Publication VI

Jouni Mäenpää. Performance evaluation of Recursive Distributed Rendezvous based service discovery for Peer-to-Peer Session Initiation Protocol. *Elsevier Journal on Computer Networks*, Volume 56, Issue 5, pp. 1612-1626, March 2012.

Computer Networks 56 (2012) 1612–1626

Contents lists available at SciVerse ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Performance evaluation of Recursive Distributed Rendezvous based service discovery for Peer-to-Peer Session Initiation Protocol

Jouni Mäenpää *

Ericsson Research, Oy L M Ericsson Ab, Hirsalantie 11, 02420 Jorvas, Finland

A R T I C L E   I N F O

Article history:
Received 8 January 2011
Received in revised form 18 June 2011
Accepted 19 January 2012
Available online 31 January 2012

Keywords:
Recursive Distributed Rendezvous
Peer-to-Peer Session Initiation Protocol
Service discovery

A B S T R A C T

Recursive Distributed Rendezvous (ReDiR) is a service discovery mechanism for Distributed Hash Table (DHT) based Peer-to-Peer (P2P) overlay networks. One of the major P2P systems that has adopted ReDiR is Peer-to-Peer Session Initiation Protocol (P2PSIP), which is a distributed communication system being standardized in the P2PSIP working group of the Internet Engineering Task Force (IETF). In a P2PSIP overlay, ReDiR can be used for instance to discover Traversal Using Relays around NAT (TURN) relay servers needed by P2PSIP nodes located behind a Network Address Translator (NAT). In this paper, we study the performance of ReDiR in a P2PSIP overlay network. We focus on metrics such as service lookup and registration delays, failure rate, traffic load, and ReDiR's ability to balance load between service providers and between nodes storing information about service providers.

1. Introduction

Recursive Distributed Rendezvous (ReDiR) [1] is a service discovery mechanism for Distributed Hash Table (DHT) based Peer-to-Peer (P2P) overlay networks. ReDiR can be used by nodes in an overlay network to discover service providers and also to register themselves as service providers. ReDiR has been adopted as a service discovery mechanism for the Peer-to-Peer Session Initiation Protocol (P2PSIP) by the P2PSIP working group of the Internet Engineering Task Force (IETF) [2].

P2PSIP is a new distributed communication system that can be used for instance for telephony and instant messaging. P2PSIP uses the REsource LOcation And Discovery (RELOAD) protocol [2], Chord DHT algorithm [3], and Session Initiation Protocol (SIP) [4] as the core building blocks. RELOAD is used as the signaling protocol between nodes in a P2PSIP overlay network. It is used for storing and retrieving data, and for maintaining the network. RELOAD is being standardized by the P2PSIP working group of the IETF. The topology of the P2PSIP overlay is organized using the Chord algorithm, which RELOAD specifies as mandatory to implement. Nodes in the P2PSIP overlay use SIP to set up real-time communication sessions. The overlay network acts as a distributed database for SIP, mapping SIP address-of-record (AoR) values to node identifiers, which can be used to reach users participating in the system. The overlay network replaces the centralized proxy-registrar servers of traditional client/server SIP.

Nodes in a P2PSIP overlay both use and provide services. As an example, some of the publically reachable nodes in the overlay can act as Traversal Using Relays around NAT (TURN) [5] relay servers for other nodes located behind Network Address Translators (NATs). To discover and register service providers, P2PSIP relies on a service discovery mechanism. We have specified how ReDiR can be applied as a service discovery mechanism for P2PSIP in [6]. The work in this paper is highly relevant for ReDiR and P2PSIP standardization efforts in the IETF.

The goal of this paper is to study the performance of ReDiR in a P2PSIP overlay network. We focus on service lookup and registration delays, failed operations, traffic load, the distribution of lookup load, and the load of service providers. We also study the impact of various ReDiR

* Tel.: +358 9 299 3283.
E-mail address: jouni.maenpaa@ericsson.com

1389-1286/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2012.01.015

parameters and the density of service providers on ReDiR performance. Further, we develop a model for configuring ReDiR in such a way that short delays and good load balance can be achieved. The remainder of the paper is structured as follows: Section 2 gives an introduction to ReDiR. Section 3 presents related work. Section 4 describes the P2PSIP simulator used in the experiments. Section 5 describes the experiments and the traffic model used. The simulator we used in the paper is validated in Section 6. Sections 7–10 present the results of the expriments. Section 11 presents a model to assist in configuring ReDiR. Section 12 concludes the paper. The terms used in the paper are defined in Table 1.

2. Recursive Distributed Rendezvous (ReDiR)

ReDiR implements service discovery by building a tree structure of the peers that provide a particular service. ReDiR trees are service-specific; each service has its own ReDiR tree. The tree structure is embedded tree node by tree node into a DHT-based P2P network by using regular Put and Get requests provided by the DHT. Each tree node in the ReDiR tree contains pointers to peers providing a particular service. The ReDiR tree is illustrated in Fig. 1. The tree has multiple levels. Each tree node belongs to a particular level. The root of the tree contains a single node at level 0. The immediate children of the root are at level 1, and so forth. The ReDiR tree has a branching factor *b*, which is 2 in the example shown in Fig. 1. At every level *i* in the tree, there are at most b^i nodes. The nodes at any level are labeled from left to right, such that a pair *(i, j)* uniquely identifies the *j*th node from the left at level *i*. The tree is embedded into the DHT by storing the values of tree node (*i,j*) at key *H* (namespace, *i,j*), where *H* is a hash function such as SHA-1, and namespace is a string that identifies the service being provided. An example of a namespace is for instance *"turn-server"*.

Each level in the ReDiR tree spans the whole identifier space of the DHT. At each level, the ID space is divided among the tree nodes. Each tree node is further divided into *b* intervals. In Fig. 1, each node has two intervals since *b* = 2. As an example, since level 2 in the figure has 4 nodes, each of which has two intervals, the whole identifier space of the DHT is at that level divided into 8 intervals. When storing a record with identifier *k* in a given level *l*, the record is stored in the interval within whose range *k* falls. Below, we will refer to this interval as $I(l,k)$.

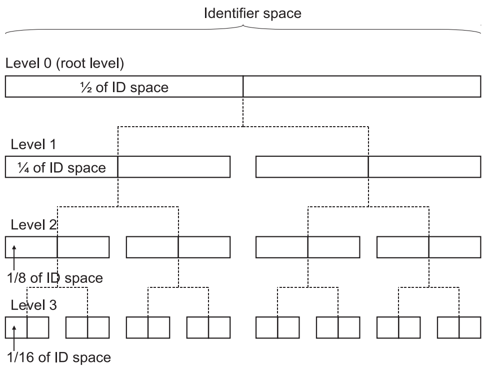

Fig. 1. ReDiR tree.

The act of storing a record in the ReDiR tree is referred to as ReDiR service registration procedure. This procedure is carried out by a service provider (SP) to store its contact information (i.e., a ReDiR data record) in the DHT. Service registration is valid for a certain amount of time, after which the SP needs to refresh its record by repeating the service registration procedure. Service registration starts from some starting level $l = l_{start}$. The default starting level is two. The input to the service registration procedure consists of the namespace of the service being registered, the key *k*, and the value that is being stored in the ReDiR tree. The first step (1) in the registration procedure is that the SP does a Get operation to obtain the contents of the tree node responsible for *k* at level *l*. The key *k* can be for instance the SP's node identifier in the Chord overlay or a randomly chosen identifier. (2) The SP also stores its record in the tree node responsible for *k* at level *l* using a Put operation. The record is stored in the interval $I(l,k)$. (3) If, after the Put operation, *k* is the lowest or highest key in the tree node, the SP starts an upward walk in the tree. During the upward walk, the SP performs Get and Put operations to store its record in higher levels of the tree (i.e., the SP repeats steps 1 and 2). The upward walk ends when the SP reaches either the root or a level at which *k* is not the lowest or highest in its interval. (4) After this, the SP performs a downward walk in the tree, at each step getting the contents of the tree node at level *l* responsible for *k*, and storing its record in the tree node if *k* is the lowest or highest key in its interval $I(l,k)$. The downward walk ends when the SP reaches a level at which it is the only SP in its interval. After this, the registration has finished.

The act of fetching a record from the ReDiR tree is referred to as ReDiR service lookup. The purpose of a service lookup for key *k* is to find the successor of *k* from the tree. The key *k* can be either a randomly chosen key or the node identifier of the node *n* performing service discovery. A service lookup starts from a starting level $l = l_{start}$. The default

Table 1
Definition of terms.

Term	Description
ReDiR tree	A tree structure of ReDiR tree nodes embedded in an overlay network
Tree node	Node in the ReDiR tree
Record	Tree nodes are stored as data records in the overlay
Peer	A network node participating in the overlay. Peers store records and tree nodes
Service Provider (SP)	Peer providing a service in the overlay
Service client	Peer using a service in the overlay
Get	The act of fetching a record
Put	The act of storing a record
Service lookup	Carried out by a service client to find an SP
Service registration	Peer registers itself as an SP in the overlay
Root	The single tree node at level 0 of the ReDiR tree

starting level is two. A service lookup consists of one or more steps. The first action (1) at each step is that n fetches the tree node associated with the current interval $I(l,k)$ using a Get request. After having fetched the tree node, n proceeds with one of the following actions: (2) if the successor of k is not present in the tree node, n sets $l = l - 1$ (i.e., proceeds with an upward walk) and performs a new Get operation (that is, n goes back to action 1). (3) If k is sandwiched between two keys (i.e., k is neither the highest nor lowest key) in $I(l,k)$, n sets $l = l + 1$ (i.e., proceeds with a downward walk) and performs a new Get (that is, n goes back to action 1). (4) If a successor is present and k is not sandwiched in its interval, the successor must be the closest successor of k and thus the service lookup is finished.

3. Related work

To the best of our knowledge, the only previous study on ReDiR performance is [1], which briefly evaluates ReDiR performance through experiments in PlanetLab. There are several differences between the work in [1] and ours. First, our work focuses on studying the suitability of ReDiR as a service discovery mechanism for P2PSIP. In contrast, the work in [1] focuses solely on OpenDHT. The P2PSIP use case (i.e., TURN server registration and discovery) is different from the OpenDHT one. Second, our ReDiR implementation follows the ReDiR specification produced by the P2PSIP working group of the IETF [6]. Third, [1] investigates only ReDiR service lookup delay and lookup consistency. We focus on a wider set of performance metrics including registration delay, failure rate, traffic load, load of service providers, and load of peers storing ReDiR tree nodes. Fourth, we also study how to configure ReDiR appropriately for different operating conditions by varying parameters such as branching factor and starting level. We also develop a model for determining appropriate values for ReDiR parameters. Fifth, the experiments in [1] are rather small-scale; five PlanetLab nodes were used and the number of clients was 256 at the maximum. In our work, the maximum number of clients is 10,000.

4. P2PSIP simulator

The results in this paper were obtained using our P2PSIP simulator, which is an event-driven, message-level simulator. It uses the same code base as our P2PSIP prototype that we have used to run experiments in PlanetLab in previous work [7–9]. We have previously used the simulator in [10]. We chose to use the simulator also in this paper because we wanted to experiment with larger network sizes than is possible to achieve in PlanetLab. The simulator is implemented in the Java programming language. It uses Peer-to-Peer Protocol (P2PP) [11] as the protocol between peers in the overlay. The RELOAD protocol [2], which is currently being standardized in the IETF, is based on P2PP (the P2PP proposal was merged with RELOAD). P2PP connections are assumed to run over an unreliable transport. The Chord DHT [3] is used to organize the overlay. Chord was chosen since the P2PSIP working group specifies it as mandatory to implement [2].

The topology generator of our simulator assigns peers randomly to 206 different locations around the world that we modeled according to PlanetLab sites. The pairwise delays between peers were set based on real pairwise delays that we measured between nodes at these sites. For the random number generators that assign peers to different sites, we used the same seeds, meaning that the same topology was used for all the simulations for a given network size. Keeping the topology constant allowed us to compare the performance of different values for ReDiR parameters. However, we also ran a series of simulations using different seeds to confirm the general validity of the results. These simulations, whose results are not included in the paper for brevity, confirm that our results remain valid even when different seeds are used. Although the topology was constant, different seeds were used for the random number generator selecting the pairwise delays from among the set of sample delays collected between each pair of PlanetLab sites.

5. Experiments

This section describes the traffic model and parameters used in our simulations. We used three different maximum network sizes (N): 100, 1000, and 10,000 peers. $N = 10{,}000$ was used as the largest network size since this was the maximum size our simulator could handle (with even larger network sizes, memory consumption became the limiting factor). $N = 100$ and $N = 1000$ were chosen since we wanted to study ReDiR performance in networks whose size is very different from $N = 10{,}000$. In each simulation run, the network was created from scratch. The network reached its maximum size at the end of the simulation. The simulated time was four hours for $N = 10{,}000$ and one hour for the other network sizes. The peer interarrival times we used in the simulations are listed in Tables 2–4, for $N = 10{,}000$, $N = 1000$, and $N = 100$, respectively. In the simulations studying the impact of branching factor, starting level, and service provider density, the network was constantly growing. No peer departures were used. This is because in these simulations, we wanted to eliminate

Table 2
User interarrival times, $N = 10{,}000$.

Start of period [s]	End of period [s]	Interarrival time [s]	N at end of period
0	1000	10	100
1000	2000	5	300
2000	4000	2.5	1100
4000	6000	2	2100
6000	13900	1	10,000

Table 3
User interarrival times, $N = 1000$.

Start of period [s]	End of period [s]	Interarrival time [s]	N at end of period
0	1000	10	100
1000	2000	5	300
2000	3750	2.5	1000

Table 4
User interarrival times, $N = 100$.

Start of period [s]	End of period [s]	Interarrival time [s]	N at end of period
0	1000	100	10
1000	2000	50	30
2000	3750	25	100

the impact of service provider and peer departures on the results. Departures would have made it more difficult to compare performance metrics such as the distribution of clients among the service providers. Although we did not use peer departures in these particular simulations, in Section 10 we will report the results of simulations whose focus was solely on studying the impact of peer departures on ReDiR performance. We have further studied the impact of peer departures and arrivals (i.e., churn) on the performance of Chord in P2PSIP overlays in previous work [7,8].

There is also another reason why we wanted the network to be constantly growing when studying the performance of different b, l_{start}, and SP density values. In P2PSIP, the most common use case for ReDiR is TURN server discovery. TURN server discovery is performed when a peer joins the overlay. Since the network was constantly growing, this ensured that there was a constant load caused by service lookups on the ReDiR tree.

The reason for choosing the specific interarrival times specified in Tables 2–4 is explained below. As already explained, we wanted the overlay to be constantly growing. The size of the overlay was increased in steps. Each step had a specific interarrival time; the smaller the network, the longer was the interarrival time. The specific interarrival time for each step was set to the maximum that a Chord ring of the given size was able to handle without the risk of becoming disconnected based on the recommendations in [12,13].

In the simulations, peers used ReDiR for TURN server registration and discovery. In the simulations studying the impact of braching factor and starting level on ReDiR performance (Sections 7 and 8,) the proportion (i.e., density) of peers capable of acting as TURN servers (i.e., publically reachable peers), was constant. In these simulations, the density of TURN servers was set to 11% based on the result in [14] according to which this was observed to be the average percentage of publically reachable Internet hosts over the countries studied. However, we also studied the impact of using different Service Provider (SP) densities (Section 9).

Following the recommendations in [3], the size of Chord's successor list and finger table were set to $\log N$ using the maximum network size, $N = 10{,}000$. In Chord, roughly $\Omega(\log^2 N)$ rounds of stabilization should occur in the time it takes for N new peers to join or $N/2$ peers to leave the overlay [13]. Chord stabilization refers to the operations that peers carry out to ensure that their routing tables stay up to date. Using the above-mentioned rule, we set the Chord stabilization interval to 15s based on the highest join rate that the overlay experienced during the simulations. The highest join rate occurred at the moment when a 300-node network faced a peer interarrival time of 2.5 s (see Table 2).

SPs, that is TURN servers, carried out the ReDiR service registration procedure upon joining the overlay. As all state is soft in the ReDiR tree, SPs refreshed their records at ten minute intervals. The ten minute refresh interval was chosen since in our simulations, we found it to be a good tradeoff between not overloading the network with ReDiR refreshes and ensuring that stale information is removed quickly enough from the overlay. A considerably more frequent refresh interval would result in there being more ReDiR traffic in the overlay than Chord stabilization traffic. This would be inefficient. Making the refresh interval considerably longer would increase the number of stale records in the overlay, resulting in an inrease in service lookup failures. Upon graceful exit, SPs removed their ReDiR records from the overlay. Users of the TURN relay service performed the ReDiR service lookup procedure upon joining the overlay.

Background (i.e., non-ReDiR) P2PSIP lookup traffic during the simulations consisted of lookups related to SIP calls and presence. Calls were modeled according to busy hour traffic volumes. Each user was assumed to initiate 13 calls per day, as suggested in [15]. 17% of these calls were used to represent busy hour traffic based on [16]. Thus, the number of busy hour call attempts per user was 2.21. This value was used as a mean rate for the arrival of calls, which was modeled as a Poisson process. Since users typically call their friends instead of strangers [17], it was assumed that $\frac{2}{3}$ of the calls are placed to users on the buddy list. The mean size of the buddy list was set to 22 based on the findings in [18]. It is worth noting that, as shown in [7], the volume of call and presence related lookup traffic is very low compared to stabilization traffic in a P2PSIP overlay. Further, this lookup traffic is completely independent from ReDiR traffic; there is no ReDiR-related signaling required when initiating calls or presence sessions. Therefore, in practice, the impact of P2PSIP background traffic on the performance of ReDiR is negligible. However, we still included background traffic in the simulations to make them more realistic. The traffic model and Chord parameters are summarized in Table 5.

The metrics we study in the experiments are delays and failure rates of service lookup and registration operations, ReDiR traffic load, Get request load, and load of SPs. Service lookup delay was measured as the time between a client sending the first Get request of a service lookup and receiving the final Get response. Registration delay was measured as the time between an SP sending the first Get

Table 5
Traffic model and Chord parameters.

Parameter	Value
Max network sizes (N)	10,000, 1000, and 100
Duration	13,900 s, 3750 s, and 3750 s
Busy hour call attempts	2.21 calls per user
% of calls to buddies	66.6
Mean size of buddy list	22
Finger pointers	14
Successors	14
Predecessors	7
Stabilization interval	30 s
ReDiR refresh interval	10 min

request of a registration operation and receiving a response to the final Put request. Service lookup failure rate was measured as the percentage of failed out of all service lookups initiated in the overlay during the measurement period. Service registration failure rate is the percentage of failed out of all registrations. ReDiR traffic load refers to all ReDiR related signaling (measured as number of bytes) exchanged in the overlay during the measurement period. Get request load refers to the number of ReDiR-related Get requests that peers storing ReDiR tree nodes receive during the measurement period. The load of an SP is the number of clients the SP is serving. We also studied how these metrics change as the ReDiR branching factor, starting level, and SP density are modified. We also studied the impact of peer and SP departures on the metrics.

6. Validating the simulator

In this section, we will validate our simulator by comparing the results obtained from it to results obtained by running our P2PSIP prototype that uses the same code base as the simulator in PlanetLab. The parameters listed in Table 5 were used both in the simulations and PlanetLab experiments with the exceptions listed below. The size of the overlay was 100 peers. The traffic model was such that during the first 1500 s of the simulations and experiments, the 100-node overlay was created from scratch. During this phase, the mean interarrival time of users was 15 s. The reason for using a rather high interarrival time was to limit the total time it takes to create a 100-node overlay. We started collecting measurement data after the overlay had reached the size of 100 peers. During the next 3600 s, both the mean interarrival time and the mean inter-departure time of peers was set to 36 s. These values were chosen since they cause the entire population of nodes in the overlay to change during the 3600 s measurement period. The topology generator of the simulator was initialized using the same set of nodes that were used in the PlanetLab experiments. Further, the delay generator of the simulator was initialized using the pair-wise delays measured between these PlanetLab nodes. The pair-wise delays were measured using six different message sizes (25, 75, 125, 250, 500, 750, and 950 bytes). The ReDiR branching factor was set to 10 and the starting level to 2 since these are the default values in ReDiR [6].

The results of the PlanetLab experiments and simulations are shown in Table 6. The values in the table are averages calculated over 20 PlanetLab experiments and 20 simulations. The values in parentheses represent 95% confidence intervals. From the table, we can observe that the results of the simulations are in general very well in line with those of the PlanetLab experiments; in the majority of cases, the differences are not statistically significant. However, in the case of the percentage of failed Get and Put requests, the differences are statistically significant; in PlanetLab, the values are significantly higher. Based on our results, the majority of Get and Put failures occur because of request timeouts. The higher failure rate of PlanetLab experiments is caused by the very dynamic load situation of PlanetLab nodes: since PlanetLab nodes are running multiple experiments of different users in parallel, the load situation of the nodes can vary dramatically during the 5100 s duration of our experiments. A node experiencing a high load may become slow in replying to P2PP messages, which results in timeouts at other nodes. We chose not to model such overload situations in our simulator since they are a characteristic of PlanetLab rather than of P2PSIP or ReDiR. Based on the results in Table 6, we can conclude that the results produced by the simulator appear to be valid compared to results obtained from a real global P2PSIP overlay.

Table 6
Validating the simulator.

Parameter	PlanetLab	Simulator
Discovery delay	1908 ms (53 ms)	1792 ms (67 ms)
Registration delay	3717 ms (97 ms)	3701 ms (91 ms)
Discovery failed	3.7% (1.7%)	2.6% (0.91%)
Registration failed	3.5% (1.5%)	2.5% (0.87%)
Gets per discovery	2.58 (0.03)	2.59 (0.02)
Gets per registration	4.03 (0.03)	4.00 (0.01)
Puts per registration	4.02 (0.02)	4.00 (0.01)
Get failed	1.7 (0.5)	0.6 (0.2)
Put failed	2.5 (0.9)	0.2 (0.1)
Get hop count	4.0 (0.2)	3.8 (0.1)
Put hop count	3.3 (0.1)	3.2 (0.1)

Finally, to show that the performance relationships between different ReDiR parameter values are similar in our simulations and in real test cases, we ran a further set of PlanetLab experiments and simulations using different values for the branching factor but otherwise using the same setup that was described above. The branching factors we used were 2, 6, 10, and 14. The results of these PlanetLab experiments and simulations (not shown here for brevity) indicate that the performance relationships between different branching factor values produced by the simulations hold also in real test cases (as one could expect based on the results in Table 6 and the fact that our P2PSIP prototype uses the same code base as our simulator).

In the sections below, we will describe the results of the simulations studying the impact of different parameter values on ReDiR performance. Section 7 studies the impact of modifying the branching factor. Section 8 studies the impact of the starting level. Section 9 studies the impact of SP density. Finally, Section 10 studies the impact of peer and SP departures. The error bars shown in the figures of these sections represent 95% confidence intervals.

7. Impact of branching factor

This section presents the impact of the branching factor on ReDiR performance, including delays, failure rates, traffic load, Get request load, and the load of SPs. In these simulations, we kept the starting level at 2, which is the default value specified by ReDiR [1].

7.1. Service lookup delay

ReDiR service lookup delay is shown in Fig. 2 for branching factors (b) ranging from 2 to 38. The figure contains results for all three network sizes. For $b = 2–14$, we

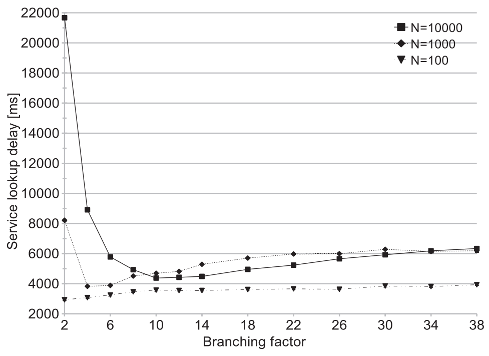

Fig. 2. Delay of ReDiR service lookups.

ran simulations for every second value of b to get a better idea of the minimum delay. From $b = 14$ onwards, we ran simulations for every fourth value. Some of the error bars are so small that they are not visible in the figure. From the figure, we can observe that the delay behaves differently for different network sizes as b is increased. This is especially visible when looking at the smallest values for b (i.e., 2–14).

To understand what causes the behavior visible in Fig. 2, we studied various factors impacting the service lookup delay, including the number of Gets per service lookup, and the average finishing level of service lookups. The results of these studies are not included here for brevity. What we found is that the main factor affecting the delay is the number of Get requests per service lookup. We also found that there is in fact a linear correlation between the number of Gets and the service lookup delay. Thus, we further studied the factors affecting the number of Gets per service lookup and found the main factor to be the density of ReDiR records in the tree at levels close to the starting level.

The reason why record density affects the number of Gets per service lookup is discussed below. Our results suggest that the larger b is, the closer to the root of the tree service lookups will finish. This is because the branching factor impacts how densely populated the ReDiR tree is. If the tree is sparsely populated at the starting level and other levels close to the root, service lookups will have a long upward walk and no downward walk at all. The upward walk is long since no successors are found from higher levels in the tree due to its sparsity. There is no downward walk because the search key is typically never sandwiched in its interval also due to the sparsity of the tree. Having no downward walk is beneficial since it decreases the service lookup delay. Therefore, when configuring ReDiR, one should aim at density of records that eliminates the downward walk and minimizes the length of the upward walk if aiming at low service lookup delays. The minimum delay is naturally achieved when there is no downward walk and the upward walk finishes already at the starting level.

To understand the conditions that cause service lookups to finish at the starting level l_{start}, it is necessary to have a look at the rules described in Section 2. Per these rules, a service lookup may finish already at l_{start} if a successor is found from the tree node fetched from l_{start} and k is not sandwiched in its interval. In such a case, the service lookup delay is optimal since the lookup finishes after only a single Get request. To reach this optimal performance, two conditions need to be satisfied. First, the average number of records per interval at l_{start} must be as close to one as possible (this ensures that k is not sandwiched in its interval). Second, at the same time, the number of records per tree node must be as high as possible (since this maximizes the probability that there is a successor present in the tree node at l_{start}). We show in Section 11 how to configure ReDiR in such a way that both of these conditions are fulfilled.

In general, our results so far suggest that configuring ReDiR can be somewhat challenging; the branching factor that results in the best performance with a given network size may become the worst possible choice if the network size changes, as can be seen from Fig. 2 for $b = 2$.

7.2. Registration delay

The delay of ReDiR registration operations is shown in Fig. 3 for different values of b and network size N. As with service lookups, the delay behaves again differently for different network sizes. As an example, the minimum registration delay is reached with different branching factors depending on the network size. We also found a linear correlation between the number of Gets (and in addition, Puts) per registration operation and the registration delay. Thus, the minimum registration delay occurs with a branching factor that results in the minimum number of Gets and Puts per registration.

For all of the network sizes, the impact of b on the upward walk of the registration procedure is as follows. If b is large, the upward walk will nearly always reach the root (because the tree nodes at levels close to the root are rather empty and k is typically the highest or lowest ID stored in the tree node). In contrast, if b is small, the starting level is rather densely populated and there is typically no upward walk at all. The impact of b on the downward walk of the registration process is as follows. If b is small, the downward walk is typically long because the tree is

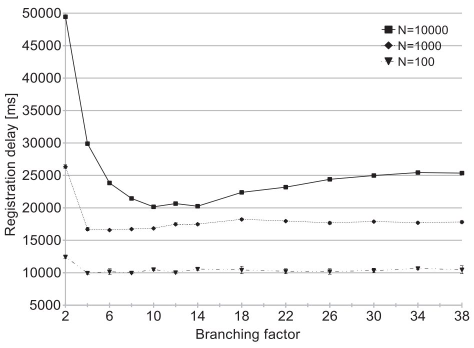

Fig. 3. Registration delay.

densely populated. However, if b is high, the downward walk will be short or there may be no downward walk at all. The smallest number of Gets and Puts is achieved when a b that minimizes the lengths of the upward and downward walks is used.

7.3. Service lookup and registration failures

The percentage of failed service lookups is shown in Fig. 4. From the figure, we can see that there are two cases in which there is a statistically significant impact on failure rate. The first case is $b = 2$ for $N = 10,000$. The second case is visible for the majority of branching factors when N increases from 1000 to 10,000. The explanation for both of these cases is that our results (not included here for brevity) show that there is a linear correlation between the failure rate and the number of Get requests per service lookup. The number of Gets per service lookup is higher for $b = 2$ than any other b when $N = 10,000$ due to the high density of records in the tree (as was discussed in Section 7.1). Also, the number of Gets per service lookup grows significantly as network size is increased. The higher is the number of Gets per service lookup, the higher is the probability that the service lookup fails. This is due to the fact that ReDiR does not by default retry failed Gets; if even one Get fails, the entire service lookup will fail.

The graphs for failed service registrations are very similar to those of failed service lookups and are thus not shown separately for brevity. Based on our results, the failure rates of service lookups and registrations are acceptable as long as the Get and Put request failure rate of the underlying DHT is acceptable. By Get and Put request failure rate of the DHT, we refer to the percentage of requests that fail due to maximum request hop count being exceeded, transaction timeouts, or routing errors caused by for instance routing loops. All of these errors were present in our simulations since they are a consequence of changes in the overlay (e.g., new peers joining) and varying delays, and did not need to be modeled separately.

7.4. ReDiR traffic load

Fig. 5 shows how large a percentage of the total traffic in the overlay is generated by ReDiR. From the figure, we

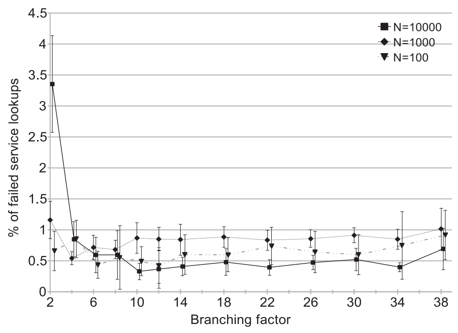

Fig. 4. Failed service lookups.

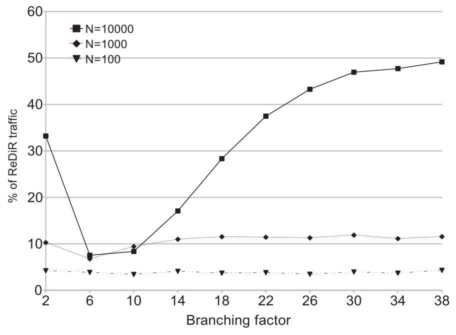

Fig. 5. Percentage of ReDiR traffic out of all traffic.

can observe that the differences between traffic volumes generated by different branching factors can be rather large. As an example, in the largest network, the amount of ReDiR traffic grows by a factor of 12 when increasing b from 6 to 38. An interesting observation for this particular case is that despite the 12-fold increase in ReDiR traffic (measured in number of bytes), the amount of ReDiR related requests increases only by 7.2% (from 157,200 to 168,500; a graph for the number of ReDiR requests is not included here for brevity). Therefore, the increase in traffic volume must be due to the increase in the average size of ReDiR messages. Further, the increase must be due to the Get response since it is the only message having a variable size. To verify this, we measured the average size of Get responses in the simulations. The results indicate that a minimum Get response size, 1153 bytes, is reached when $b = 6$. The maximum Get response size, 15,735 bytes, is reached when $b = 38$. This growth is explained by the fact that when $b = 38$, 93% of registrations and 6% of service lookups reach the root level due to the fact that the tree is very sparsely populated. For registrations, this means that nearly all registrations store a record in the root node (consequently, the root has a total of 923 records) and also fetch the contents of the root node (which explains the large average Get response size).

In contrast to $b = 38$, when $b = 6$, only 9% of registrations reach the root. The maximum number of records stored by the root node is only 71. Further, only 0.7% of service lookups reach the root. This ensures that the average Get response is much smaller for $b = 6$. The maximum, median, and 95th percentile of number of records stored per tree node is depicted in Fig. 6. In the figure, the maximum is shown in the primary y-axis, whereas the median and 95th percentile are in the secondary y-axis. From the figure, we can observe that ReDiR does not do a very good job at balancing the load of storing pointers to SPs equally among the tree nodes. Even in the best case the root node is holding over 70 times more records compared to the median.

7.5. Get request load

The percentage of Get requests received by the top 5% of the most loaded peers storing ReDiR tree nodes is shown in

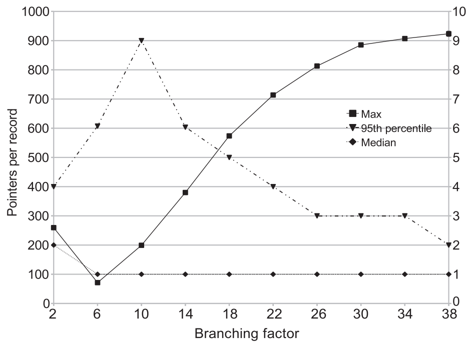

Fig. 6. Pointers per tree node.

Fig. 7. The high load experienced by the top 5% indicates that ReDiR does not distribute the load of answering Get requests equally among the peers.

We also investigated the median, average and maximum number of Gets received by peers storing ReDiR tree nodes. What we found out is that for instance for $N = 10{,}000$, the most loaded peer receives 951–7039 Gets depending on the branching factor. From $b = 6$ onwards, the most loaded peer is always one of the peers responsible for storing the root node (note that the responsibility for storing the root and other nodes may be transferred between peers in the overlay due to new peers joining the system). The most loaded peer handles alone 1.1–5.8% of all Gets depending on the branching factor. The median of received Gets is 9–11 for branching factors other than $b = 2$, for which it is 18. The standard deviation of received Gets is very large (around 100 for all branching factors; the average varies between 25–72) for all branching factors, which shows that ReDiR does indeed a poor job at balancing the load equally among tree nodes and thus also among the peers storing the tree nodes regardless of the branching factor.

It should be noted that the the results look slightly different when the overlay is static (i.e., when there is no peer arrivals and N stays constant at 10,000) since in that case the root of the tree is always stored by the same peer. In a static setup, the percentage of Gets handled by the most loaded peer was 20% in the case of the b with the worst performance ($b = 38$).

7.6. Load of service providers

To analyze how well ReDiR balances load between SPs, we studied the median and 95th percentile of number of clients per SP, and the number of SPs having no clients at all. For all of the network sizes, the average number of TURN clients per server would be 8.1 in case of optimal load balancing. However, based on our results, the median number of clients per server is 5, 5, and 6 for $N = 100$, $N = 1000$, and $N = 10{,}000$, respectively. The 95th percentiles are shown in Fig. 8.

Fig. 8 shows also the number of SPs having no clients at all. From the figure, we can observe that depending on the network size, there can be a large number of unused TURN servers. The large number of unused TURN servers explains why the median load is lower than the ideal load. From Fig. 8, we can conclude that by modifying b, one can achieve some improvement in load balancing especially in the largest network. However, even with these improvements, ReDiR still does a rather poor job at balancing the load equally among service providers.

The histogram in Fig. 9 shows a typical distribution of clients among TURN servers. The figure is from a simulation run with $N = 10{,}000$ and $b = 10$. In the ideal case, the distribution in the figure would be Gaussian with a mean around 8.1. Unfortunately, the distribution in Fig. 9 is rather far from the ideal case: there are roughly 100 servers with no clients at all and also a high number of servers having only a few clients. Also, 38% of the servers have a load higher and 45% a load lower than the ideal load (i.e., 8.1 clients). 15% of the servers have a load that is at least two times higher than the fair load (the fair load is 8.1 clients) and 40% of them have a load that is less than half of the fair load.

As a summary, branching factor has a significant impact on delays, traffic load, and the load of answering ReDiR-related Get and Put requests. However, it has a smaller impact on the load of service providers and only a minor impact on the failure rate. Also network size has a clear

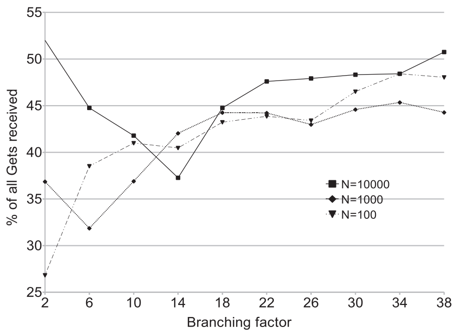

Fig. 7. Gets received by top 5% most loaded peers in ReDiR tree.

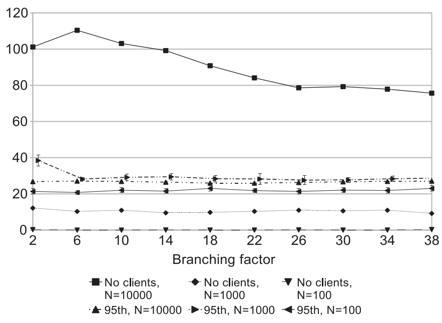

Fig. 8. TURN servers with no clients and 95th percentile of clients per server.

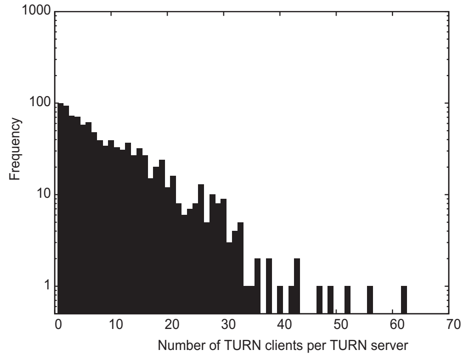

Fig. 9. TURN clients per server, $b = 10$ and $N = 10{,}000$.

impact on ReDiR performance. ReDiR seems to fail to distribute the load evenly among the service providers; many servers have an exceptionally high load. At the same time, many other servers remain rather lightly loaded or do not have any clients at all.

8. Impact of starting level

In the previous section, we studied the impact of the branching factor on ReDiR performance. In this section, we will first study the impact of choosing the starting level adaptively. Next, we will study the impact of using different fixed starting levels. The measurements were carried out using one network size, $N = 10{,}000$. The branching factor was set to 10, as this value was observed to minimize delays for $N = 10{,}000$ in the previous section.

8.1. Adaptive starting level selection

ReDiR specifies a mechanism to determine the starting level (l_{start}) adaptively [1]. In this approach, service clients take, for registrations, l_{start} to be the lowest level at which registration last completed. For service lookups, clients record the levels at which the last 16 service lookups completed and take l_{start} to be the mode of those depths. Table 7 compares the cases when the mechanism has been switched on and off. In the table, the 95% confidence intervals are shown in parentheses after each value. Surprisingly, using the mechanism results in significant degradation in performance for many of the metrics studied.

Table 7
Effect of using adaptive starting level selection.

	Adaptiveness off	Adaptiveness on
Discovery delay	4373 ms (10 ms)	4372 ms (13 ms)
Registr. delay	20162 ms (22 ms)	82577 ms (375 ms)
Discovery failed	0.33% (0.13%)	0.29% (0.10%)
Registr. failed	0.89% (0.19%)	4.55% (1.03%)
ReDiR traffic	947 MB (3 MB)	2300 MB (19 MB)
Lookups, top 5%	41.8%	39.14%
Clients, 95th	26.85 (0.17)	26.44 (0.28)

The reason why adaptive starting level selection results in degraded performance is as follows. First, the mechanism can result in a performance improvement for service lookups only if a typical client performs them rather frequently. However, this is not the case in our P2PSIP use case, since each client typically performs only one service lookup (to discover a TURN server upon joining the network). Therefore, the mechanism is not very useful in our use case. Second, we observed the mechanism to cause the starting level of registrations to be rather high in the tree. Since the starting level is high, the upward walk of registrations becomes very long, which results in increased delays and failures.

Based on these results, we can conclude that the specific adaptive starting level selection mechanism proposed in [1] should be switched off in the TURN server discovery use case and other similar use cases. We will propose an alternative mechanism for adapting the starting level in Section 11.

8.2. Service lookup and registration delays

The impact of modifying the default starting level on ReDiR service lookup and registration delays is shown in Fig. 10. From the figure, we can conclude that at least for $N = 10{,}000$ and service provider density of 11%, $b = 10$ and $l_{start} = 2$ result in the lowest ReDiR delays.

8.3. Service lookup and registration failures

Fig. 11 shows the impact of different starting levels on the percentage of failed service lookups and registrations. From the figure, we can observe that the lowest number of failures is achieved when l_{start} =2. The difference between this starting level and starting levels 0, 1, and 3 are not statistically significant.

In the case of the branching factor, we conluded that the delays and failure rates depend on the density of records in the ReDiR tree. The same explanation holds also in the case of the starting level; in fact, to reach the lowest delays and failure rates, one must find the optimal combination of b and l_{start} values for a given number of SPs. We will study how to determine the optimal combination in Section 11.

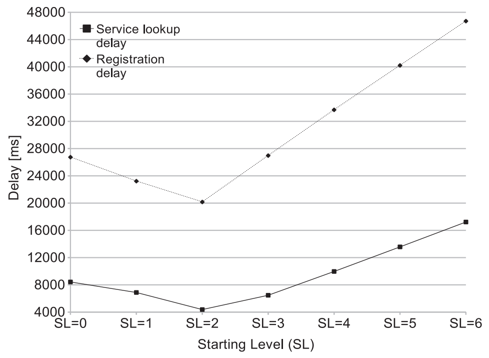

Fig. 10. Impact of starting level on delays.

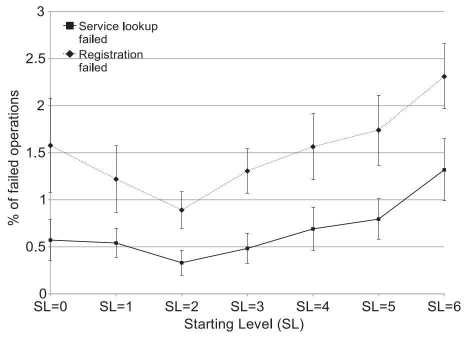

Fig. 11. Impact of starting level on failures.

8.4. ReDiR traffic

We also studied the impact of different starting levels on the traffic ReDiR generates. Our results (not shown here for brevity) indicate that $l_{start} = 0$ creates roughly seven times more traffic than any other starting level. The explanation is the same as in Section 7.4; when $l_{start} = 0$, every SP ends up storing its record in the root node, which results in that node becoming extremely large. Since also service lookups start from the root level, every client has to fetch the root node. This results in a high amount of ReDiR traffic when $l_{start} = 0$. The starting level generating the least amount of traffic (0.9 GB during the duration of the simulations) is $l_{start} = 2$. The rest of the starting levels generate roughly twice as much traffic as $l_{start} = 2$.

8.5. Lookup load and load of service providers

Fig. 12 shows the impact of the starting level on the number of Get requests that peers storing ReDiR tree nodes receive. The figure shows the maximum, 95th percentile, median, and average number of Gets received. The maximum is shown in the primary y-axis and the other values in the secondary y-axis. From the figure, we can observe that increasing l_{start} has the effect of bringing the 95th percentile of received Gets down. It also results in the median number of received Gets going up. We also studied the percentage of Get requests received by the top 5% most loaded peers storing ReDiR records. Our results (not shown here for brevity) indicate that as l_{start} increases, the top 5% become less loaded. However, even in the best case ($l_{start} = 6$), the top 5% of the peers still handle 41% of all Get requests, which cannot be considered satisfactory. Based on these results, we can conclude that increasing l_{start} seems to result in the load of answering Gets becoming slightly better distributed among the peers storing ReDiR tree nodes. The reason for this is that a larger l_{start} increases the probability that service lookups will finish higher in the tree; our results show that as l_{start} is increased, the average service lookup finish level goes up. A higher finishing level means that more Gets are handled by the large number of nodes at the higher levels of the tree, which reduces the load of the few nodes close to the root level.

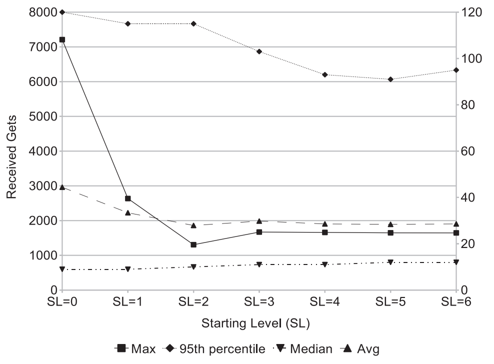

Fig. 12. Impact of starting level on Gets received.

We also studied the impact of starting level on the number of clients per TURN server. Our results (not shown here for brevity), indicate that changing the starting level has in practice no impact on the load of the TURN servers; the maximum, average, median, and 95th percentile number of clients per server stays flat regardless of the starting level.

As a summary, the adaptive starting level selection mechanism specified by ReDiR should be disabled if clients perform service lookups infrequently. Starting level can have a considerable impact on ReDiR delays and failure rates. It also effects the amount of ReDiR traffic and the distribution of Get load among peers storing tree nodes. However, l_{start} does not seem to have an impact of the distribution of load among SPs.

9. Impact of service provider density

We also wanted to understand the impact of SP density on ReDiR performance. For this, we carried out an extra set of simulations in which we varied the percentage of TURN servers from 0.1% to 90%. N was set to 10000, meaning that the number of TURN servers varied between 10 and 9000. In these simulations, we focused on comparing the delays, failures, lookup load, and load of SPs. We will not discuss the traffic load, since it simply increases in a linear fashion as the density of SPs grows. In these simulations, b was set to 10, and l_{start} to 2. These specific values were chosen for b and l_{start} since they produced good results in the previous simulations.

The impact of different SP densities on service lookup and registration delays is shown in Fig. 13 for N = 10,000. From the figure, we can conclude that in addition to a low SP density, also a high SP density increases the service lookup and registration delays considerably.

Fig. 14 shows the impact of different service provider densities on ReDiR service lookup and registration failures. Although the differences between the densities are not in all cases statistically significant, we can still see that very low and high densities result in a higher percentage of failures than the medium level densities.

When the density is 0.1%, there are only 26 nodes in the ReDiR tree and one node receives on the average 10,459

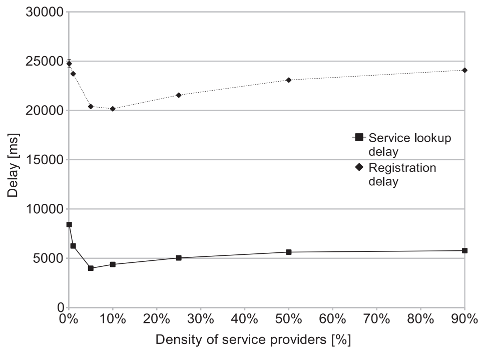

Fig. 13. Impact of service provider density on ReDiR delays.

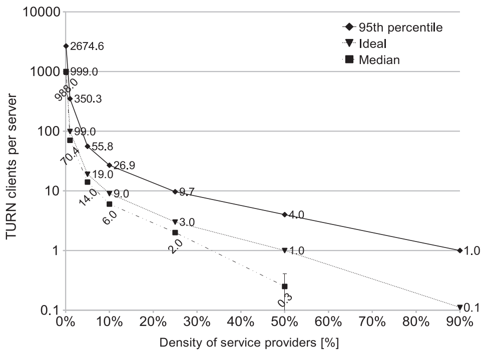

Fig. 15. Impact of service provider density on TURN clients per server.

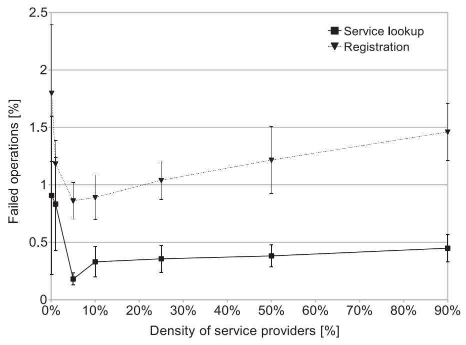

Fig. 14. Impact of service provider density on failures.

Get requests during a simulation run. When the density is increased to 90%, the number of tree nodes grows to 12143 and the number of Gets an average tree node receives drops to 363. This means that when density is 0.1%, a peer storing tree nodes receives on the average 0.75 Gets per second, and only 0.03 Gets per second when the density is 90%. Thus, we can conclude that when the density is very low, ReDiR places a considerable load on peers storing tree nodes.

The impact of TURN server density on the number of clients per TURN server is shown in Fig. 15. The figure shows the median, 95th percentile, and ideal number of clients per server. The ideal number of clients is the number of clients each TURN server would have if the load was perfectly balanced among the servers. Based on the figure, we can conclude that the higher is the density of SPs, the worse job ReDiR does at balancing the load among them. This is because the percentage of unused TURN servers increases as the density increases.

It is interesting to investigate why a very high and low SP density results in decreased performance (in terms of delays and failures). A high SP density results in increased delays since the ReDiR tree becomes so densely populated at the starting level and levels close to it that the average length of upward and downward walks increases. The increased length of upward and downward walks also explains why the percentage of failed registrations and service lookups goes up (the relationship between failures and the number of requests per registrations and service lookups was discussed in Section 7.3). Further, a very low SP density results in increased delays and failure rates since the tree becomes so sparsely populated that upward and downward walks become longer.

Based on these results, we can conclude that also SP density has an impact on ReDiR performance. One of our findings was that increasing the density does not always result in better performance; especially very high densities result in degraded performance. The reasons for this were discussed above. Regardless of the density, the load is not equally distributed among the SPs. Finally, when the density of SPs is rather low and a high number of service lookups are being carried out, the peers storing ReDiR tree nodes can experience a very high load.

10. Impact of peer departures

In this section, we will focus on studying the impact of peer and SP departures on ReDiR performance.

10.1. Growing overlay

We will first study the impact of departures in a growing overlay whose maximum size is $N = 1000$. We used the setup described in Table 5. The traffic model, which is described in Table 8, is identical to that used in earlier simulations for $N = 1000$ (see Table 3) with the exception that a peer inter-departure time was used. An inter-departure time of 28.8 s was chosen since based on Little's law, it

Table 8
User interarrival and departure times, $N = 1000$.

Start of period [s]	End of period [s]	Interarrival time [s]	Inter-departure time [s]	N at end of period
0	1000	10	0	100
1000	2000	5	28.8	265
2000	3750	2.5	28.8	904

corresponds to the mean inter-departure time of users whose average online time is 8 h (i.e., a full working day) in a 1000-peer system. For instance a company internal P2PSIP telephony network could have a user online time of 8 h. In the simulations, b was set to 10 and l_{start} to 2 since these are the default values in ReDiR. Density of SPs was 11% (i.e., the same as in earlier simulations). The purpose of the simulations was to study how peer and SP departures impact the results described in the earlier sections. The arrival and departure of peers was modeled as a Poisson process. 10% of departures were crashes, meaning that the peer or SP left the overlay ungracefully without informing other peers. A 10% crash percentage was chosen based on the assumption that the clear majority of users leave gracefully from a P2PSIP telephony system. The peers and SPs to leave and crash were selected randomly from among all the peers participating in the overlay.

The results of the simulations are described in Table 9. The table compares two scenarios: in the first one, peer departures were used, whereas in the second, there were no peer departures. The values in the table are averages calculated over 25 simulations. The numbers shown in the parentheses represent the 95% confidence intervals. From the table, we can see that the introduction of departures does not have a statistically significant impact on delays. However, the amount of failed service lookup and registration operations and the amount of failed Get and Put requests grows significantly. The most common reason for registration and service lookup failures was observed to be Get and Put request timeouts resulting from the temporary instability caused by departing and crashing peers.

10.2. Churn in an overlay having constant size

So far, we have been studying the performance of ReDiR in constantly growing overlays. In this subsection, we will study ReDiR performance in an overlay whose size remains constant over time. We studied two different scenarios. In the first scenario, the overlay was churning at such a rate that its size remained constant. The average mean online time of users was again 8 h. The size of the overlay (N) was 1000 peers. Using Little's law, this results in mean user interarrival and departure times of 28.8s. We started collecting measurement data at the moment when N reached 1000 peers and continued collecting data for 3600 s. The second scenario was otherwise identical except that there was no churn at all. We repeated the simulations 25 times. The results of these simulations are presented in Table 10. The results indicate that the presence of churn does not have a statistically significant impact on delays. In contrast, churn does have a statistically significant impact on the percentage of failed service lookup and registration operations. This is caused by a high percentage of Get requests failing due to the instability caused by peer departures. Nearly all Get requests fail due to timeouts. The percentage of failed Put requests is much lower than the percentage of failed Get requests due to the fact that the average hop count of Put requests is almost three times lower than that of Get requests.

Table 9
Impact of departures on ReDiR performance.

	Departures	No departures
Service lookup delay [ms]	5005 (176)	4961 (182)
Registration delay [ms]	17454 (305)	17951 (358)
Service lookup failed [%]	1.4 (0.3)	0.8 (0.3)
Registration failed [%]	3.8 (1.0)	1.4 (0.4)
ReDiR traffic [MB]	27.0 (1.0)	28.9 (1.2)
All departures	72.8 (2.5)	0 (0)
SP departures	17.5 (1.2)	0 (0)
Failed Get requests [%]	0.65 (0.12)	0.19 (0.05)
Failed Put requests [%]	0.24 (0.06)	0.06 (0.02)
Gets per registration	3.89 (0.02)	3.87 (0.02)
Puts per registration	3.89 (0.02)	3.87 (0.02)
% of Gets handled by top 5%	43.0	44.4
Median of clients per SP	6	7

Table 10
ReDiR performance in an overlay with constant size.

	Churn	No churn
Service lookup delay [ms]	5004 (126)	4882 (159)
Registration delay [ms]	21724 (550)	22619 (1077)
Service lookup failed [%]	0.9 (0.4)	0 (0)
Registration failed [%]	2.5 (1.2)	0.6 (0.3)
ReDiR traffic [MB]	31.5 (1.1)	36.2 (2.9)
All departures	124.9 (3.5)	0 (0)
SP departures	12.4 (1.4)	0 (0)
All joins	124.8 (4.0)	0 (0)
All SP joins	13.7 (1.0)	0 (0)
Failed Get requests [%]	0.7 (0.5)	0.15 (0.1)
Failed Put requests [%]	0.07 (0.05)	0 (0)
Gets per registration	3.89 (0.02)	3.91 (0.02)
Puts per registration	3.89 (0.02)	3.91 (0.02)
% of Gets handled by top 5%	44.7	39.4
Median of clients per SP	4	6

Note that most of the values in Tables 9 and 10 are not comparable between the tables since the duration of the simulations was different and since results from the period before N reached 1000 are not included in Table 10. However, we can still see that there are no statistically significant differences for instance in the number of Put and Get requests per registration between the different scenarios. This is because the structure of the ReDiR tree is not affected by churn (tree nodes are simply transferred between peers as peers come and go) and also because it is not affected by whether N is constant or growing.

Based on the results in this section, we can conclude that the main impact peer departures have on ReDiR performance is the increased failure rate of service lookup and registration operations. Peer departures also impact the distribution of Get load and load of SPs. This is because records stored by leaving peers and clients served by leaving SPs are transferred to some other node in the overlay, which on the average doubles the load of that node.

11. Selecting branching factor and starting level

Our results suggest that knowing the number of SPs, it is possible to determine the optimal values for b and l_{start}. We have seen that the lowest service lookup and registration delays are reached when the density of records in the ReDiR tree is neither too high nor too low, that is, when the majority of service lookups finish already at l_{start}. Thus, we can determine the parameters resulting in the lowest delays by calculating the probability that service lookups will finish at l_{start}. This probability can be calculated as follows:

$$P(\mathit{finish\ at}\ l_{start}) = P(\mathit{successor\ found}) \times P(k\ \mathit{not\ sandwiched}), \tag{1}$$

where k is the key being searched for, *P (successor found)* refers to the probability that the successor of k is found at l_{start}, and *P (k not sandwiched)* refers to the probability that k is either the highest or lowest identifier in its interval at l_{start}. *P (successor found)* can be calculated as follows:

$$P(\mathit{successor\ found}) = \frac{\frac{N_{SP}}{b^{l_{start}}}}{\frac{N_{SP}}{b^{l_{start}}}+1}, \tag{2}$$

where N_{SP} is the number of service providers. Eq. (2) and also other equations presented in this section assume uniform distribution of peer identifiers in the identifier space of the overlay. The numerator of Eq. (2) gives the average number of records $N_{rec} = \frac{N_{SP}}{b^{l_{start}}}$ per tree node at l_{start}. The denominator gives the average number of positions that a randomly chosen key can occupy within its tree node at l_{start} relative to keys already stored in the tree node. More specifically, an average tree node at l_{start} has N_{rec} records. The keys of these records divide the part of the identifier space that the tree node owns into $N_{rec}+1$ parts. There is a successor of a key k present in the tree node if k falls within the range of any other of these parts except for the last one. Further, the probability *P (k not sandwiched)* is given by:

$$P(k\ \mathit{not\ sandwiched}) = \min\left(\frac{2}{\frac{N_{SP}}{b^{l_{start}+1}}+1},\ 1\right) \tag{3}$$

The numerator in Eq. (3) is two since a given key is not sandwiched within an interval if it is either the first or last key within the interval. In the denominator, $\frac{N_{SP}}{b^{l_{start}+1}}$ gives the average number of records per interval at l_{start}. This value is incremented by one in the denominator to get the number of subintervals into which the keys of the records divide the interval.

Eq. (1) makes it possible to determine which values of l_{start} and b result in the highest probability that service lookups will finish at the starting level. This is enough to guarantee that delays stay as low as possible. However, it is not enough to guarantee a good distribution of the load of answering Get requests. To achieve a good distribution of Get load, we need to, in addition to having as many service lookups as possible finish at l_{start}, to also have as many registrations as possible to not have an upward walk. The reason to not have an upward walk is to minimize the load of the few tree nodes located at the root level and levels close to it. Although most of the service lookups finish at l_{start} if Eq. (1) has been used to select optimal parameters, typically most of service registrations will proceed all the way to the root especially if a large b is used. This causes a high load at the lowest levels of the tree. To reduce the load, one should minimize the number of registrations having an upward walk. The probability that a registration does not have an upward walk can be calculated as follows:

$$P(\mathit{no\ upward\ walk}) = 1 - P(k\ \mathit{not\ sandwiched}) \tag{4}$$

Eq. (4) follows from the fact that a registration does not have an upward walk if k is sandwiched in its interval at l_{start}. Now, using Eqs. (1) and (4), the probability that both a short delay and good distribution of Get load is achieved can be calculated as follows:

$$P(\mathit{short\ delay\ and\ good\ load\ distribution}) = P(\mathit{finish\ at}\ l_{start}) \times P(\mathit{no\ upward\ walk}) \tag{5}$$

As a summary, knowing N_{SP}, one can use different values of b and l_{start} as input to Eq. (5) to determine which combination of the parameters results in the highest probability value. These parameters can then be used to configure ReDiR in such a way that short delays and good distribution of Get load is achieved.

Naturally, P2P overlays are typically dynamic and thus the number of peers and SPs changes over time. Therefore, we will next explain how the model can be used to dynamically adapt ReDiR parameters as a response to changes in the number of SPs (N_{SP}). The number of SPs present in the overlay can be estimated as follows. As discussed above, every SP needs to store its record at l_{start}. Thus, an estimate of the number of SPs $N_{SP_{est}}$ can be calculated by fetching $M \geqslant 1$ tree nodes from l_{start} and observing the average number of records N_{rec} that the tree nodes contain. Based on this information, $N_{SP_{est}}$ can be calculated as follows:

$$N_{SP_{est}} = \frac{\sum_{i=0}^{M} N_{rec_i}}{M} \times b^{l_{start}}, \tag{6}$$

where $b^{l_{start}}$ gives the number of tree nodes at l_{start}. In our simulations, we found Eq. (6) to provide decent estimates for N_{SP}, especially when M is sufficiently large. As an example, when $M = 3$, $b = 4$, and $l_{start} = 2$, the estimate is on the average accurate within 15% of the real value of N_{SP} in a network with 1000 peers, 100 of which are SPs. In the same setup, $M = 10$, improves the estimate to be accurate within 9% of the real value. The estimate produced by Eq. (6) can be used as input to Eq. (5) to calculate the best b and l_{start} in an adaptive fashion during the runtime of a P2P overlay.

Adapting l_{start} during the runtime of a P2P overlay is rather straighforward as it does not require changes to the structure of the ReDiR tree. However, adapting b is a more complex procedure since it requires re-building the tree. The way a system can migrate to a new tree is described below. The first SP detecting the need to switch to a different b creates a new tree by storing its record using the new b. Further SPs detecting the need to migrate will add more records and tree nodes to the new tree. To ensure that other SPs can detect the need to migrate, the old tree needs to be maintained in parallel with the new tree for the duration of one ReDiR record refresh period. This is because the SPs detect the need to migrate by observing record density in the old tree. Thus, all SPs migrating to the new tree still need to refresh their record in the old tree. At the end of the refresh period, all SPs are already part of the new tree and thus the old tree can be abandoned. As a summary, the cost of the tree migration procedure is to maintain two ReDiR trees in parallel for the duration of a single refresh period. From the viewpoint of a single SP, this means performing two registration procedures instead of a single one during that period.

11.1. Validating the model

We validated the model presented in the previous subsection using simulations. We ran simulations for $N = 100$, $N = 1000$, and $N = 10{,}000$ using a large number of combinations of starting levels between 1–9 and the branching factor values used in the previous simulations (i.e., from $b = 2$ to $b = 38$). In the simulations, we observed for each b and l_{start} combination how many service lookups finish at the starting level (i.e., $P(\textit{finish at } l_{start})$) and how many registrations have no upward walk (i.e., $P(\textit{no upward walk})$) and compared these results against the model (i.e., Eq. (5)). Our results (not shown here for brevity) indicate that the model can accurately predict the best b and l_{start} combination for each network size. Thus, we can conclude that the model provides good guidance for selecting appropriate values for ReDiR parameters.

12. Conclusions and future work

In this paper, we studied the performance of ReDiR when applied as a mechanism for TURN server discovery in a P2PSIP overlay. Although we focused on the TURN server use case, our results are applicable to any popular service in a DHT-based network. We made several observations. First, ReDiR is rather difficult to configure. Configuring it inappropriately can have non-negligible impact on both ReDiR performance (i.e., on delays, traffic, and load balance) and the performance of the entire P2P overlay (e.g., due to the traffic load ReDiR generates). On one hand, configuring ReDiR so that delays are minimized can result in sub-optimal load balance. On the other hand, if the goal is optimal load balance, delays can grow dramatically. Parameters that are optimal for one network size and service provider density may perform very badly in a different setting.

If configured inappropriately, ReDiR can cause a high Get, Put, and storage load on tree nodes at the lowest levels of the tree. Especially the root node can become overloaded. If the number of service providers is low, the cost of being responsible for a tree node can be high for a peer. In such a case the traffic load caused by ReDiR on these peers is considerably higher than that caused by the DHT. To assist in problems with configuring ReDiR, we developed a model that can be used to determine parameters resulting in both short delays and good load balance. The model was validated through simulations.

If ReDiR has been configured appropriately for the number of service providers in the overlay, the average service lookup delay is only slightly higher than the delay of a single DHT Get operation. However, even in the best case, the average registration delay is multiple (typically, at least five) times longer than the delay of a DHT Put operation. The most important factor impacting ReDiR delays is the density of records at the starting level.

ReDiR does not add any additional uncertainty on top of the underlying DHT; the failure rate of service lookups and registrations is proportional to the Get and Put failure rate of the DHT. We also saw that ReDiR does not do a very good job at balancing the number of clients among service providers.

Some conclusions can also be drawn for the P2PSIP use case. First, due to the large delays associated with ReDiR service lookups, it is not practical to use ReDiR for TURN server discovery during P2PSIP call setup. Instead, TURN server discovery should either be performed in advance before the call or an existing TURN server should be used. Also, since service lookup delays can be large, it makes sense for a peer to maintain a backup TURN server to speed up the process of re-establishing connections if the primary server fails.

Based on our results, ReDiR could be improved in multiple ways. One simple way to reduce the traffic load is not to return the entire tree node (i.e., all records) in a Get response but instead return records selectively (e.g., only the record of the successor or enough information to indicate that the search key is sandwiched). This comes at the cost of extra logic required in every peer. Using the model for determining suitable branching factor and starting level proposed in this paper, one can also make ReDiR adaptive to address the difficulties associated with configuring the parameters. As an example, using the model, it would be possible for peers to adjust the starting level as a response to changes in the number of service providers. When the number of requests per operation is high, ReDiR failure rate could be reduced by retransmitting failed Gets and Puts. ReDiR would also benefit greatly from better load balancing between service providers. ReDiR also does not take service provider heterogeneity into account. We plan to study ReDiR improvements in future work.

References

[1] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, H. Yu, Opendht: a public dht service and its uses, in: SIGCOMM'05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications, ACM, New York, NY, USA, 2005, pp. 73–84. <http://dx.doi.org/http://doi.acm.org/10.1145/1080091.1080102>.

[2] C. Jennings, B. Lowekamp, E. Rescorla, S. Baset, H. Schulzrinne, REsource LOcation And Discovery (RELOAD) Base Protocol, Internet draft – work in progress, IETF, November 2010.

[3] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, H. Balakrishnan, Chord: a scalable peer-to-peer lookup protocol for internet applications, IEEE/ACM Trans. Netw. 11 (1) (2003) 17–32. <http://dx.doi.org/http://dx.doi.org/10.1109/TNET.2002.808407>.

[4] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, E. Schooler, SIP: Session Initiation Protocol, RFC 3261, 2002.

[5] R. Mahy, P. Matthews, J. Rosenberg, Traversal using relays around NAT (TURN): relay extensions to session traversal utilities for NAT (STUN), RFC 5766, April 2010.

[6] J. Mäenpää, G. Camarillo, Service discovery usage for REsource LOcation And Discovery (RELOAD), Internet draft – work in progress, IETF, July 2010.

[7] J. Mäenpää, G. Camarillo, Study on maintenance operations in a peer-to-peer session initiation protocol overlay network, in: Proceedings of IEEE IPDPS, 2009.

[8] J. Mäenpää, G. Camarillo, Analysis of delays in a Peer-to-Peer session initiation protocol overlay network, in: Proceedings of IEEE CCNC, Las Vegas, USA, 2010.

[9] J. Mäenpää, V. Andersson, A. Keränen, G. Camarillo, Impact of network address translator traversal on delays in peer-to-peer session initiation traversal, in: Proceedings of IEEE GLOBECOM, Miami, USA, 2010.

[10] J. Mäenpää, G. Camarillo, Estimating operating conditions in a session initiation protocol overlay network, in: Proceedings of IEEE IPDPS, Atlanta, USA, 2010.

[11] S. Baset, H. Schulzrinne, M. Matuszewski, Peer-to-peer protocol (P2PP), Internet draft – work in progress, IETF, November 2007.

[12] J. Mäenpää, G. Camarillo, J. Hautakorpi, A Self-tuning Distributed Hash Table (DHT) for REsource LOcation And Discovery (RELOAD), Internet draft – work in progress, IETF, January 2011.
[13] D. Liben-Nowell, H. Balakrishnan, D. Karger, Observations on the dynamic evolution of peer-to-peer networks, in: Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS'01), Springer-Verlag, London, UK, 2002, pp. 22–33.
[14] L. D'Acunto, J. Pouwelse, H. Sips, A measurement of NAT & firewall characteristics in peer-to-peer systems, in: L.W. Theo Gevers, Herbert Bos (Ed.), Proc. 15-th ASCI Conference, Advanced School for Computing and Imaging (ASCI), P.O. Box 5031, 2600 GA Delft, The Netherlands, 2009, pp. 1–5.
[15] B. Athwal, F.C. Harmatzis, V.P. Tanguturi, Replacing centric voice services with hosted voip services: an application of real options approach, in: Proc. of the 16th International Telecommunications Society (ITS) European Regional Conference, 2006.
[16] Traffic analysis for voice over ip, 2001. <http://www.cisco.com>.
[17] C. Cheng, S. Tsao, J. Chou, Unstructured peer-to-peer session initiation protocol for mobile environment, in: Proceedings of the 18th Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC'07), 2007, pp. 1–5.
[18] B.A. Nardi, S. Whittaker, E. Bradner, Interaction and outeraction: instant messaging in action, in: Proceedings of the 2000 ACM conference on Computer supported cooperative work (CSCW'00), ACM, New York, NY, USA, 2000, pp. 79–88. <http://dx.doi.org/http://doi.acm.org/10.1145/358916.358975>.

Jouni Mäenpää received his M.Sc. degree in Telecommunications Engineering from Helsinki University of Technology in 2005. He is currently the manager of Communication Services research at the NomadicLab research laboratory of Ericsson Research. He is working towards a Ph.D. degree. His research interests are in the areas of communication technologies, peer-to-peer networking, and Internet of things.