

However, many of these properties can only be guaranteed within certain strong assumptions, such as moderate node churn, transitive communication links, accurate failure detection and NAT transparency, among others. When these assumptions are not met, system performance and behaviour might become unstable.

In this work we are investigating the behaviour of a peer-to-peer system when we relax the assumption of a transitive underlying network. In general, a set of connections are said to be *non-transitive* if the fact that a node A can talk to node B, and B can talk to C does *not* imply that node A can talk to C. Non-transitivity directly influences a system by introducing false suspicions in failure detectors since a node cannot a-priori determine if a node has departed or is unable to communicate due to link failure.

In practice this study is motivated by the increasing presences of Network Address Translators (NATs) on today's Internet [1]. As many peer-to-peer protocols are designed with open networks in mind, NAT-traversal techniques are becoming a common tool in system design [13].

One of the most well studies peer-to-peer overlays, at least from a theoretic point of view, is the distributed hash table (DHT) Chord [15]. Chord and other ring based DHTs are especially sensitive to non-transitive networks since they rely on the fact that each participating node needs to communicate with the node succeeding it on the ring in order to perform maintenance. Even without considering NATs the authors of Chord [15], Kademlia [10], and OpenDHT [12], experienced the problems of non-transitive connectivity when running their networks on PlanetLab [3], where all participating peers have public IP address. Several patches to these problems have been proposed [3] but they only work with if the system has very small amount of non-transitive links, as in PlanetLab, where every node has a public IP address.

In this work, we construct an analytic model of the ring-based DHT, Chord, running on top of non-transitive network under node churn. Our aim is to quantify the impact a non-transitive underlay network to the performance of the overlay application. We evaluate the systems working range and examine the underlying mechanisms that causes its failure in terms of churn rate and presence of NATs. Our results indicate that it is possible to patch the Chord protocol to be robust and provide consistent lookups even in the absence of NAT-traversal protocols. Our main contributions are:

- Introduction of a new inconsistency measure, Q, for ring based DHT's. Using this metric we can quantify the amount of potential lookup inconsistency for a given parameter setting.
- Quantification of the load imbalance. We show that in the presence of NATs, nodes with open IP addresses receive unproportional amounts of traffic and maintenance work load.
- A novel investigation of an inherent limitation for the number of nodes that can join the DHT ring. Since each node needs to be able to communicate with its successor the available key range in which a node behind a NAT can join is limited.
- Evaluation of two modifications to the original Chord protocol, namely predecessor list routing and the introduction of a recovery list containing only peers with open IP addresses.

Throughout the paper we will use the words *node* and *peer* interchangeably. We will also use the term *NATed* or expressions such as *being behind a NAT* for a node behind a non-traversable NAT. It simply means that two nodes with this property are unable to communicate directly. Section 2 discusses related work, which justifies our design decisions for the evaluation model, which is discussed in Section 3. We present our analysis on lookup-consistency and resilience in Sections 4 and 5 respectively. The paper concludes by discussing some limitations on the behaviour of Chord and ring based DHTs in general.

2 Related Work

Understanding how peer-to-peer systems behave on the Internet has received a lot of attention in the recent years. The increase of NAT devices has posed a big challenge to

[3] PlanetLab: An open platform for developing, deploying, and accessing planetary-scale services. `http://www.planet-lab.org`

system developers as well as those designing and simulating overlay networks. Existing studies are mostly related to systems providing file-sharing, voice over IP, video-streaming and video-on-demand. Such systems use overlay topologies different from Chord-like ring, or at most they integrate the ring as one of the components to provide a DHT. Therefore, they do not provide any insight regarding the influence of NAT on ring-based DHTs.

A deep study of Coolstreaming [8], a large peer-to-peer system for video-streaming, shows that at least 45% of their peers sit behind NAT devices. They are able to run the system despite NATs by relying on permanent servers logging successful communication to NATed peers, to be reused in new communication. In general, their architecture relies on servers out of the peer-to-peer network to keep the service running. A similar system, PPLive, that in addition to streaming provides video-on-demand. Their measurements on May 2008 [4] indicates 80% of peers behind NATs. The system also uses servers as loggers for NAT-traversal techniques, and the use of DHT is only as a component to help trackers with file distribution.

With respect to file-sharing, a study on the impact of NAT devices on BitTorrent [9] shows that NATed peers get an unfair participation. They have to contribute more to the system than what they get from it, mainly because they cannot connect to other peers behind NATs. It is the opposite for peers with public IP addresses because they can connect to many more nodes. It is shown that the more peers behind NATs, the more unfair the system is. According to [5], another result related to BitTorrent is that NAT devices are responsible for the poor performance of DHTs as "DNS" for torrents. Such conclusion is shared by apt-p2p [2] where peers behind NATs are not allowed to join the DHT. Apt-p2p is a real application for software distribution used by a small community within Debian/Ubuntu users. It uses a Kademlia-based DHT to locate peers hosting software packages [10]. Peers behind NATs, around 50% according to their measurements, can download and upload software, but they are not part of the DHT, because they break it. To appreciate the impact NAT devices are having on the Internet, apart from the more system specific measurements referenced above, we refer to the more complete quantitative measurements done in [1]. Taking geography into account, it is shown that one of the worst scenarios is France, where 93% of nodes are behind NATs. One of the best cases is Italy, with 77%. Another important measurement indicates that 62% of nodes have a time-out in communication of 2 minutes, which is too much for ring-based DHT protocols. Being aware of several NAT-traversal techniques, the problem is still far from being solved. We identify Nylon [6] as promising recent attempt to incorporate NATs in the system design. Nylon uses a reactive hole punching protocol to create paths of relay peers to set-up communication. In their work a combination of four kinds of NATs is considered and they are being able to traverse all of them in simulations and run the system with 90% of peers behind NATs. However, their approach does not consider a complete set of NAT types. The NATCracker [13] makes a classification of 27 types of NATs, where there is a certain amount of combinations which cannot be traversed, even with the techniques of Nylon.

3 Evaluation Model

3.1 Chord model

Chord [15] is a called distributed hash table (DHT) which provides a key-value mappings and a distributed way to receive the value for a specific key, a *lookup*. The keys belongs to the range $[0 : 2^K[$ ($K = 20$ in our case). Each participating node is responsible for a subset of this range and stores the values that those keys map to. It is important that this responsibility is strictly divided among the participating peers to avoid inconsistent lookups. In order to achieve this each node contains a pointer to the node responsible for the range succeeding its own, its *successor*. Since the successor might leave the system at any point a set of succeeding nodes are stored in a *successor list* of some predefined length.

Lookups are performed by relying the message clockwise along the key range direction. When the lookup reaches the node preceding the key it will return the identifier of its successor as the owner of the key. In order to speed the process up some shortcuts are created known as *fingers*. The influence of NATs to the fingers is limited and are only discussed briefly in this work. Each nodes performs maintenance at asynchronous regular intervals. During maintenance the node pings the nodes on its successor list and removes those who have left the system. In order to update the list the node queries its successor for its list and appends the successors id.

3.2 NAT model

In order to study the effect of NATs we construct a model that reflects the connectivity quality of the network.

We consider two types of peer nodes: *open* peers and *NATed* peers. An open peer is a node with public IP address, or sitting behind a traversable-NAT, meaning that it can establish a direct link to any other node. A NATed peer is a node behind a NAT that cannot be traversed from another NATed peer, or that it is so costly to traverse, that it is not suitable for peer-to-peer protocols. Open peers can talk to NATed peers, but NATed peers cannot talk with each other. In the model, when joining the system, each node has a fixed probability p of being a NATed peer. The connectivity quality q of a network, defined as the fraction of available links will then be:

$$q = 1 - p^2 = 1 - c \tag{1}$$

Where c is the fraction of unavailable links in the system. Proof of equation 1 is straightforward and can be found in appendix [4].

We assume, without loss of generality, that all non-available communication links in the system are due to NATs. In practice we are well aware of the existence of several more or less reliable NAT-traversal protocols. [13] provides a very good overview and concludes that some NAT types, however, are still considered non-traversable (for instance random port-dependent ones) and the time delays they introduce might in many cases be unreasonable for structured overlay networks.

[4] `http://www.info.ucl.ac.be/~bmc/apx-proof.pdf`

3.3 Churn model

We use an analytic model of Chord similar to the one analysed in [7] to study the influence of churn on the ring. Namely we model the Chord system as a M/M/∞ queue containing N (2^{12} in our case) nodes, with independent Poisson distributed join, leave and stabilisation events with rates λ_j, λ_f and λ_s respectively. The fail event considers both graceful leaves and abrupt failures and the stabilisation is done continuously by all nodes in the system.

The amount of churn in the system is quantified by the average number of stabilisation rounds that a node issues in its lifetime $r = \frac{\lambda_s}{\lambda_f}$. Low r means less stabilisation and therefore higher churn. The effect of churn to Chord (studied in [7]) is mainly that the entire successor lists becomes outdated quickly for large churn rates resulting in degraded performance and inconsistent lookups. Under churn or in the presence of NATs the chain of successor pointers may not form a perfect ring. In this paper we will use the term *core ring* to denote the periodic chain of successor pointers inherent in the Chord protocol. Due to NATs, some nodes are not part of the core ring and are said to sit on a *branch*. Figure 1(a) depicts such configuration.

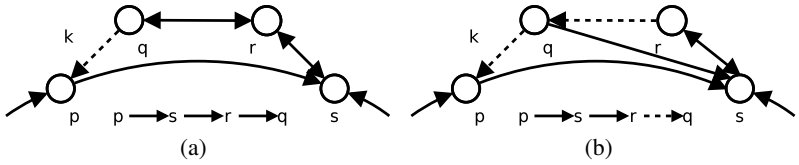


Fig. 1. The creation of branches. Dotted lines indicates that the identifier of the peer is known but unable to communicate due to an intercepting NATs. 1(a) Peers p and q cannot communication which leaves peers p and r in a *branch* rooted at peer s. 1(b) In order to route messages to peers q and r, s has to maintain pointers to both of them in its *predecessor list* since path $s \rightarrow r \rightarrow q$ is not available.

4 Lookup Consistency

One of the most important properties of any lookup mechanism is consistency. It is however a well known fact [3,14] that lookup consistency in Chord is compromised when the ring is not perfectly connected.

If two nodes queries the system for the same key they should receive identical values but due to node churn a node's pointers might become outdated or wrong before the node had time to correct them in a stabilisation round. In any case where a peer wrongly assigns its successor pointer to a node not succeeding it, a potential inconsistency is created.

The configuration in Figure 1(a) can lead to a potential lookup inconsistency in the range $]q,\ r]$. Upon lookup request for a key in that range, peer q will think r is responsible for the key whereas peer p think peer s is.

These lookup inconsistencies due to incorrect successor pointers can be both due to NATs (a node cannot communicate with its successor) or due to churn (a node is not aware of its successor since it did not maintain the pointer).

A solution proposed by Chord's authors in [3] is to forward the lookup request to a candidate responsible node. The candidate will verify its *local responsibility* by checking its predecessor pointer. If its predecessor its a better candidate, the lookup is sent backwards until reaching the node truly responsible for the key.

In order to quantify how reliable the result of a lookup really is it is important to estimate the amount of inconsistency in the system as function of churn and other system parameters. Measuring lookup inconsistency has been considered in a related but not equivalent way in [14].

We define the *responsibility range, Q* as the range of keys any node can be hold responsible for. By the nature of the Chord lookup protocol a node is responsible for a key if its preceding peer says so. From a global perspective it is then possible to ask each node who it thinks succeeds it on the ring and sum up the distance between them. However, as shown in Figure 2, the mere fact that two nodes think they have the same successor does not lead inconsistent lookups.

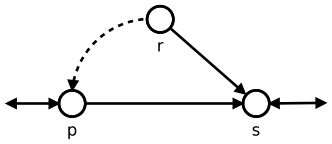


Fig. 2. A lookup for a key owned by s will not give inconsistent results due to the fact that r is not a successor of another node.

In order to have an inconsistent lookup two nodes need to think they are the immediate predecessor of the key and have different successor pointers. In order to quantify the amount of inconsistency in the system we then need to find, for each node the largest separating distance between it and any node that has it as its successor. This value is the maximum range the node can be held responsible for. The sum of each such range divided by the number of keys in the system will indicate the deviation from the ideal case where $Q = 1$. The procedure is outlined in listing 1.1.

Listing 1.1. Inconsistency calculation

```
for n in peers do
   // First alive and reachable succeeding node
   m = n.getSucc()
   d = dist(n,m)
   // Store the largest range
   m.range = max(d,m.range)
end for

globalRange = sum(m.range)
```

The responsibility range is measured for various fractions of NATs as function of churn and the results are plotted in Figure 3. We see that even for low churn rates the responsibility range after introducing the NATs are greater than 1. This indicates that on average more than one node think they are responsible for each key which causes a lot of inconsistent lookups. Important to note is also that without NATs (c=0) the churn induced inconsistency for low r is much higher than 1 .

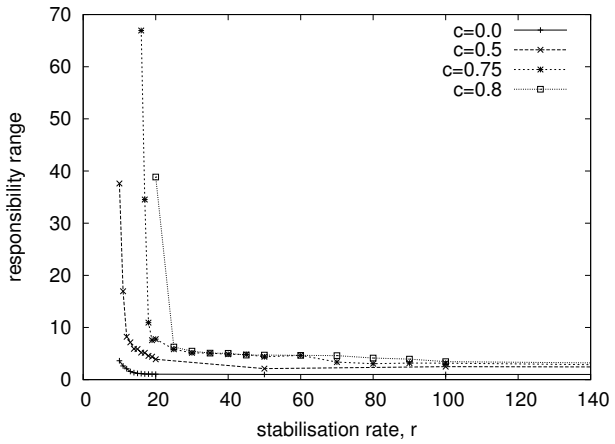


Fig. 3. Responsibility range as function of stabilisation rate for various values of c. Even for low churn, NATs introduce a large amount of lookup inconsistencies as indicated by the range being greater than 1

4.1 Predecessor list (PL) routing

The result indicates that merely assigning the responsibility to your successor does not provide robust lookups in the presence of NATs or for high churn. Using *local responsibility* instead of the successor pointer to answer lookup requests, and routing through the predecessor will ensure that only the correct owner answers the query. Because more peers become reachable, there are less lookup inconsistencies, and more peers can use their local responsibility to share the load of the address space. Again, taking the configuration in figure 1(a) as example, we can see that peer q is unreachable in traditional Chord, and therefore, the range $]p,\ q]$ is unavailable. By using the predecessor pointer for routing, a lookup for key k can be successfully handled following path $p \rightarrow s \rightarrow r \rightarrow q$. However, by only keeping one predecessor pointer, lookups will still be inconstant on configurations such as the one depicted in Figure 1(b).

To be able to route to any node in a branch, a peer needs to maintain a *predecessor list*. This is not the equivalent backward of the successor list, which is used for resilience. The predecessor list is used for routing, and it contains all nodes pointing to a given peer as its successor. If peers would use the predecessor list in the depicted examples, the predecessor list of s in Figure 1(a) would be $\{p,\ r\}$. The list in Figure 1(b) would be $\{p,\ q,\ r\}$. In a perfect ring, the predecessor list of each node contains only the preceding node. This means that the size of the routing table is not affected when the quality of the connectivity is very good. In the presence of NATs or churn, however, it is necessary to keep a predecessor list in order to enable routing on node branches . From a local perspective, a peer knows that it is the root of a branch if the size of its predecessor list is greater that one. Such predecessor list (PL) routing is used for instance in the relaxed-ring system [11] to handle non-transitive underlay networks.

In order to evaluate the performance of PL routing we define another responsibility measure, the *branch range,* Q_b. Since the PL routing protocol lets the root of a branch decide who is responsible (or whom to relay the lookup to) it is important that the root correctly knows the accumulated responsibility range of all nodes in its branch. The only way a PL lookup can be inconsistent is if two distinct root nodes on the same distance from the core ring think they should relay the lookup to one of its predecessors. The problem is depicted in Figure 4.

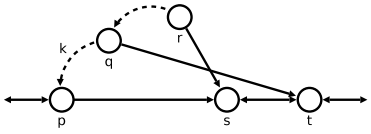


Fig. 4. Peer q misses its successor s. Peer s routes *lookup(k)* to r because it does not know q.

The branch range is calculated in a similar way as the previous predecessor range. Each node, on the core ring, queries all its predecessors except for the last one (the one furthest away) for their range. The predecessors iteratively queries their predecessors and so forth until the end of the branch is reached. Each node then returns, among all the answers from its predecessors, the end point furthest away from the root. The root, in turn, compares the returned value with the key of its last predecessor. If the returned key lies behind the last predecessor the range can give rise to an inconsistent lookup. The procedure is outlined in listing 1.2.

Listing 1.2. Branch inconsistency calculation

```
// Peers in the core ring are first
// marked by cycle detection.
for n in corePeers do
    n.range = 0;
    for m in n.preds
      // Don't add pred from core ring
      if(m!=n.preds.last)
        n.range+=m.getBranchEnd(m.key)
    end for
    n.range+=n.preds.last
end for

function getBranchEnd(p)
   for m in n.preds
      p=min(p,m.getBranchEnd(m.key))
   end for
   return p
end function

globalBranchRange = sum(m.range)
```

The branch responsibility range, Q_b, includes the former range, Q, found without PL routing and adds any extra inconsistency caused by overlapping branches. Figure 5 shows the additional responsibility range induced overlapping branches. We see that the responsibility ranges resulting from overlapping trees are orders of magnitude smaller the ranges due to problems with successor based routing. Even in the worst case the

range does not exceed 1.5 which means that at least 50% of the lookups will be reliable in worst case. Interesting also to note that the churn based inconsistency ($c = 0$) does not exceed 0.2 in any case which means the lookups are reliable to at least 80%.

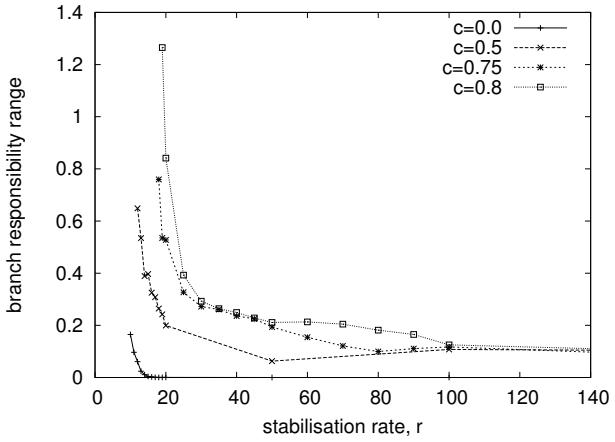


Fig. 5. Branch responsibility range as function of stabilisation rate. As the system breaks down due to high churn the trees starts to overlap resulting in additional lookup inconsistencies.

4.2 Skewed key range

In Chord, it is mandatory for the joining peer to be able to communicate with its successor in order to perform maintenance.

When joining the system a node is provided a successor on the ring by a boot-strap service. If joining node is unable to communicate with the assigned successor due to a NAT all that remains to do is for the node to *re-join* the system.

As more and more NATed peers join the network the fraction of potential successors decrease creating a skewed key distribution range available for NAT nodes trying to join the ring. This leaves joining peers behind a NAT to join only closer and closer to the root. Figure 6 depicts how the situation evolves with more NATed peers.

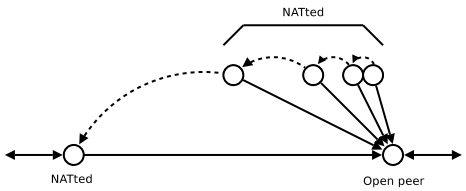


Fig. 6. Skewed distribution in a branch, shrinking the space for joining

As the fraction of NATed peers increase in the system additional join attempts are need in order to find a key succeeded by an open peer. The number of re-join attempts as function of c is shown in Figure 7(a) and the variance in Figure 7(b). Note that both the average and the variance for the number of join attempts start to grow super exponential at some critical c value ≈ 0.8 which indicates a behavioural transition between functional and congested system state.

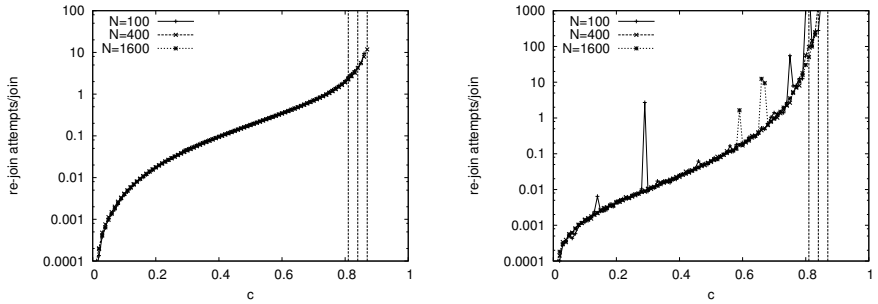


Fig. 7. Distribution of number of re-join attempts as function of c. 7(a) Average number of re-join attempts before a node is able to join the ring for different values of c. Vertical lines indicate the point where a some node in the system needs to re-try more than 10.000*N times. 7(b) Variance of the number of re-join attempts.

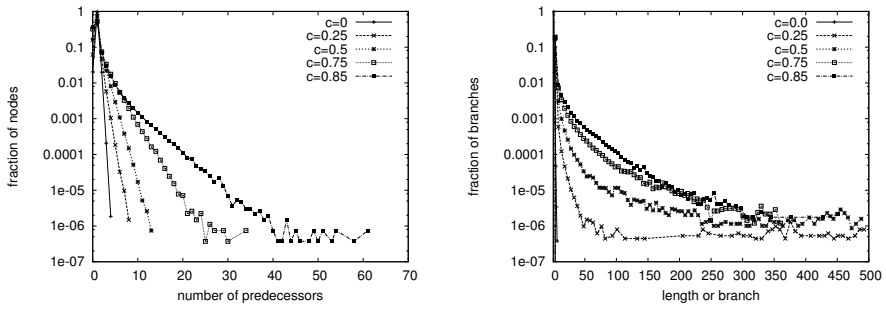
5 Resilience

5.1 Load balancing

Even for the c range where nodes can join in reasonable amount of time the open nodes on the network will be successors for an increasing number of NATed peers. Being the successor of a node implies relaying its lookups and sending it successor- and recovery lists while stabilizing. Increasing the number of predecessors therefore increase the workload of open peers.

In the case of predecessor list (PL) relays, open nodes as branch roots will be responsible to relay lookups not only to itself and its predecessors but to all nodes in its branch. Figure 8(a) shows the distribution of the amount of predecessors per node and Figure 8(b) shows the size distribution of existing branches.

For large c values we note that some nodes have almost 1% of the participating nodes a predecessors and are the root in branches of size $0.2 * N$. When such a overloaded node fails a large re-arrangement is necessary at high cost.



(a) (b)

Fig. 8. The load on the open peers increases with c as they receive more predecessors and acts as relay nodes for the branches. The system contains 2^{12} nodes. 8(a) Fraction of nodes with a given number of predecessors. For large values of c the number for the open peers become orders of magnitude higher than on the NATed peers. 8(b) Size of branches in the system for various values of c. For low values trees are rare and they have small size. As the fraction of NATs grow the length of branches grow too.

5.2 Sparse successor lists

The size of the successor list, typically $log(N)$, is the resilience factor of the network. The ring is broken if for one node, all peers in its successor list are dead. In the presence of NATed peers the resilient factor is reduced to $log(N) - n$ for nodes behind a NAT, where n is the average number of NATed peers in the successor list. This is because NATed peers cannot use other NATed peers for failure recovery. The decrease in resilience by the fraction of NATed nodes is possible to cope with for low fractions c. Since there still is a high probability to find another alive peer which does not sit behind a NAT the ring holds together. In fact the NATs can be considered as additional churn and the effective churn rate becomes $r_{eff} = r(1 - c)$

For larger values of c, however, the situation becomes intractable. The effective churn rate quickly becomes very high and breaks the ring.

5.3 Recovery list

As we previously mentioned, the resilient factor of the network decreases to $log(N) - n$. To remedy this a first idea to improve resilience is to filter out NATed peers from the successor list. However, the successor list is propagated backwards, and therefore, the predecessor might need some of the peers filtered out by its successor in order to maintain consistent key ranges.

We propose to use a second list looking ahead in the ring, denoted the *recovery list*. The idea is that the successor list is used for propagation of accurate information about peer order and the recovery list is used for failure recovery, and it only contains peers that all nodes can communicate with, that is, open peers. The recovery list is initially constructed by filtering out NATed peers from the successor list. If the size

after filtering is less than $log(N)$, the peer requests the successor list of the last peer on the list in order to keep on constructing the recovery list. Ideally, both lists would be of size $log(N)$. Both lists are propagated backwards as the preceding nodes perform maintenance. Figure 9 shows the construction of the recovery list at a NATed peer.

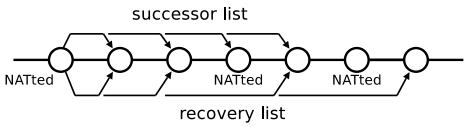
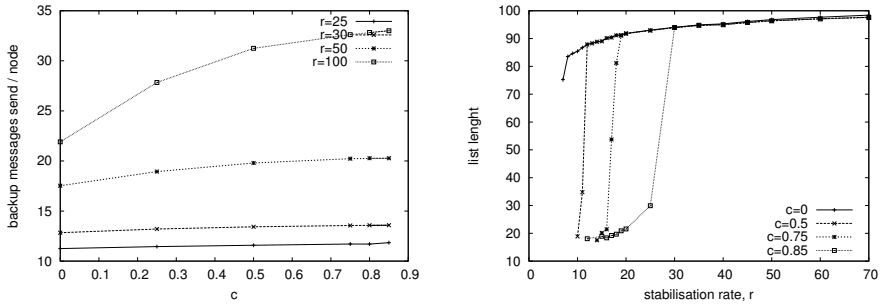


Fig. 9. Two lists looking ahead: the successor and the recovery list.

Because both lists are propagated backwards, we have observed that even for open peers is best to filter out NATed peers from their recovery lists even when they can establish connection to them. The reason is that if an open peer keeps references on its recovery list, those values will be propagated backward to a NATed peer who will not be able to use them for recovery, reducing its resilient factor, incrementing its cost of rebuilding a valid recovery list, and therefore, decreasing performance of the whole ring. If no NATed peers are used in any recovery list, the ring is able to survive a much higher degree of churn in comparison to rings only using the successor list for failure recovery.

The recovery lists are populated in a reactive manner until it finds a complete set of open peers or can only communicate with peers with overlapping set of recovery nodes. The number of messages sent during stabilisation of the recovery list is shown in Figure 10(a). For large stabilisation rates and high c values more messages are generated as the nodes need to make more queries in order to fill its recovery list.



(a) (b)

Fig. 10. 10(a) Average number of backup messages sent on each stabilisation round by a node for different values of c as function of stabilisation rate. 10(b) Size of the recovery list for various c values as function of stabilisation rate. Before break-up there is a rapid decrease in number of available pointers.

Figure 10 shows how the average size of the recovery list varies with both churn r and fraction of NATs c. In the absence of NATs ($c = 0$) the recovery list maintains about the same size over the whole r-range. In the presence of NATs on the other hand, the size of the list abruptly decreases to only a fraction of its maximum. The reason for this effect is that for high enough churn large branches tends to grow. Large part

of the ring are transferred to branches and while new nodes join they grow while the core ring shrinks. The small size of the recovery list reflects the fact that there are only a few open peers left on the core ring for high c and churn. Since the updates are done backwards and most open peers are situated in a branches nodes outside of the branch will not receive any information about their existence. The system can still function in this large branch state but becomes very sensitive to failures of open peers on the core ring.

5.4 Permanent Nodes

To make the ring resilient to extreme conditions of churn and large amount of NATed peers our investigation shows that it is necessary to introduce permanent open peers, and keep some of them in the recovery list of each peer. This strategy does not go against the self-organisation property of the system, nor against self-configuration. It only adds predefined resilient information to each peer. It can be seen as a requirement for service providers that want to guarantee the stability of the system.

Having permanent nodes is a technique already used by existing peer-to-peer systems. In apt-p2p [2], bootstrapping peers running on PlanetLab are used to keep a Kademlia ring alive. Coolstreaming [8] and PPLive [4] use logging servers to help NATed peers to establish communication with other peers. These examples indicates that using permanent nodes appear as a feasible approach to keep the service running.

6 Working range limits

Summarizing, we can categories the system behavior in the following ranges of the parameter c:

- $0 \leq c < 0.05$. In this range the influence of NATs is minor and can be seen as an effective increased churn rate. If a NATed node find itself behind another NATed node it can simply re-join the system without to much relative overhead.
- $0.05 \leq c < 0.5$. In the intermediate range the open peers are still in majority. Letting NATed nodes re-join when they obtain a NATed successor will however cause a lot of overhead and high churn. This implies that successor stabilisation between NATed peers need to use an open relay node. Multiple predecessor pointers are needed in order to avoid lookup inconsistencies and peers will experience a slightly higher workload.
- $0.5 \leq c < 0.8$. When the majority of the nodes are behind NATs the main problem becomes the inability to re-join for NATed peers. In effect, the open peers in the network will have relatively small responsibility ranges but will in turn relay the majority of all requests to NATed peers. The number of reachable nodes in the NATed nodes successor lists decrease rapidly with churn. A separate *recovery list* with only open peers is needed in addition to avoid system breakdown.
- $0.8 \leq c < 1$. In the high range the only viable solution is to let only open peers participate and have the NATed nodes directly connected to the open peers as clients. The open peers can then split workload and manage resources among its clients but are solely responsible for the key range between it and the first preceding open peer.

To make the system function for even higher churn rates and NAT ratio, our conclusion is that one should only let open peers join the network and then attach the NATed peers evenly to them as clients. Since NATed peers do more harm than good, if there are to many of them we see no other option than to leave them out. Since the open peers get most (or all) of the work load, in any case it is better to spread it evenly.

7 Conclusions

In this work we have studied a model of Network Address Translators (NATs) and how they impact the performance of ring based DHT's, namely Chord. We examine the performance gains of using *predecessor based* routing and introducing a *recovery list* with open peers. We show that adding theses elements to Chord makes the system run and function in highly dynamic and NAT constrained networks. We quantify how the necessary adjustments needed to perform reliable lookups vary with the fraction of nodes behind non-traversable NATs.

We also note that having NATed nodes per se does not dramatically increase the probability of system failure due to break-up of the ring, as long as nodes behind non-traversable NATs can use some communication relay. The main reason why the ring eventually fails due to large churn is that the branches becomes larger than the length of the successor- and recovery list. Information about new peers in the ring cannot then reach nodes at the end of the branch who's pointers will quickly become deprecated. At the same time, as branches grow large, new nodes will have a high probability of joining a branch instead of the actual ring which will worsen the situation further in a feedback that eventually breaks the system.

Our conclusion is that it is indeed possible to adjust Chord, and related ring based DHT protocols, to function in the presence of NATs without the need for traversal techniques. It further shows that it is possible to construct robust overlay applications without the assumption of an open underlying network.

8 Acknowledgements

The authors would like to thank Jim Dowling for valuable comments and discussion. The project is supported by SICS Center for Networked Systems (CNS) and the European Community's Seventh Framework Programme (FP7/2007-2013) under Grant agreement n 214898. The Mancoosi Project.

References

1. D'Acunto, L., Pouwelse, J., Sips, H.: A measurement of NAT and firewall characteristics in peer-to-peer systems. In: Theo Gevers, Herbert Bos, L.W. (ed.) Proc. 15-th ASCI Conference. pp. 1–5. Advanced School for Computing and Imaging (ASCI), P.O. Box 5031, 2600 GA Delft, The Netherlands (June 2009)
2. Dale, C., Liu, J.: apt-p2p: A peer-to-peer distribution system for software package releases and updates. In: IEEE INFOCOM. Rio de Janeiro, Brazil (April 2009)

3. Freedman, M.J., Lakshminarayanan, K., Rhea, S., Stoica, I.: Non-transitive connectivity and DHTs. In: WORLDS'05: Proceedings of the 2nd conference on Real, Large Distributed Systems. pp. 55–60. USENIX Association, Berkeley, CA, USA (2005)
4. Huang, Y., Fu, T.Z., Chiu, D.M., Lui, J.C., Huang, C.: Challenges, design and analysis of a large-scale p2p-vod system. SIGCOMM Comput. Commun. Rev. 38(4), 375–388 (2008)
5. Jimenez, R., Osmani, F., Knutsson, B.: Connectivity properties of mainline BitTorrent DHT nodes. In: Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on. pp. 262–270 (2009), http://dx.doi.org/10.1109/P2P.2009.5284530
6. Kermarrec, A.M., Pace, A., Quema, V., Schiavoni, V.: NAT-resilient gossip peer sampling. Distributed Computing Systems, International Conference on 0, 360–367 (2009)
7. Krishnamurthy, S., El-Ansary, S., Aurell, E.A., Haridi, S.: An analytical study of a structured overlay in the presence of dynamic membership. IEEE/ACM Transactions on Networking 16, 814–825 (2008)
8. Li, B., Qu, Y., Keung, Y., Xie, S., Lin, C., Liu, J., Zhang, X.: Inside the New Coolstreaming: Principles, Measurements and Performance Implications. In: INFOCOM 2008. The 27th Conference on Computer Communications. IEEE (2008)
9. Liu, Y., Pan, J.: The impact of NAT on BitTorrent-like p2p systems. In: Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on. pp. 242–251 (2009), http://dx.doi.org/10.1109/P2P.2009.5284521
10. Maymounkov, P., Mazieres, D.: Kademlia: A peer-to-peer information system based on the xor metric (2002)
11. Mejías, B., Van Roy, P.: The relaxed-ring: a fault-tolerant topology for structured overlay networks. Parallel Processing Letters 18(3), 411–432 (2008)
12. Rhea, S., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I., Yu, H.: OpenDHT: A public DHT service and its uses (2005), citeseer.ist.psu.edu/rhea05opendht.html
13. Roverso, R., El-Ansary, S., Haridi, S.: NATCracker: NAT combinations matter. Computer Communications and Networks, International Conference on 0, 1–7 (2009)
14. Shafaat, T.M., Moser, M., Schütt, T., Reinefeld, A., Ghodsi, A., Haridi, S.: Key-based consistency and availability in structured overlay networks. In: Proceedings of the 3rd International ICST Conference on Scalable Information Systems (Infoscale'08). ACM (jun 2008)
15. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proceedings of the 2001 ACM SIGCOMM Conference. pp. 149–160 (2001)