HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Communications and Networking
Networking Laboratory

Simo Sibakov

Simulating a Mobile Peer-to-Peer Network

Master's thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology.

Espoo, 23th November, 2009

Supervisor: Professor Raimo Kantola
Instructor: Lic.Sc.(Tech.) Nicklas Beijar

HELSINKI UNIVERSITY OF TECHNOLOGY	ABSTRACT OF THE MASTER'S THESIS

Author:	Simo Johannes Sibakov		
Name of the Thesis:	Simulating a Mobile Peer-to-Peer Network		
Date:	November 23th, 2009	**Number of Pages:**	x + 101
Faculty:	Faculty of Electronics, Communications and Automation		
Department:	Department of Communications and Networking		
Professorship:	S-38		
Supervisor:	Professor Raimo Kantola		
Instructor:	Lic.Sc.(Tech.) Nicklas Beijar		

Peer-to-peer (P2P) applications have become available for portable devices as the processing power and the storage capacity of the devices as well as the network bandwidth have increased. The traditional P2P applications have been considered too heavy for mobile usage. New, lightweight P2P protocols are needed for mobile P2P applications. P2P Session Initiation Protocol (P2PSIP) is a protocol under development that provides the session establishment functions of SIP in a serverless fashion.

The main objective of this thesis was to simulate a P2PSIP overlay network operating with Resource Location And Delivery (RELOAD) peer protocol and study its bandwidth consumption, lookup overhead and lookup success rate. The effect of churn, the process of nodes arriving to the overlay and leaving it, on these results was also one of our concerns. We study if this kind of network is applicable to be implemented on top of current mobile telephone networks. This study compares the performance of two distributed hash table (DHT) algorithms, Chord and Kademlia. The simulations were carried out with OverSim overlay network simulator implemented in the C++ programming language.

This study shows that P2PSIP network's bandwidth usage is low enough to operate on top of the 2G mobile telephone networks. Kademlia uses more bandwidth than Chord but it has a shorter lookup delay and a higher lookup success rate than Chord. The results show that with the parameter settings used in our study the lookup success rates are in general too low for many applications to give them a satisfying quality of service.

Keywords: Peer-to-Peer, Simulation, P2PSIP, RELOAD, Chord, Kademlia

TEKNILLINEN KORKEAKOULU		DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Simo Johannes Sibakov	
Työn nimi:	Mobiilin vertaisverkon tutkiminen simuloimalla	
Päivämäärä:	23.11.2009	**Sivuja:** x + 101
Tiedekunta:	Elektroniikan, tietoliikenteen ja automaation tiedekunta	
Laitos:	Tietoliikenne- ja tietoverkkotekniikan laitos	
Professuuri:	S-38	
Työn valvoja:	Professori Raimo Kantola	
Työn ohjaaja:	TkL Nicklas Beijar	

Vertaisverkkosovellukset (P2P-sovellukset) ovat saapuneet kannettaviin laitteisiin, kun laitteiden prosessoriteho, tallennuskapasiteetti sekä matkapuhelinverkkojen kaistanleveys on kasvanut. Perinteiset P2P-sovellukset ovat liian raskaita mobiilikäyttöön. Uusia, kevyempiä vertaisverkkoprotokollia tarvitaan mahdollistamaan P2P-sovellusten toiminta mobiiliympäristössä. P2P Session Initiation Protocol (P2PSIP) on kehitteillä oleva protokolla, jonka avulla SIP protokollan yhteydenmuodostus toteutetaan ilman palvelimia.

Tämän työn tavoitteena oli simuloinnin avulla tutkia Resource Location and Delivery (RELOAD) vertaisprotokollaa käyttävää P2PSIP päällysverkkoa. Tarkastelun kohteena olivat erityisesti kaistanleveyden tarve, hakujen kustannukset sekä hakujen onnistuminen. Myös solmujen päällysverkkoon liittymisestä ja päällysverkosta poistumisesta aiheutuvat vaikutukset tuloksiin olivat mielenkiinnon kohteina. Tämä työ vertailee kahden hajautetun tiivistetaulualgoritmin, Chordin ja Kademlian, suorituskykyä. Simulaatiot on suoritettu C++ ohjelmointikielellä toteutetulla OverSim-päällysverkkosimulaattorilla.

Tämä työ osoittaa, että 2G-matkapuhelinverkon kaistanleveys on riittävä P2PSIP-päällysverkon tarpeisiin. Kademliaa käytettäessä kaistanleveys on suurempi, mutta hakujen viive pienempi ja onnistumisprosentti suurempi kuin Chordia käytettäessä. Tulokset osoittavat, että tässä työssä käytetyillä parametreilla hakujen onnistumisprosentti on monille sovelluksille liian alhainen eikä mahdollista niille tyydyttävää palvelun laatua.

Avainsanat: Vertaisverkko, Simulointi, P2PSIP, Chord, Kademlia

Acknowledgements

This Master’s thesis has been written at the Department of Communications and Networking of Helsinki University of Technology. The thesis is part of a larger DECICOM project concentrating on mobile peer-to-peer research.

I would like to thank my instructor Nicklas Beijar for his well-informed comments and guidance throughout my work. I would also like to thank my supervisor, Professor Raimo Kantola, for sharing his views on mobile peer-to-peer and academic writing.

Espoo, November 23, 2009

Simo Sibakov

Table of Contents

List of Figures

List of Tables

List of Abbreviations

3GPP	3rd Generation Partnership Project
AP	Admitting Peer
CDF	Cumulative Distribution Function
CPU	Central Processing Unit
DHT	Distributed Hash Table
EDGE	Enhanced Data rates for GSM Evolution
EWMA	Exponential Weighted Moving Average
GPRS	General Packet Radio Service
GSM	Global System for Mobile communications
HSCSD	High-Speed Circuit-Switched Data
HSDPA	High-Speed Downlink Packet Access
HSUPA	High-Speed Uplink Packet Access
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
IANA	Internet Assigned Numbers Authority
ICE	Interactive Connectivity Establishment
IETF	Internet Engineering Task Force
IMS	IP Multimedia Subsystem
IP	Internet Protocol
JP	Joining Peer
NAT	Network Address Translation
RELOAD	Resource Location And Delivery
P2P	Peer-to-Peer
P2PSIP	Peer-to-peer SIP
PDF	Probability Density Function
PP	Predecessor Peer
RAM	Random Access Memory
RFC	Request For Comments
RTT	Round-Trip Time
SDP	Session Description Protocol
SHA-1	Secure Hash Algorithm 1
SHA-256	Secure Hash Algorithm 256

SIP	Session Initiation Protocol
SMS	Short Message Service
STUN	Session Traversal Utilities for NAT
TLS	Transport Layer Security
TTL	Time To Live
TURN	Traversal Using Relay NAT
UA	User Agent
UAC	User Agent Client
UAS	User Agent Server
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
WCDMA	Wideband Code Division Multiple Access
WG	Working Group

1 Introduction

Peer-to-Peer (P2P) networks have been a target of an intense study ever since the first file sharing applications arrived in the late 1990's. P2P has been mainly perceived as a technique for illegal file sharing over the internet. Recently also its commercial potential has been discovered and new commercial applications using P2P technique are starting to emerge.

P2P research has concentrated on making the P2P systems independent of any centralized elements but still scalable and efficient in bandwidth usage and load balancing. The latest trend in P2P research is to study the possibilities to use P2P technique in mobile devices. The mobile environment requires different features than the traditional P2P protocols that have been running on desktop and laptop environments.

The traditional P2P applications have been considered too heavy for mobile usage, demanding too much bandwidth to work effectively in the mobile environment. P2P Applications are becoming available also for portable devices and because of the limited bandwidth in mobile telephone networks, the bandwidth usage is an important factor and an essential question for a study.

Session Initiation Protocol (SIP) is a lightweight client-server protocol that needs some modifications to work in a pure P2P manner. A new protocol, P2PSIP, is under development to enable SIP functionality without any centralized entities. This thesis examines P2PSIP, particularly one of the proposed peer protocols, Resource Location And Delivery (RELOAD), and tries to investigate if P2PSIP is applicable to mobile telephone networks used today.

This study aims to answer the following questions: How much bandwidth is needed for P2PSIP to work properly. How much overhead and delay is there in the key lookup process? How does the process of nodes continuously arriving to the network and leaving it affect these output parameters? Which distributed hash table (DHT) algorithm works best with P2PSIP?

The research methods used in this thesis are literary research that covers the techniques essential for P2P overlay network simulations. We try to keep the level of details reasonable without compromising the intelligibility of the study. The actual research is carried out by simulating a P2PSIP network with the OverSim overlay network simulator. This requires some programming work to model the desired features of RELOAD into the simulator and to prepare the simulation scenarios. This study contains altogether 57 different simulation scenarios. The input parameters in this study are selected to make the scenarios describe communication composed of phone calls and short message service (SMS) messages. The parameter selection makes this study different from the other studies that have mostly been investigating file sharing scenarios. The work simulates two DHT algorithms Chord and Kademlia in both iterative and recursive routing mode. The DHT algorithms are tested as they are configured by default in OverSim and they are not tuned or developed in any way.

1.1 Scope of the thesis

In this thesis we study a peer-to-peer network operating over a mobile telephone network. The P2P protocol used in our study is P2PSIP. The P2PSIP network under study contains only peers with equal functionalities and responsibilities in the overlay. P2PSIP clients are left outside of the study. Only the parts of the RELOAD protocol that are relevant from a bandwidth usage perspective are modeled to OverSim. Minor details are bypassed because of the rather inaccurate nature of the RELOAD draft that is the main source of information for the protocol.

1.2 Outline of the thesis

This thesis consists of a theory part, Chapters 2–4, which gives the essential information for a reader to understand the phenomenon studied in the thesis. These chapters give a compact presentation about the theories behind the Overlay networking techniques in the form of a literary research. After the technical background information, Chapter 5 then presents the simulation setup used in this study. The simulation environment as well as the input and output parameters are introduced. In Chapter 6 we present the results that are divided in seven categories according to the output parameters introduced in Chapter 5. The discussion part of the thesis covers the results and the factors affecting them as well as comparison with similar studies in Chapter 7. Three interesting phenomenon about the results are pointed out. Extrapolation of the results is also presented. Chapter 8 concludes the thesis with the main findings and presents proposals for future research.

2 P2P networking principles

This Chapter presents the overlay network concept and the different P2P architectures used in P2P networks. We discuss the alternatives for the routing mode selection and present an introduction on the distributed hash table algorithms used in this study.

2.1 General information about P2P networks

P2P-overlays can be roughly divided on the grounds of the degree of centralization and the structure of the overlay as well as by the type of index used in the overlay. These characteristics affect the scalability and the resilience of the overlay, the bandwidth usage and the search coverage.

2.1.1 The Overlay concept

Overlay networks are built on top of one or more existing networks. P2P overlay networks run on top of the IP network. The links in the overlay network are virtual and they can correspond to a path with many physical links. The purpose of the P2P overlay networks is to implement services that are not available in the underlying IP network.

Because of the virtual nature of the links between the peers, the topology of the P2P overlay network also differs from the IP network it is built on. The overlay network topology is determined by a specific algorithm that defines which peers should have a virtual link between each other. [Sep07]

The peers look for resources in the P2P network. The resource may be a multimedia file or as in the P2PSIP case a text file containing a contact address. Once the resource is found, usually a session of some sort is established between the holder of the resource and the requester.

2.1.2 Unstructured architectures

Centralized

The first-generation P2P networks had a *centralized* network architecture. In these kinds of networks the query works in a client-server fashion. Figure 1 illustrates the lookup process. The requesting peer sends the query to the server pool (phase 1) which holds the information about the connected peers and their resources. The server pool then performs the search for the requested resources (2) and responds with the address of the peer holding the requested resource (3). Only the actual data transfer is made directly between the peers (4). [Har+04]

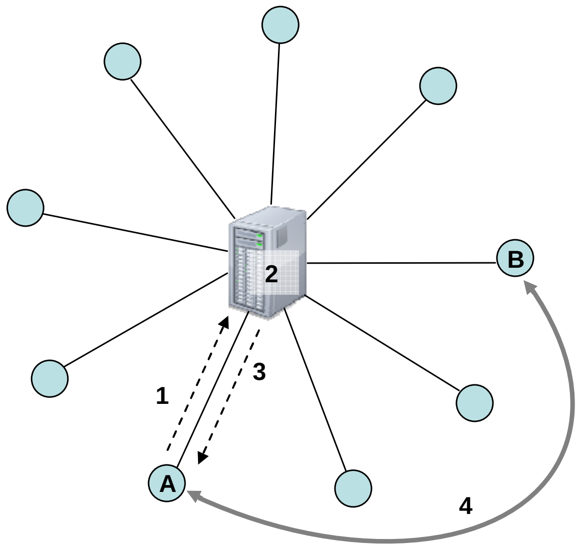


Figure 1 – Lookup in the centralized P2P network architecture

Decentralized

The first-generation networks need maintenance for the servers, do not scale well and are vulnerable because of their server-centricity. The second generation of P2P networks tried to improve these drawbacks. The solution was to build a serverless network, where every peer has equal functionality and responsibilities for routing the messages. This kind of network architecture is called *decentralized*. Searches in decentralized networks are done in a broadcast manner with a flooding algorithm where the query messages have a Time To Live (TTL) field defining the number of hops the query message can travel. An example of the lookup process in a decentralized P2P network is depicted in Figure 2. Peer A initiates a lookup and the requests are flooded in the network with the TTL field value of 2 in the first message. The requested resource is found in peer B. [Har+04]

As we can see from Figure 2, only four messages (phases 1-4) are needed to complete the lookup. However, the lookup sends lots of additional messages that do not reach any peer holding the desired resource. In large networks with thousands of peers this approach does not scale well because of the amounts of excess traffic it produces. The lookup process is also slow, because when queries are flooded only the neighbor peers can be contacted using one routing hop. To reach a more distant peer, the number of hops becomes very large. Some decentralized P2P networks also use random walk queries where the targets of the query messages are chosen in random. This improves the search efficiency but even with random walk it still remains lower than in the first generation P2P networks [Gka+04].

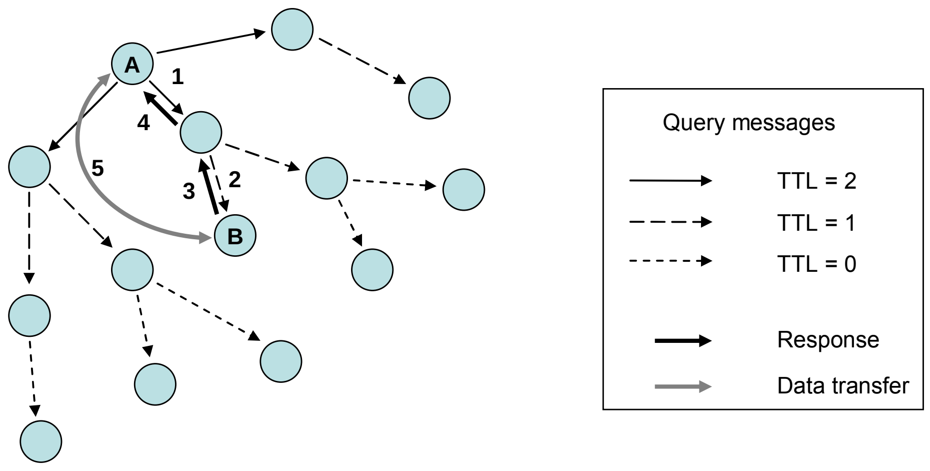


Figure 2 – Lookup in the decentralized P2P network architecture

Hybrid

Although the purely decentralized networks managed to fix many of the problems the first-generation P2P networks were facing, the lack of any centralized servers introduced new issues with overhead. The second generation P2P networks generated enormous amounts of network traffic as the lookups were done in a broadcast manner. The third generation P2P networks attacked these problems with a *hybrid* network architecture. In this solution, some peers act as super-peers that have more functionality than the ordinary peers. The super-peers have the same functionalities as the servers in the centralized P2P networks. Only the super-peers participate in the peer and resource lookup. Each peer uses one super-peer as a gateway to the P2P network. [Har+04]

Figure 3 illustrates the lookup process in a hybrid P2P network. As can be seen, from the perspective of an ordinary peer the hybrid architecture is similar to the centralized architecture (Figure 1). The difference is that in the hybrid architecture the super-peers perform the search for the requested resources (phases 2-5) instead of the server pool that was responsible for that function in the first generation P2P networks. [Har+04]

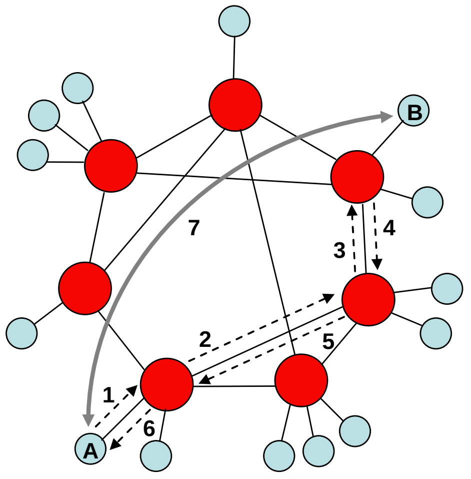


Figure 3 – Lookup in the hybrid P2P network architecture

The different P2P network architectures are compared by Lehtinen [Leh06]. This comparison is presented in Table 1. The table equates the servers of the 1st generation architecture with the super-peers of the 3rd generation architecture when comparing the signaling overhead. As we can see from the table, every generation has its advantages. The best choice of architecture depends on what the P2P network will be used for and who will be using it. If some entity wants to be in control of the network, decentralized architecture is out of the question. For centralized and hybrid networks an operator control can be included because of their dependency of servers or super-peers. [Leh06]

Table 1 – Architectural comparison of unstructured P2P networks [Leh06]

Architecture	**Decentralized (2nd gen.)**	**Hybrid (3rd gen.)**	**Centralized (1st gen.)**
Scalability	Low	*Very High*	Medium
Signaling overhead in super-peer	-	High	Very High
Signaling overhead in ordinary peer	High	Low	Low
Resiliency	*Very High*	Medium	Low
Operator control	Low	High	Very High
Search coverage	Medium	High	*Very High*

Decentralized and hybrid unstructured P2P networks cannot give any guarantees about finding a desired resource from the network with a limited amount of hops at a given moment even if the resource exists in the network. This is due to the fact that the network does not know where to look for a given resource. A desired resource can reside in any peer and the holder of this information can be any of the super-peers (in the hybrid architecture case). Because of this, the amount of hops needed to reach the target peer cannot be predicted. The routing costs can become very high regardless of the success of the lookup. The structured P2P network architectures discussed next, greatly improve these features. [Ris+07]

2.1.3 Structured architectures

In structured P2P architectures the network topology is strictly determined by an algorithm. The peer is given an identifier when it joins the network. The identifier defines the logical location of the peer in the overlay and thereby also the set of other peers it connects to. The resources that are stored in the network also get identifiers. With these identifiers the address of the node holding that resource can be found. The algorithms that are used for these purposes are called *Distributed Hash Tables* (see chapter 2.3).

Peer and content lookup is efficient in the structured P2P networks because the search mechanisms can be made simple, as based on the identifier being searched the querying node already has an idea of the location of the searched resource. Structured architectures can guarantee location of a given target within a bounded number of hops. This guarantee holds also for queries for resources that do not exist in the network. The non-existence can also be verified within the same bounded number of hops. [Ris+07]

The structured P2P networks have been criticized for their inability to handle churn and for the lack of support for keyword searches. According to [Cha+03] the keyword search is considered to be an important feature as the complete file name of the searched object is rarely known by the users of standard file sharing systems. The line between unstructured and structured architectures is becoming more blurry as most unstructured P2P networks these days have some structure built in. [Ris+07]

2.1.4 Indexes

Index is “a collection of terms with pointers to places where information about documents can be bound” as stated in [RFC4981]. P2P networks use local, centralized and distributed indexes.

Local indexes

When local indexes are used, each peer only keeps references to data it retains itself. Peers do not receive references for data that other peers are storing. Only local indexes are used in decentralized P2P network architecture (Figure 2). Search process with local indexes is based on flooded queries. Rich queries are enabled as key word search can be used. Local indexes are good if there are many replicas of the data items existing in the network. In finding a data item which exists only in a single peer in the network, local indexes are very inefficient and the discovery of that item can not be guaranteed.

Centralized indexes

In P2P networks using centralized indexes all references to the stored items are kept in a server (or server pool) to which peers can connect. There are three levels of centralization in this index category. Networks with *unchained servers* cannot provide the peers any information about the indexes in other servers so peers can exchange data only with the peers that share the same server with them. In networks using *chained servers* a query will be forwarded to other servers if the index is not found from the server the requesting peer is connected to. There are also *full replication* networks, in which all servers keep a complete index of the data stored in all of the peers. Figure 1 illustrates a network with centralized indexes.

Distributed indexes

Most P2P designs today use distributed indexes, in which the references to the stored items are located at several nodes. Chord (Section 2.3.2) and Kademlia (Section 2.3.3) are examples of P2P architectures with distributed indexes. The number of nodes a given data item is stored in depends on the number of replicas used in the system.

2.2 Routing in P2P networks

2.2.1 Iterative routing

In iterative routing an intermediate node receiving a query message, returns a response message that includes the information about the next hop node. The responsibility of actually contacting the next hop remains with the initiating node. Iterative routing is illustrated in Figure 4 where nodes are arranged in a ring shape as in the Chord protocol (Section 2.3.2). The initiating node monitors the routing process and decides how long it wants to wait for a response message before it considers the queried node absent. For a lookup path of N nodes $2N - 1$ messages are needed for reaching the target node. [Kun05]

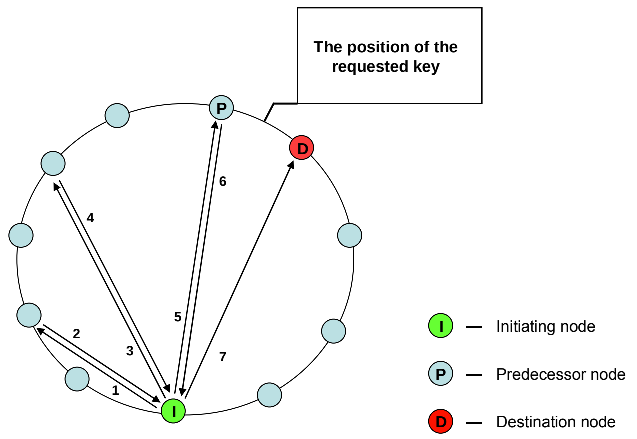


Figure 4 – Iterative routing

2.2.2 Recursive routing

Figure 5 depicts the same lookup process done in a recursive manner. The intermediate nodes simply forward the query to the next hop node without informing the initiating node. This means that the initiating node has no control over the routing after it has sent the first query message. There are different versions of recursive routing, in which the routing of the response message from the destination node to the initiating node varies. These include symmetric recursive (Figure 5), forward-only and direct response. For a lookup path of N nodes recursive routing needs N messages to reach the target node [Kun05]

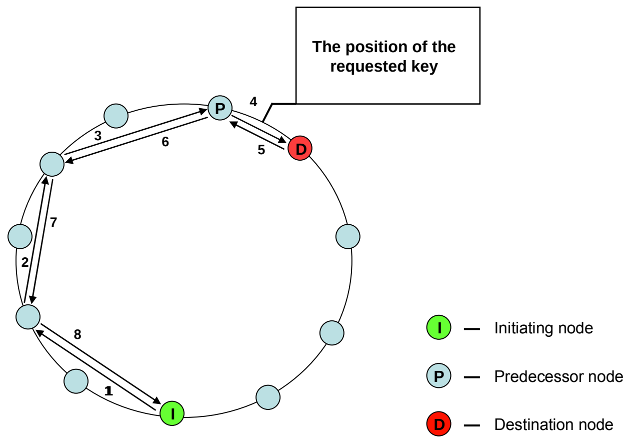


Figure 5 – Symmetric recursive routing

Symmetric recursive

In symmetric recursive routing (Figure 5) the response message visits the same nodes as the request message did only in a reversed order. This requires that the nodes remember from which node they received the request message. A via list storing information about the visited nodes can be included in the query message. This allows the response message to follow the same path in reverse. [Jen+08]

Forward-only

If the forward-only recursive routing is used, the response message is routed as a new message initiated by the recipient of the query. The route of the response message is thereby independent from the route of the query message. [Jen+08]

Direct response

The response message can also be sent directly to the initiating node. Direct response recursive routing illustrated in Figure 6 takes this approach. The initiator needs to encode its contact address in the query message so that the recipient node knows where to send the response. This is the optimal routing technique but it can rarely be used due to connectivity issues like NATs. [Jen+08]

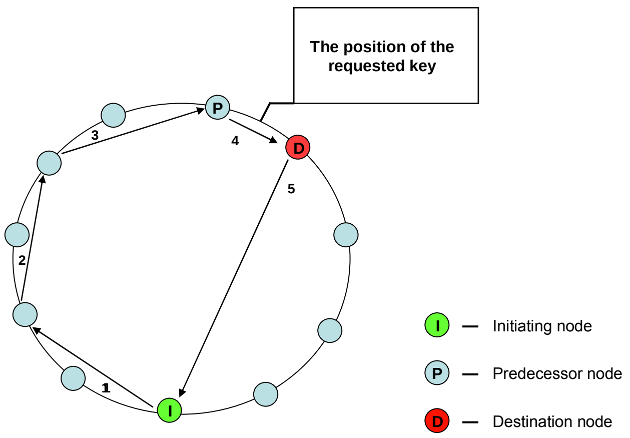


Figure 6 - Direct response recursive routing

Pros and cons of different routing solutions

Lookups that are routed recursively are proved to take only 60% of the time compared to those that are routed iteratively [Dab+04]. This holds at least in an error-free network environment. Recursive lookups are on the other hand more prone to churn (see chapter 4.2) than iterative ones. This derives from the fact that in the recursive routing mode the initiating node cannot monitor the query as accurately as it can in the iterative mode. In the iterative mode the initiator can monitor each routing hop separately, while in the recursive mode the initiator receives information only from the last hop node (considering there are no failures). [Kun05]

The lookup requests are monitored with timers by the initiating nodes. A timer is launched every time a lookup is sent. When there is no response for a lookup, a new lookup will be sent after the timeout (i.e. after the timer has expired). The timeout values differ significantly in recursive and iterative routing. As defined in [Wu+06] we denote the length (number of nodes) of the routing path by l, the average one-hop routing latency by Δ and the average round-trip time by $RTT = 2\Delta$. In recursive routing the timeout value T_l should be at least the time it takes to complete the entire lookup, $T_l \geq (l + 1)\,\Delta$. In iterative routing the timeout value T_h should only be larger than the round-trip time for one hop, $T_h \geq \mathrm{RTT} = 2\Delta$. Even though the number of hops in the distributed hash tables of our study (see chapter 2.3) is O(logN), we can see that as the number of nodes in the network increases, the time it takes for the initiating node to react to a failure, discovered by the absence of the response, is much longer when recursive routing is used.

In a network where nodes frequently join and leave the network, there is a probability of routing a message to a node that has already left the network. It is obvious that with recursive routing churn will cause more problems than in the iterative mode. In iterative routing the initiating node does not only detect the failure quicker but it can also react to the problem itself by continuing the lookup somewhere next to the failed node. In the recursive mode the initiating node has no way of knowing where the failure took place and it cannot inform the intermediate nodes to try a different route.

The performance of the two routing modes is measured in [Wu+06]. The study shows that in low churn rates and short routing paths ($l \leq 6$) recursive routing can outperform iterative routing. Under high churn and with longer routing paths iterative routing performs better.

2.3 Distributed Hash Tables

2.3.1 General

A distributed hash table (DHT) distributes the data and query load to several nodes in the network. The data is stored as (key, value) pairs and the node responsible for storing the data is determined by the DHT algorithm. DHTs make it possible to find an object from a network of thousands of hosts on the grounds of the object's key.

The abstract keyspace is divided among the participating nodes. Each node has an identifier, *node ID*, that defines its logical location in the network. The stored data is also identified by *a resource ID* generated by a hash function. The key for a data value can be hashed for example from the file name or from the objects keywords. In the case of SIP-communications the key is generated from the node's URI. The data is stored at a node whose identifier is "closest" to the key. DHT algorithms differ in how they define this distance.

The size of the keyspace a node is responsible for is determined by the number of nodes in the network. Although the keyspace is divided evenly between the participating nodes, there can be differences in the number of keys that nodes have to store. This is due to the fact that the hash algorithms are not always optimal, especially when there is churn affecting the network. DHTs aim to balance the responsibilities evenly on every node. There are various load balancing methods that address this problem [God+05].

DHT algorithms typically have four design constraints [Hel03]. The *Few neighbours* constraint means that each node keeps routing information for usually only log N other nodes in a network of N nodes. By distributing the routing information evenly on every node DHTs can handle the arrival and departure of nodes in a decent number of update messages. When the routing information is distributed, the node arrival or departure process affects only a small number of nodes and only those nodes need to update their routing table information.

The *Low latency* constraint indicates that all nodes should reach any other node in the network in a small number of hops. Usually this means that the maximum hop count between two nodes is log N. Nodes also need to be able to make their own routing decisions. This constraint called *greedy routing decisions* ensures that node lookups are efficient and every node can make its own routing decisions without the help of other

nodes. Network should also be able to withstand the effects of churn and retain connectivity and ability to route packets correctly as the nodes arrive and depart. DHTs should balance the load evenly so that there won't exist any overloaded nodes and links. These characteristics are demanded by *robustness* constraint.
[Hel03]

Various DHT algorithms have a different routing geometry for the keyspace. The keyspace can take the form of a hypercube, ring, tree or a butterfly. The routing geometry affects route and neighbor selection as well as the DHT's performance and resilience.

Consistent hashing
Most DHT algorithms use a technique called consistent hashing that is designed to minimize the need for changes in the key ownerships between nodes as the network is affected by churn (i.e. the nodes arriving and departing the overlay). In traditional hash functions a change in the number of key storing nodes would mean that the hash value for a given object would change. This would lead into a situation where lookups would be directed towards wrong nodes as an object could hash into several hashed values depending on the number of participating nodes at a given time.

DHTs used in this study
In this thesis we focus on two DHT algorithms Chord [Sto+01] and Kademlia [May+02]. Chord has a ring geometry where the nodes lie on a one-dimensional cyclic identifier space. The distance from an identifier A to B is calculated as the clockwise numeric distance on the circle. Kademlia measures the distance between two nodes as the numeric value of the exclusive OR (XOR) of their identifiers.

Different routing geometries have been compared against their static resilience in [Gum+03]. Static resilience measures the DHT's ability to route lookups correctly in the situation where there are failed nodes in the network and the recovery mechanisms have not yet taken any actions. The paper suggests that Chord's ring geometry has twice the amount of alternate routing paths than Kademlia's XOR geometry and because of that it has superior static resilience compared to Kademlia.

Li, et al. compare DHT designs by their performance and communication cost under churn [Li+05]. The paper states that Chord can ensure correct lookups better than other DHT algorithms when there is very little bandwidth available. This is due to the separation of successor table and the routing (finger) table in Chord. Chord only needs to update the successor lists to route lookups correctly while other DHT algorithms need to update all their routing information. When there is more bandwidth available Kademlia seems to defeat Chord in both efficiency and scalability as stated in [Kov07]. The paper has some lack of information about the Chord parameters so the results have to be interpreted carefully.

2.3.2 Chord

Chord is a distributed lookup protocol that maps a given key onto a node. This Section is based on paper [Sto+01] that first introduced the algorithm. Chord uses consistent hashing to ensure that all nodes receive roughly the same number of keys. The consistent hash function assigns each node and value to an *m*-bit identifier. The identifier length *m* must be large enough so that at a very low probability two nodes or values will hash to the same identifier. In this study 160 bit node identifiers are used.

In Chord the nodes are logically arranged in a ring called the identifier circle. The node's place on the circle is defined by the identifier that is hashed from the node's IP-address (in P2PSIP/RELOAD from the user's public key). Key identifiers are hashed from the values that the key identifiers are pointing to. A key *k* is assigned to *k*'s successor, the first node whose identifier is equal to *k* or follows *k* in the identifier space. Figure 7 illustrates how the responsibility of the keys is handled in Chord.

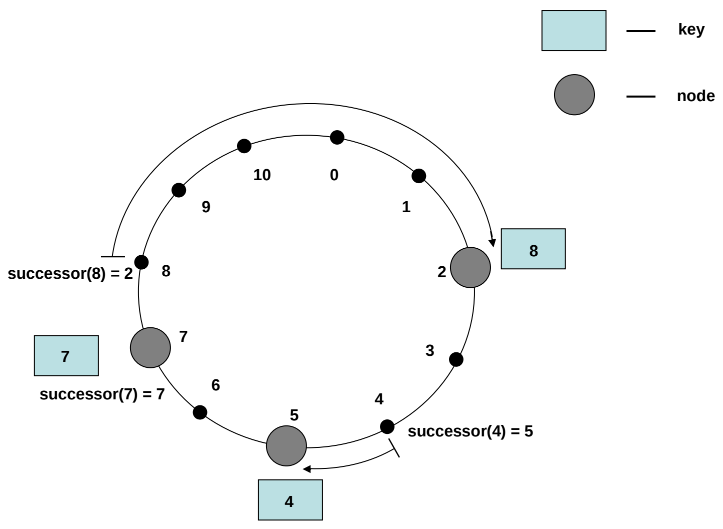


Figure 7 – Chord identifier ring with three nodes and three keys

When nodes join and leave the network, the responsibilities over the keys need to change from one node to another in order to maintain the consistent hashing. When node n joins the network, it gains the responsibility over some keys previously assigned to node *n*'s successor. When node *n* leaves the network, all of its keys will be reassigned to its successor. Looking at Figure 7, if a new node joins the network with identifier 10, the key with the identifier 8 will be reassigned from node 2 to this new node. If node 5 leaves the network, the key with identifier 4 will be reassigned to node 7.

The application running over Chord interacts with Chord in two ways. Chord provides a lookup(key) algorithm that returns the address of the node responsible for the key that the application looked for. Chord also informs the application about any changes in the set of keys that a given node is responsible for.

Routing tables

Chord keeps information about other nodes in two independent tables. Every Chord node keeps a list of the nodes succeeding it on the identifier circle. This *successor list* contains *r* nearest successors of the node. Correct lookups are achieved as long as the successor information is maintained correctly even if each node is aware of only its closest successor node ($r = 1$). The lookup is done in a one way manner so in the worst case it is required to traverse all *N* nodes to complete the lookup.

To accelerate the routing process every Chord node also keeps a *finger table* where additional routing information is stored. Node *n* keeps a finger table, whose entries are nodes that lie at exponentially increasing distances on the identifier circle from node *n*. The i^{th} finger of node *n* is the first node to succeed identifier $n+2^{i-1}$ on the circle. This means that for every destination key node *n* always has some finger pointing at least halfway to the destination. When each iteration loop in the lookup process halves the distance to the target identifier, it follows that log *N* halvings will be enough to find the target node. The first finger of node *n* is also the successor of node *n*. Finger tables can have at most *m* entries. The maximum number of entries *m* is the number of bits in the key/node identifiers.

Updating and stabilizing the routing table

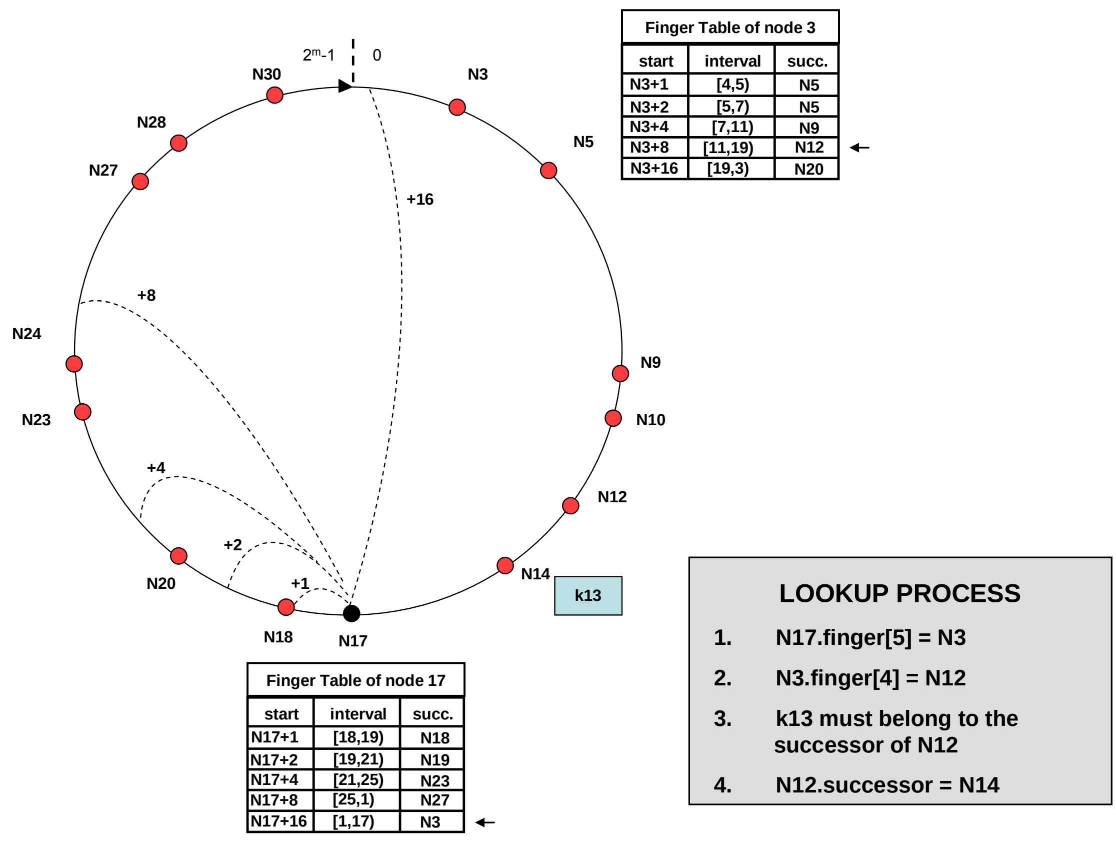
Nodes joining and leaving will cause the successor lists and the finger tables to get out of date. An update process for successor lists is needed to ensure the correctness of lookups. Finger tables also need to maintain the performance of the Chord algorithm, although finger table incorrectness won't prevent Chord from performing lookups correctly.

A stabilization protocol, *stabilize*, is run periodically at every node to keep the successor pointers up to date. The correct successor pointers are then used to update the finger tables. This protocol updating the finger pointers is called *fix fingers*. Intervals for running *stabilize* and *fix fingers* can be adjusted to give the best overall performance for a system with certain constraints. These constraints usually include lookup success rate and the overall bandwidth used by Chord.

Lookup process

In the beginning of a key lookup process the initiating node looks if it has the node responsible for the requested key in its successor list. If there is no match in the node's successor list, it uses its finger table to find a finger[*i*] which most immediately precedes the requested key. The initiating node then contacts this node and the contacted node repeats the same lookup procedure. The lookup process continues until the node that immediately precedes the desired identifier is found. This node then returns its successor as the lookup value. The lookup process is illustrated in Figure 8**.** Depending on the routing style the contacted node either gives the initiating node the address of the next node to contact (iterative style) or it contacts the next node itself (recursive style). These two routing modes have different constraints about the connections between the network nodes. Which style is preferred, depends on the type of network Chord is used in. Section 2.2 presents the comparison between these two routing styles.

Figure 8 shows an illustration of node with ID 17 performing a lookup for a key with ID 13. Node 17 looks for the closest node to ID 13 from its finger table and notices that 13 belongs to 5th finger interval [1, 17). The corresponding entry in its finger table is node 3 which is the first node in this interval. Node 17 will then ask node 3 to find the successor of ID 13. Node 3 then checks its finger table and finds out that ID 13 belongs to the 4th interval [11, 19) where the first node 12. Finally node 12 knows that ID 13 must belong to its successor node. Node 12 now replies that the node responsible for ID 13 is node 14.



Finger Table of node 3

start	interval	succ.
N3+1	[4,5)	N5
N3+2	[5,7)	N5
N3+4	[7,11)	N9
N3+8	[11,19)	N12 ←
N3+16	[19,3)	N20

Finger Table of node 17

start	interval	succ.
N17+1	[18,19)	N18
N17+2	[19,21)	N19
N17+4	[21,25)	N23
N17+8	[25,1)	N27
N17+16	[1,17)	N3 ←

LOOKUP PROCESS

1. N17.finger[5] = N3
2. N3.finger[4] = N12
3. k13 must belong to the successor of N12
4. N12.successor = N14

Figure 8 - Chord's lookup process. Node 17 looking for key 13

Performance

Chord does not waste nodes' capacity but uses it effectively. For efficient routing each node needs to maintain information for only O(log *N*) other nodes. The lookups are also resolved in an efficient manner. Each node can carry out a lookup to any given node via O(log *N*) messages to other nodes [Sto+01].

2.3.3 Kademlia

Kademlia is a DHT algorithm that has already been used in various peer-to-peer applications. It has been utilized by a number of very popular applications like eMule and BitTorrent. Kadelmia takes advantage of every message it sends by using them to keep the routing tables up to date. This way the need for explicit update messages is minimized.

Node identifiers and keys have 160 bits. Kademlia uses exclusive or (XOR)-metric to define logical distances between two nodes in the network. The distance between identifiers *a* and *b* is an integer interpretation of their bitwise XOR. Nodes are considered as leaves of a binary tree. The XOR topology is symmetric unlike the ring topology used in Chord. The symmetry feature means that the distance from node *x* to node *y* equals the distance from *y* to *x*. Figure 9 depicts the XOR metric.

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

$$\begin{array}{rr} & 00110101 \\ \textbf{XOR} & 11100011 \\ \hline & 11010110 \end{array}$$

Figure 9 – XOR-metric

Nodes are treated as leaves of a binary tree where each node gets its location according to the shortest unique prefix of the node's identifier. Every node sees the network as a group of subtrees that the node itself doesn't belong to. The highest subtree consists of the other half of the binary tree not containing the node itself. The next subtree includes the half of the remaining binary tree that is not part of the highest subtree. The subtree model is illustrated in Figure 10.

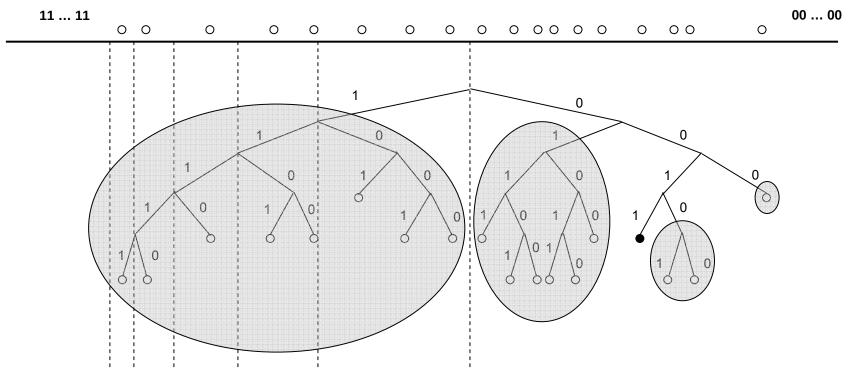


Figure 10 – Subtree model used in Kademlia

K-bucket concept

Every node stores contact information about other nodes in lists called k-buckets. Each node keeps a k-bucket for nodes that lie from a distance between 2^i and 2^{i+1} from itself. As $0 \leq i < 160$ this results in 160 k-buckets for every node. K-bucket entries are sorted by the time that has passed since the entries have been seen. The least recently seen node is the first node on the list and the most recently seen node is the last. Parameter *k* stands for the size of k-buckets. Each k-bucket can store at most *k* entries.

Routing table

Kademlia nodes use their k-buckets to form routing tables. Entries in each k-bucket have a common prefix of their IDs. The k-buckets are arranged in a binary tree and their positions are determined by the prefix. This binary tree with k-buckets as its leaves is the node's routing table. The routing table covers the whole ID range. As all k-buckets have equal size each node knows more about the ID ranges near its own ID than about those that are further away.

Routing table updating

Kademlia minimizes the need of update messages by including pieces of information about the overlay in every lookup message sent. As a node receives any message it updates the appropriate k-bucket. If the k-bucket already contains the sender's node ID, the sending node is moved at the end of the list. If the sending node doesn't exist in the appropriate k-bucket and the k-bucket is not full, the sending node is inserted at the tail of the list.

In the case that the k-bucket is already full the receiving node pings the least recently seen node in the k-bucket to know whether that node is still alive. If the ping is responded, the least recently seen node is moved at the end of the list and the new node is not added to the k-bucket. To keep the k-buckets stable and resilient against lost messages Kademlia doesn't evict a node from a k-bucket until it has failed to respond to five consecutive messages. An optimization has also been made to the process of pinging of the least recently seen node depicted above. The original algorithm results in a large number of ping messages and the optimized algorithm delays contacting the least recently seen node until there is a useful message it can send to this node.

Kademlia still keeps a timer for updating the k-buckets. If there are no lookups for a particular ID range the corresponding bucket could end up out of date. To prevent this from happening, each node performs a refresh procedure to every bucket that hasn't been a target of a lookup in the past hour. This is done by performing a node search for a random ID in the k-bucket's ID range.

Lookup process

Kademlia's has a node lookup procedure in which parameter α determines the number of parallel requests sent. The initiator picks α closest nodes to the requested node ID from its k-buckets and sends the lookup requests to these nodes. The recipients then return *k* closest nodes to the requested ID they know of (a single k-bucket if the bucket is full).

As the initiator gets the responses it picks the next α nodes that will be the new recipients of a request. This process goes on until the initiator has queried all the *k* closest nodes it has learned during the lookup process and also gotten a response from all of them. The lookup process is illustrated in Figure 11.

Performance

Each Kademlia node keeps information about $\mathrm{B}\,log_{\mathrm{B}}N + \mathrm{B}$ other nodes in its routing table. Here the letter B stands for the number of different node IDs in the network. Kademlia uses 160 bit node IDs which gives B a value of 2^{160} [May+02]. Node lookups are performed in $\mathrm{O}(log_{\mathrm{B}}N) + c$ hops where c indicates a small constant.

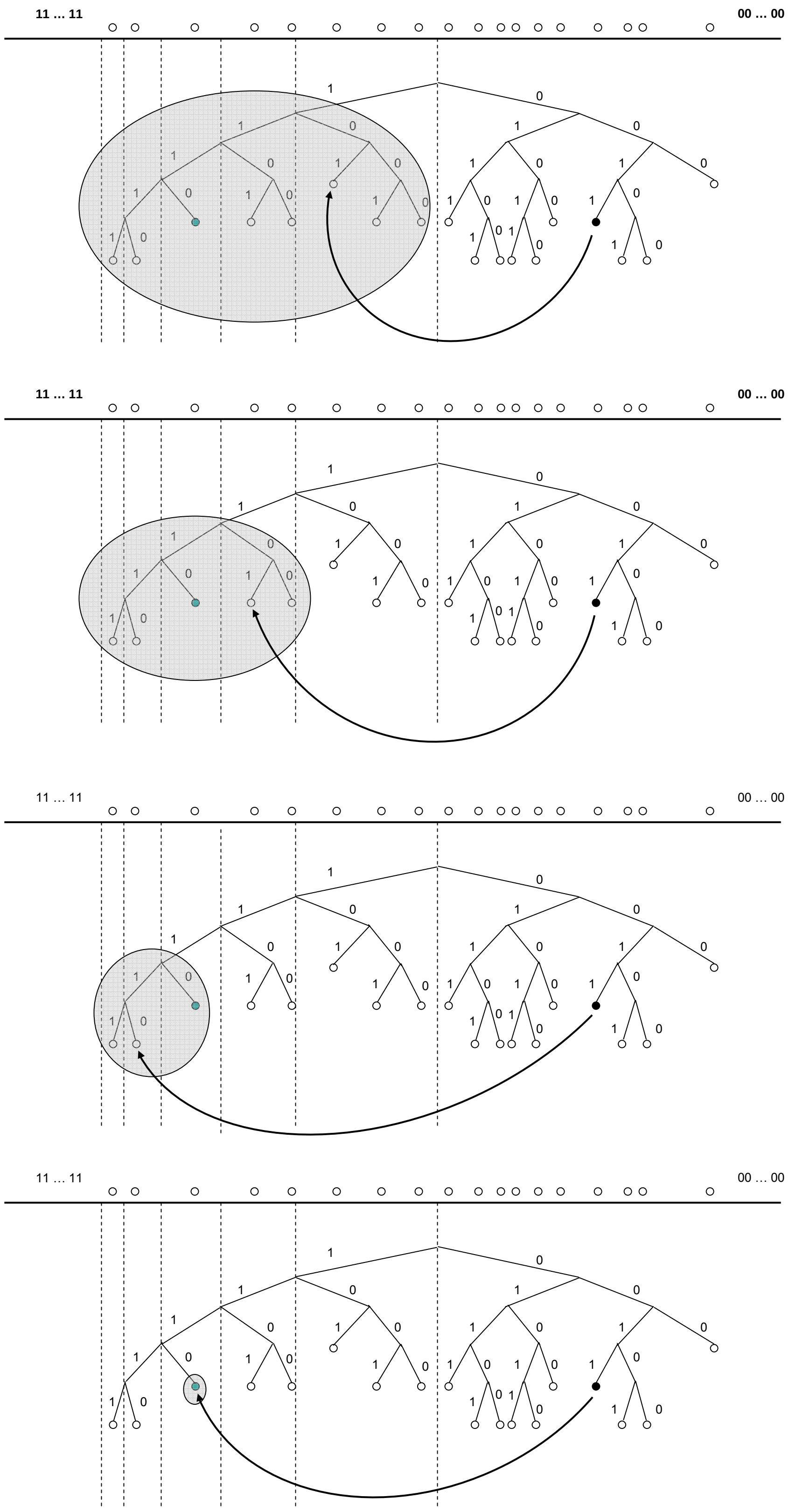


Figure 11 – Kademlia's lookup process

3 SIP and P2PSIP

This Chapter discusses two signaling protocols for session establishment. SIP, a widely used client-server protocol for controlling multimedia communication sessions, and P2PSIP, a novel protocol implementing the SIP functions without server elements.

3.1 SIP

SIP is a signalling protocol that is used to set up, modify and tear down multimedia sessions such as voice and video calls between one or more participants. SIP is standardized by the IETF and the protocol is used in 3GPP signaling and is a permanent element of the IP multimedia subsystem (IMS) architecture.

SIP delivers session descriptions to users and enables them to create a multimedia session between each other. SIP supports name mapping and redirection services which together enable mobility of users. SIP can locate the desired user, determine its willingness to participate in a session and get information of the user's support for given methods or protocols. After these phases SIP can determine the session parameters and establish the session. Established sessions can also be modified with SIP by inviting more participants to them or adding media, video for example, to an ongoing voice call session.

SIP supports user mobility and location with a user specific externally visible identifier called public Uniform Resource Identifier (URI). SIP is an application layer protocol very similar to Hypertext Transfer Protocol (HTTP). Like HTTP, SIP is text-based, uses a client-server architecture and a request/response transaction model where a transaction consists of a request and the responses triggered by the request.

The Session Description Protocol (SDP) [RFC2327] is used for describing the session parameters like the type of media, codec and sampling rate. The session parameters are included in a SDP message that is contained in the body of a SIP message. [RFC3261]

3.1.1 SIP ENTITIES

User agent (UA)
User Agents are SIP endpoints that establish and manage SIP sessions between each other. UAs send and receive the session descriptions and finally agree on the session parameters. UAs usually interact with a human user but they can also operate without user intervention like in the case of SIP voice mail service.

UAs are implemented on top of many different kinds of devices and systems including desktop SIP phones, software running on a computer or on mobile devices like laptops PDAs or mobile phones. The user interface can be very different between the various devices but the base functionality of the UA itself is always the same.

User agents divide into two logical entities: user agent clients (UAC) and user agent servers (UAS). UAC creates SIP requests and UAS creates responses to SIP requests. UAs take these roles only for the duration of that prevailing transaction.

Proxy (Proxy server)
A proxy server, or proxy for short, is a SIP router that routes the SIP messages toward their destination. Proxies receive SIP messages from UAs or other proxies. Proxies can be used to enforce different policies as they can interpret and rewrite request messages before they are forwarded. Proxies route the SIP messages in a recursive manner.

There are two operating modes for proxies. In stateless mode proxy acts as a simple forwarding element, which does not maintain any information about the messages it has forwarded. As a result, stateless proxies cannot retransmit any messages on their own. Proxies operating in stateful mode remember information about each transaction they have made. Stateful proxies can also “fork” incoming requests by routing them to more than one destination. Keeping state of transactions enables stateful proxies to perform certain operations that stateless proxies are not capable of.

Redirect server
Redirect servers also act as SIP routers but not in the same manner that proxies do. Redirect servers route the SIP messages in an iterative manner. Redirect servers inform the UA sending the message to try an alternative URI or multiple URIs. Redirect servers are not active in sending messages, they just answer to UAs’ requests.

Registrar

SIP users register their current SIP URIs by sending register requests to the registrar at their public URI´s domain. Registrar is a server that handles requests that are addressed to the domain it operates in. Registrar is responsible for updating the location service lists that include mappings between the user's public URI and SIP URIs. Usually the lists are kept in a physically separate location server. The current SIP URI is registered into the registrar and incoming requests can always be forwarded to the correct destination.

Location server

Location servers are not SIP entities though they play an important role in providing SIP's location service function (see chapter 3.1.3). Location information is stored on the location server by the SIP registrars and queried by the SIP proxies or UAs. SIP is not used as a communication protocol between the SIP entities and the location server. [Cam02]

Logical roles

SIP entities rarely all exist in physically separate elements. One physical element takes different logical roles on separate transactions. For example the proxy, the redirect server and the location server can all be physically located in a single device.

3.1.2 SIP Messages

SIP messages can be dived in two categories. Requests are upstream messages from a client to a server. Responses are downstream messages from a server to a client, which are sent upon reception of a request.

Requests

There are six requests defined in the SIP core specification [RFC 3261]. Those are listed in the list below. Extensions made to the core SIP specification include also other requests.

- REGISTER

Users register their contact information by sending a REGISTER request to the registrar of their domain.

- INVITE

The session setup is initiated by the INVITE request from the calling party. The message contains the session description and invites the callee to participate in a session.

- ACK

Final responses to INVITE requests are acknowledged by ACK requests. The client that sent the INVITE request is also the sender of the ACK request.

- CANCEL

Pending session set up can be revoked by the CANCEL request. This is possible as long as the server side has not yet returned the final response. If the final response has already been sent, the CANCEL request will be ignored.

- BYE

If one of the participants wants to abandon a session, a BYE request is used. If a session covers only two participants the BYE request automatically terminates the session. In a multi-party session the sender of the BYE request simply leaves the session.

- OPTIONS

SIP servers and UAs can be asked about their support for different methods and session description protocols by sending them an OPTIONS request.

Responses

SIP responses contain an integer code that holds information about the status of the transaction. Status codes from 100 to 199 indicate a provisional response and codes from 200 to 699 are used in final responses. SIP responses also carry a text-based reason phrase that represents the information of the status code in a way that is more informative to the end user. SIP entities ignore these phrases and act only according to the numerical status codes. Figure 13 illustrates some SIP requests and the responses triggered by them.

3.1.3 Location service function

The function of mapping a single public URI to one or more contact addresses (SIP URIs), where the holder of the public URI can be reached, is called *Location Service*. This abstract service provides address bindings for the domain the public URI belongs to. [RFC3261]

SIP users can be reached at any given moment with the same identifier regardless of their whereabouts. User's public URI is an email-like address which consists of a username and a domain name. A public URI can for example look like this:
`sip:john.smith@domain.com`

Other SIP users can always reach John via this address. John, for one, can have multiple SIP URIs depending on his current location. When John is working his SIP URI is
`sip:jsmith@ws20.company.com`

and as John is connected with his laptop to a public WLAN at a cafe John has the SIP URI
`sip:johns@cafeteria.com`

It is apparent that SIP needs a way to map these different URIs to Johns public URI. To make this possible, John has to let the SIP protocol know his current address. This is done with the help of a registrar. To remain reachable via SIP, John needs to register his new location with the registrar responsible for his public URI at domain.com. As the registrar at domain.com is aware of Johns current SIP URI, it can forward any requests addressed to `sip:John.Smith@domain.com`.

The registration process is depicted in Figure 12**.** In order to stay reachable, John registers his SIP URI by sending a *register* request to the registrar in his public URI's domain. Johns new location is stored to the location service and mapped to his public URI. Alice then wants to establish a session with John and sends an *invite* to his public URI. The Proxy at domain.com receives the invite and queries Johns SIP URI (or URIs if there are many) from the location service. After this the proxy sends a new invite addressed to Johns current SIP URI. The roles of the registrar and the proxy are logical and they can both be physically located in a single device.

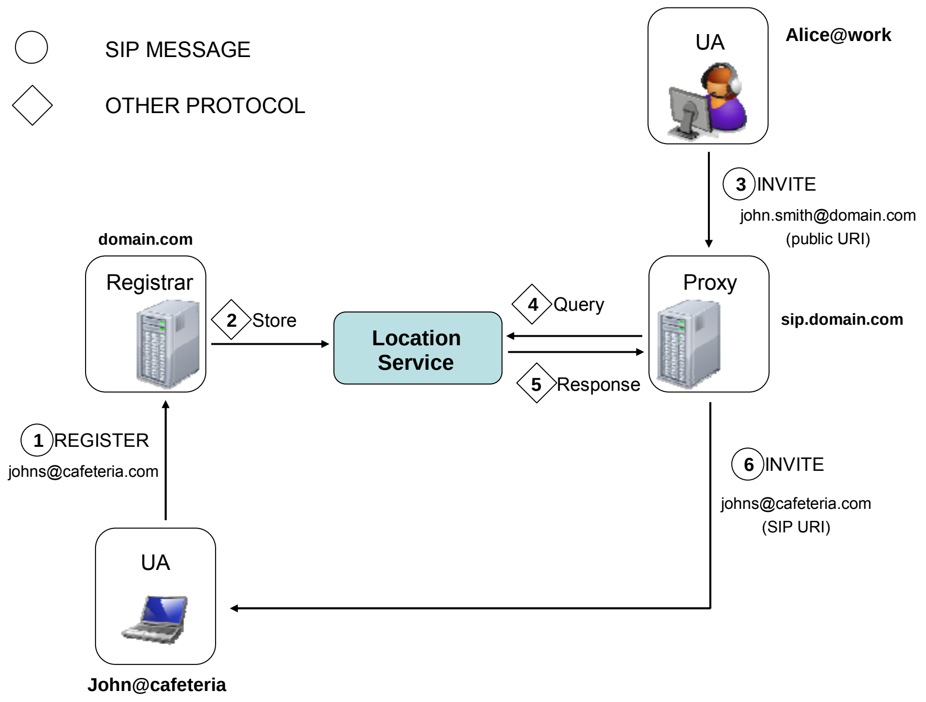


Figure 12 – Registration of a SIP URI and user localisation with the public URI

3.1.4 SIP session setup

Registered users can set up connections between each other. Figure 13 depicts an example session setup for two SIP user agents including two proxy servers. Alice (or Alice's UA to be exact) first sends an *invite* request addressed to John's public URI. The SIP proxy at Alice's domain (proxy A) receives the message and forwards it towards the SIP proxy of the domain that is responsible for John's public URI (proxy B). Then proxy A sends a *100 Trying* response to Alice, which indicates that it has received the request and taken the required action due to the request. Proxy B then forwards the *invite* request to John's UA. After this Proxy B sends a *100 Trying* response to Proxy A to inform it about receiving the *invite* request. Upon receipt of the *invite* request, John's UA replies with a *180 Ringing* response to alert Alice that the request has gone through. The response is then forwarded by the proxies to Alice's UA. Right after sending the 180 Ringing response, John's UA also sends a *200 OK* final response to Alice indicating that the *invite* request was successful. This response is also forwarded to Alice's UA by the proxies. Alice's UA then acknowledges this final response with an *ACK* request. The ACK request is delivered straight to John's UA without the proxies.

At this stage the media is established between Alice and John. In Figure 13 the ending of the session is initiated by John. After John has hung up, his UA sends a *BYE* request to Alice's UA, which then replies with a *200 OK* final response.

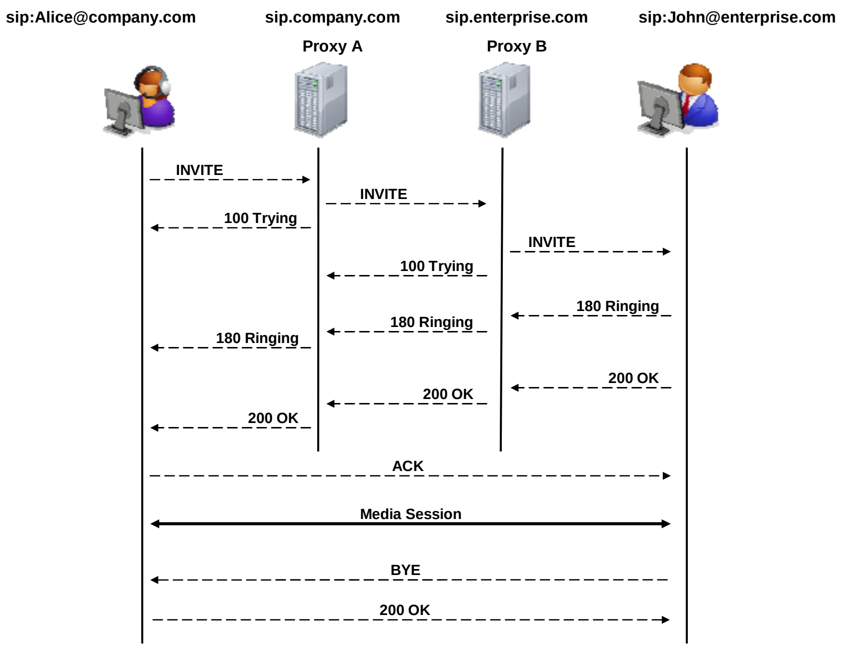


Figure 13 - SIP Session setup and tear down.

3.2 P2PSIP

P2PSIP is a set of protocols and mechanisms providing an alternative solution to the session establishment that conventional client/server SIP offers. It replaces the somewhat fixed topology of SIP with a DHT-based structured peer-to-peer overlay network. The overlay nodes, called peers, collaborate and provide the same location service function that maps Addresses of Records (public URIs) to overlay locations (SIP URIs) as conventional SIP does. In P2PSIP this is done in a distributed manner, every peer taking responsibility over routing and location information storing. To be able to provide the distributed location service function, P2PSIP offers distributed database- and transport functions.

The P2PSIP Working Group was founded by the IETF in 2007 to develop standards for serverless use of SIP. P2PSIP is still largely under development. There are many internet drafts discussing potential solutions for different mechanisms of P2PSIP. The current state of the general P2PSIP framework is documented in a WG draft [Bry+08].

3.2.1 High Level Description

In P2PSIP the overlay nodes are organized in a peer-to-peer manner to make SIP based real time communication possible. The P2PSIP network consists of P2PSIP peers and P2PSIP clients. Peers participate in the P2PSIP overlay and they provide storage and transport services to other peers in that overlay. The role of a P2PSIP client is still under debate. One approach is that the client is not itself part of the overlay but interacts with the overlay through an associated peer. Clients can store information in the overlay and retrieve information from it, but they do not contribute any resources to the overlay, and thereby do not route messages or store information for other nodes. [Bry+08]

Peers offer services to other peers to enable the provision of larger functions by the overlay. The services offered by every P2PSIP node are storage and transport. These services are needed to provide the location function which is a core function of the client/server SIP. The overlay needs to be aware of the services its peers support, in order to know which functions can be provided by the overlay. To make this possible, some information about the supported services must be stored into the distributed database. [Bry+08]

Peer Protocol

A specific protocol is needed for P2PSIP to enable communication between the peers. The peer protocol routes messages within the overlay, maintains the overlay and stores and retrieves data in the overlay. It was first suggested that SIP should be used for this inter-peer communication [Bry+08]. The P2PSIP WG is now working towards a new protocol for this purpose. Several drafts for the peer protocol have been proposed. At the moment the P2PSIP WG has one peer protocol draft as a working group item. It is called REsource LOcation And Discovery (RELOAD). This protocol is described in a more detailed fashion in Section 3.2.3.

A peer protocol called P2PP (Peer-to-Peer Protocol) has already been used in existing P2PSIP implementations for both fixed and mobile platforms [Har+07]. P2PP, like other peer protocol candidates, has still many problems to be solved before reaching an RFC status.

Client Protocol

A protocol to allow communication between the clients and the peers is also needed. This protocol has a working title Client Protocol and it is agreed to be a logical subset of the peer protocol. This means that any operation supported by the client protocol is also supported by the peer protocol. [Bry+08]

3.2.2 Reference model

It is expected that most P2PSIP peers and clients will be coupled with SIP entities [Bry+08]. The P2PSIP Reference Model illustrates this assumption in Figure 14. The scenario presented in the figure is only an example of a possible P2PSIP overlay. Other compositions are also possible.

Peer names refer to the SIP entity the peer is coupled with. Proxy peer is coupled with a SIP proxy, Redir peer with SIP redirect server and UA peers with a SIP user agent. It is also possible that a peer is coupled with more than one SIP entity. A Gateway peer connects the overlay to other networks.

The nodes can connect to a P2PSIP overlay in several ways. User agent peers A and B are connected straight to the overlay while UA peer C is connected behind a NAT. The UA client connects with the overlay via UA peer D using the P2PSIP client protocol. The SIP user agent interacts with the overlay through the proxy peer using SIP to communicate with the proxy peer. SIP UA can also use the Redir peer as an adapter node to interact with the P2PSIP overlay. The Proxy peer and the Redir peer speak both SIP and the peer protocol used in the overlay (RELOAD in this thesis). The overlay can connect to other networks such as PSTN through the gateway peer that speaks the appropriate protocols.

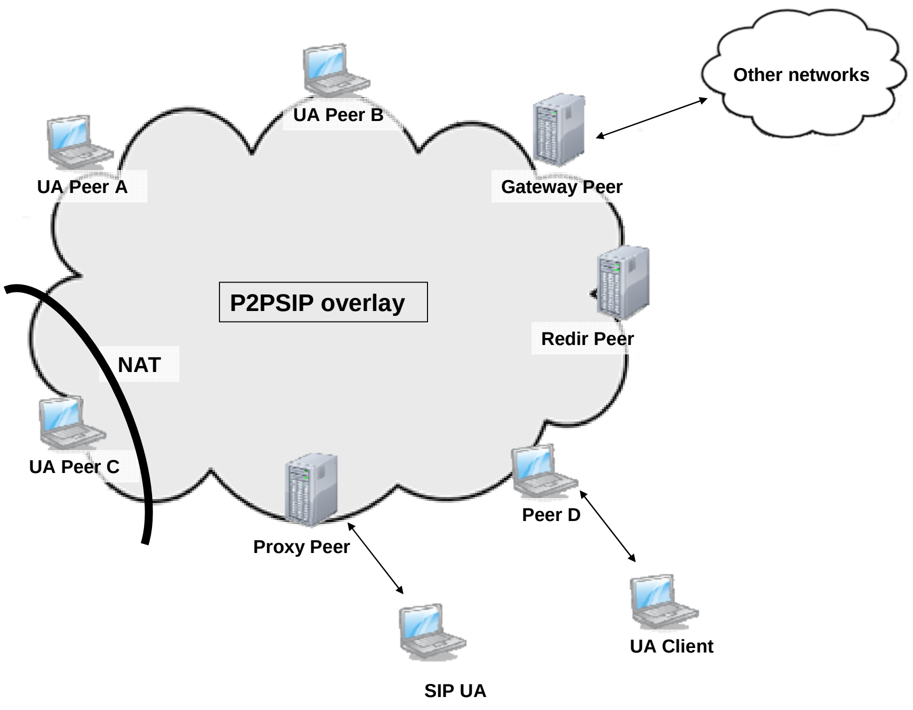


Figure 14 - P2PSIP Reference Model

NAT Traversal

Network address translation (NAT) is a technique in which the IP packet's address fields are translated as the packet traverses a router or a firewall performing the NAT function. The hosts that are located behind a NAT device have an IP address that is unique only in their own domain. If they want to communicate with a host outside their domain, they have to do it via a NAT that translates the source address of the IP packet to some globally unique address. Since the IP service model assumes globally unique addresses for every node, the presence of NATs causes problems for many protocols and applications. Several techniques have been developed to solve this problem. These techniques are referred to as NAT traversal. [Pet+03]

Interactive Connectivity Establishment (ICE) [Ros07] is a protocol that determines the best way to establish connectivity through NAT. ICE makes use of other protocols designed for NAT traversal and is therefore capable of avoiding the drawbacks of using only one NAT traversal method. [Leh05]

RELOAD is designed to support environments where nodes are behind NATs or firewalls. For NAT traversal RELOAD uses ICE to establish new RELOAD or application protocol connections. Tunneling is used for the application protocols when ICE is unable to establish a direct connection. [Jen+08]

3.2.3 RELOAD

This chapter is based on the current proposal of P2PSIP working group for peer protocol called Resource Location and Discovery (RELOAD) [Jen+08]. The details given here can still change but this Section aims to present an overview of the RELOAD protocol.

Applications make use of the RELOAD protocol by defining RELOAD *usages*. A usage specifies the supported data kinds and also the data structures and access control rules for those data kinds. Usages also specify the way the Resource Names are formed. The RELOAD draft defines two usages: SIP usage for the serverless implementation of the SIP protocol and diagnostics usage for monitoring the state of the overlay.

With the SIP usage, the RELOAD overlay can implement the user location function of SIP in a distributed manner. The SIP usage enables the RELOAD overlay to perform the registration and rendezvous functions, which in the traditional SIP network are carried out by the associated servers. With the registration function the SIP UA can store the mappings from its public URI to its current node ID as well as to retrieve other UAs' node IDs. The rendezvous function allows SIP UAs to use the RELOAD's message routing system to form direct connections to other SIP UAs.

Table 2 – RELOAD's Kind IDs [Jen+08]

Kind	Kind ID
SIP registration	1
TURN service	2
Certificate	3
Routing table size	4
Software version	5
Machine uptime	6
Application uptime	7
Memory footprint	8
Datasize stored	9
Instances stored	10
Messages sent/received	11
EWMA bytes sent	12
EWMA bytes received	13
Last contact	14
RTT	15

Data is stored in the overlay as elements called *data kinds*. One resource ID (i.e. peer) can contain several data kinds identified by a kind ID. The Internet Assigned Numbers Authority (IANA) has defined 15 kind IDs for RELOAD. The data kinds are listed in Table 2. The SIP usage defines the SIP registration (location), certificate, STUN server and TURN server data kinds.

Data can be saved in three different *data models*. The single value data model simply holds one DataValue element. There can be only one single value element in a kind ID per resource ID. The array data model is a set of array entries that include DataValue elements which are addressed by an integer index. An array can also hold empty indexes. The dictionary data model is a set of dictionary elements in which DataValue elements are indexed with a key for each value. One possible scenario for a resource ID's contents is illustrated in Figure 15.

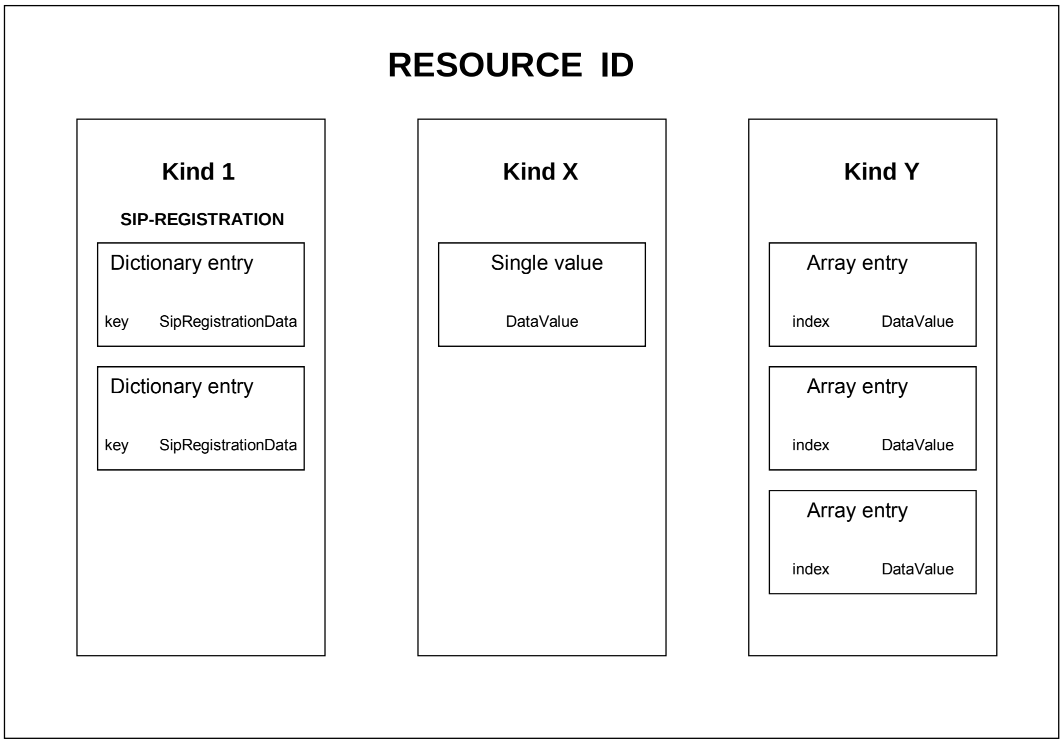


Figure 15 - Data storage model of P2PSIP

RELOAD messages

Reload uses messages for overlay maintenance and forming connections between the overlay nodes. Most messages have both request- and answer forms. The messages used for overlay topology maintenance are Join, Leave, Update and Route Query.

- JOIN

A Join request is sent by a new peer (joining peer) that wishes to join the overlay. The Join request's destination is the admitting peer and its purpose is to inform the admitting peer that a new peer wants to take responsibility over some part of the overlay. The admitting peer replies with a Join answer message.

- LEAVE

A node should send a leave message to every node it has a direct connection when it is about to leave the overlay.

- UPDATE

Update request is a message by which the sending node wants to notify the receiving node about the senders *routing state* (the sender's view of the current state of the overlay). Update answer is sent as a response and it indicates either success or an error.

- ROUTE QUERY

With the Route Query request message the sender can ask for the next hop peer for a message directed to a given destination. Route Query messages are mainly used for iterative routing (see Section 2.2.1).

The messages used for setting up and maintaining connections between the peers are Attach, Ping and Tunnel

- ATTACH

A Node can set up a connection to another node by sending the target node an Attach message through the overlay.

- PING

Ping messages are used for checking the connectivity between nodes and gathering information about the target peer's resources.

- TUNNEL

If a node has only a few messages to send for the destination node, applications using RELOAD can use Tunnel message to route the messages through the overlay instead of setting up a direct connection.

The messages associated with RELOAD's data storage protocol are Store, Fetch, Remove and Find.

- STORE

A Store request is sent by a node that wants to store data in the overlay. Several data kinds can be stored to a single resource ID with one store request message. Successful requests are responded with a Store answer message.

- FETCH

When a node wants to get one or more data elements from the overlay, it uses the Fetch request message. The request contains the resource ID where the data elements will be fetched from. Multiple data kinds can be retrieved with a single request. The requested data is included in the Fetch answer message, which is sent as a response to a successful request.

- REMOVE

To remove a stored data element from the overlay, a node sends a Remove request to the resource ID containing the given data. Only the creator of the data normally has the right to remove it from the overlay.

- FIND

Nodes can examine the overlay by sending Find requests. These requests contain several resource IDs and kind IDs that the requesting node wants to get information about. The Find answer message contains the closest resource ID for each kind ID that was included in the request message.

Overlay Operations

There are several operations that have to be performed when a peer wants to be a part of an existing overlay. This Section discusses those operations beginning with a peer willing to join an overlay and concluding in the situation where the peer is a fully functioning member of the overlay.

Enrollment

Before peers can function as members of an overlay they need to go through an enrollment process. Although P2PSIP aims for independence of centralized network entities, an enrollment server is usually needed in order to securely authenticate peers and admit them to join existing overlays. A peer wanting to join an existing overlay is called a joining peer. The joining peer gets its node ID and credentials during the enrollment process. The node ID is unique and it will identify the peer and its logical position in the overlay. After the enrollment process the joining peer can attempt to join the overlay by contacting a bootstrap node (see Peer joining and registration section below).

To find the enrollment server the joining peer initiates the discovery process. First the joining peer needs to know the name of the overlay it is about to join. P2PSIP does not determine how this information is acquired. The connection to the enrollment server's IP address is then established with HTTPS. The enrollment server replies with an overlay configuration document. The document includes information about the expiration time of this overlay configuration, overlay (DHT) algorithm being used, certificates, supported data kinds, addresses for credential server and bootstrapping etc. [Jen+08]

If the document the joining peer receives contains a credential server address, it means that credentials are required to be able to join the overlay. To get credentials the joining peer contacts the credential server with a 'Simple Enrollment Request' message over HTTPS. The credential server authenticates the joining peer using the user name and password included in the request. After a successful authentication the certificate is returned in a 'Simple Enrollment Response' message. It contains one or more node IDs which for security reasons must be unpredictable to the requesting (joining) peer. The response also contains the names this particular user is allowed to use when participating in this overlay. [Jen+08]

In the absence of the credential server the joining peer generates its own self-signed certificate. In this case the user can choose any username. The node ID must be computed from the user's public key using a hash algorithm (SHA-1 or SHA-256) defined in the configuration document. When self signed certificates are accepted by other nodes, the matching between the node ID and the public key of the peer in question is checked so that no peer can steal other peer's node ID. [Jen+08]

Enrollment is crucial for the security of the overlay. Malicious peers can try to disturb the overlay for example by frequently joining and leaving and thereby causing excessive churn which can overload the other peers. This is why the enrollment server must prevent one device from getting many IDs in a P2PSIP overlay. The IDs must be unique and the registered IDs cannot be modified. These features make it hard for malicious nodes to populate the overlay. [Son+08]

Peer joining and registration

After the joining peer has obtained a node ID and credentials it is ready to join an overlay. The joining peer first needs to locate a bootstrap peer for the overlay. There is a bootstrap mechanism in P2PSIP that helps the joining peer to locate a bootstrap peer, the first point of contact in the overlay the joining peer wants to participate. There are four different ways of locating a bootstrap peer defined in [Bry+08].

The joining peer can:

- Cache addresses of peers that participated in the overlay last time the peer was a member of the overlay
- Use a multicast discovery mechanism by sending a Ping request to the address given in the overlay configuration document
- Use manual configuration
- Contact a bootstrap server and ask for an address of a bootstrap peer

The first two options are included in the RELOAD draft [Jen+08] and they are suggested to be used in the order they are presented above.

After the address of a bootstrap peer has been resolved, the joining peer contacts the bootstrap peer that refers the joining peer to an admitting peer. The admitting peer is a peer already participating in the overlay, which will help the joining peer to become a fully functional peer. The choice of the admitting peer often depends on the peer ID of the joining peer. The admitting peer usually is a peer that will be a neighbor of the joining peer after it has become a member of the overlay. This comes from natural reasons, since the admitting peer's role is to help the joining peer learn about the other peers in the overlay. The peers that are logically near each other (i.e. have peer IDs that are close to each other) have very similar routing tables and it is wise to pick such an admitting peer that already has most of the information the joining peer will need.

The joining process is illustrated in Figures 16-19. The joining peer (JP) sends an attach message to the admitting peer (AP). The message goes via the bootstrap peer because the joining peer does not know the location of the admitting peer. The Bootstrap peer (BP) contacts the admitting peer through AP's predecessor (PP) and AP responds with an Attach_ans message which travels via BP to the joining peer. JP and AP then use ICE to connect and then set up a TSL connection. Figure 16 illustrates these actions.

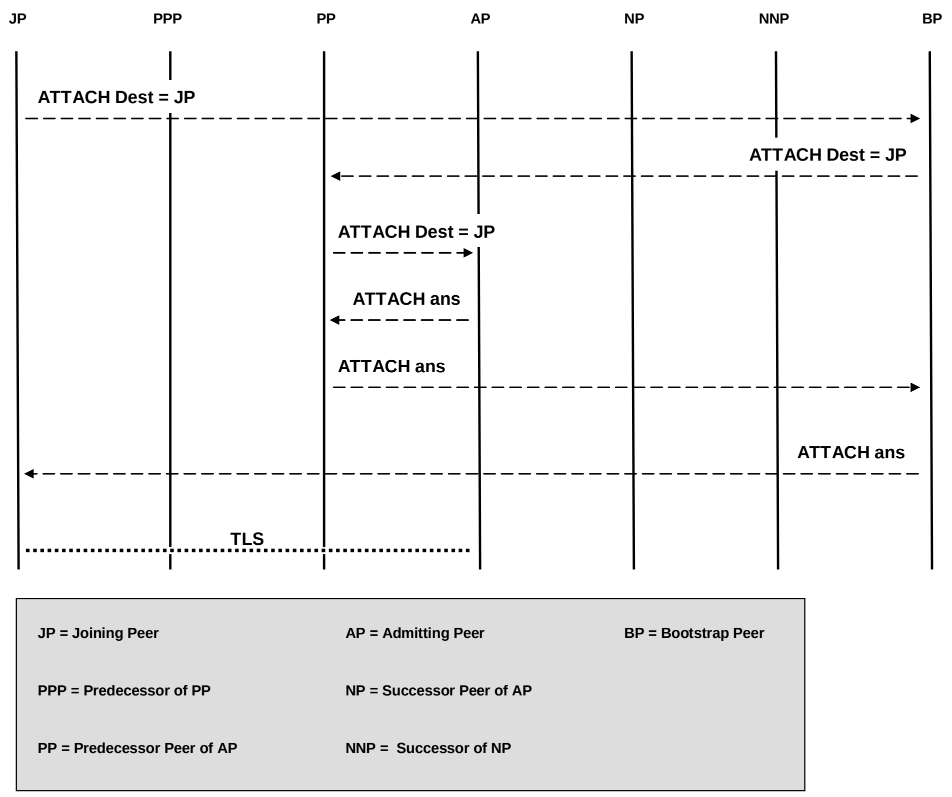


Figure 16 - Joining Peer discovering its Admitting Peer

After establishing a connection with AP, the joining peer needs to contact the peers it is supposed to have in its routing table. The figures here represent the case where Chord is used as the overlay algorithm. First JP populates its neighbor table by sending an Attach message to AP’s successor node NP. This is again done through AP and the destination address is set to AP+1. The final connection set up is again done with ICE and the connection will be TLS secured (Figure 17). This routine is repeated until every entry in the neighbor table is filled up.

When using Chord also the finger table entries need to be populated. This is done by sending Attach messages to locations exponentially further away from the previous finger table entry (the Chord’s finger table concept is described in Section 2.3.2). Figure 18 depicts JP contacting its last finger table entry halfway around the Chord ring. The Attach message is sent with a destination ID of JP+2^127. The message is routed to a node XX which denotes the predecessor node of the target peer.

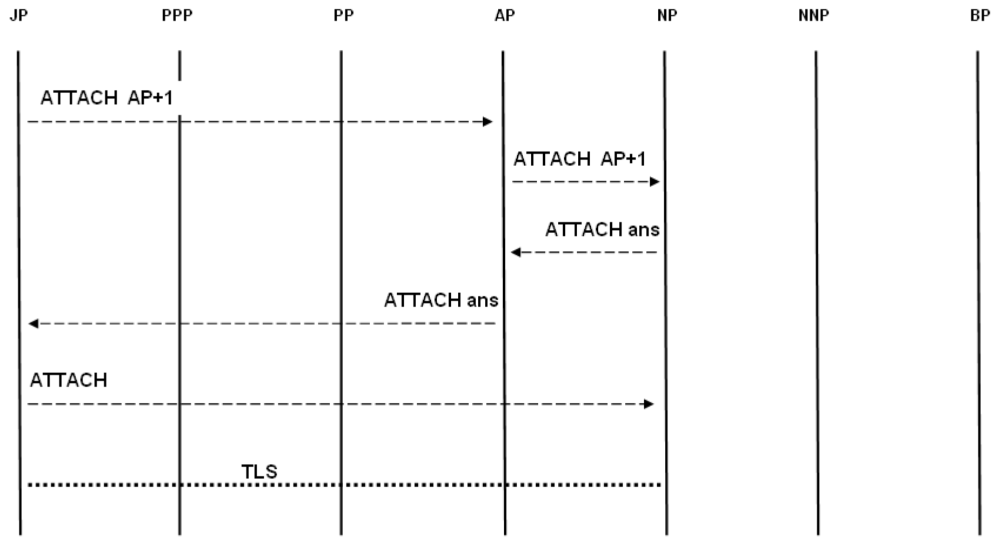


Figure 17 – Joining Peer populating its neighbor table

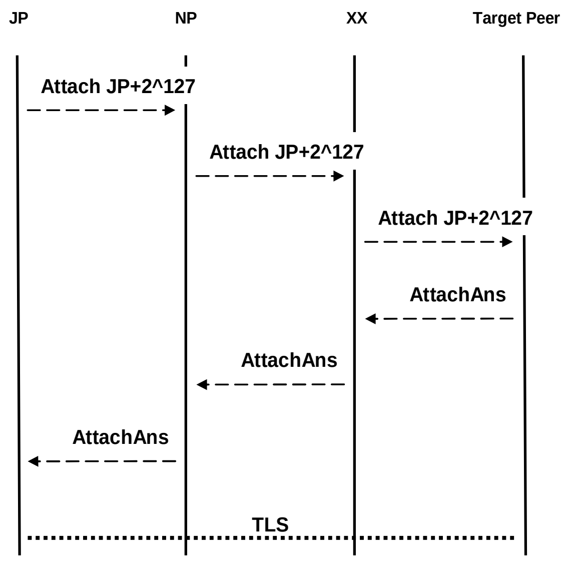


Figure 18 – Joining Peer populating its last finger table entry

When the routing tables are populated, JP is ready to take its own place and be a fully functioning member in the overlay. JP sends a Join request message to AP which triggers the process of JP getting the data it will be responsible for. After receiving the Join request message from JP, AP replies with a JoinAns message and then starts sending the data items (or pointers to the data items) to JP. After the data items have been transferred, AP sends an Update message to JP and indicates that JP is its predecessor. This process is illustrated in Figure 19. At this point JP knows the ID-space which it has responsibility for in the overlay.

JP has not yet been able to contact its predecessors because Chord has no way to route to them unless they are known. As JP received the Update message from AP the message contained information about AP's predecessors. These predecessors are also JP's predecessors so JP sends Attach messages to them and then sets up the connections with the appropriate predecessor peers. The final phase of joining the overlay consists of JP sending Update messages to every entry on its routing table to let them know that it is ready to operate as a fully functional peer.

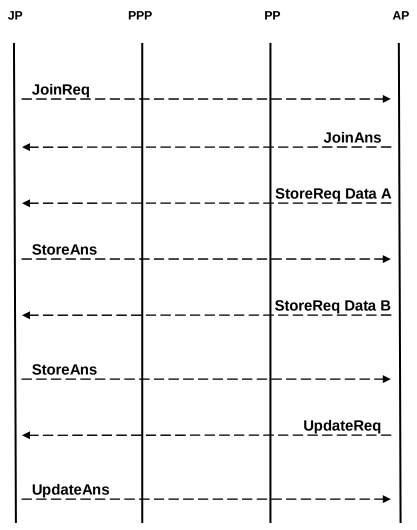


Figure 19 – Joining Peer receiving the data items it will be responsible for from the Admitting Peer

Resource registration

In order to implement the location service function, a peer has to register its location information in the overlay. The registration is done by storing a Sip Registration Data structure under the peer's own public URI.

SIP registration kind defines dictionary as its data model. Data is stored into dictionary entries as SIP Registration Data elements. User's public URI serves as the resource name, which is hashed in order to solve the resource ID that will be responsible for storing the data element. SIP Registration kind accepts two types of data;

- SIP Registration URI is an address where the user can be reached at.
- SIP Registration Route is a list of destinations (addresses) to the user's peer.

When using the former type, a user can inform anyone who tries to call him to try another URI instead. The latter type is used to direct callers to contact a specific peer where the user will be reached at. It is suggested in [Jen+08] that both the dictionary key and value are set to the peer ID to be contacted. Multiple registrations for a single public URI are supported in P2PSIP using the latter approach. The example dictionary entries for the two SIP Registration Data types are illustrated in Figure 20.

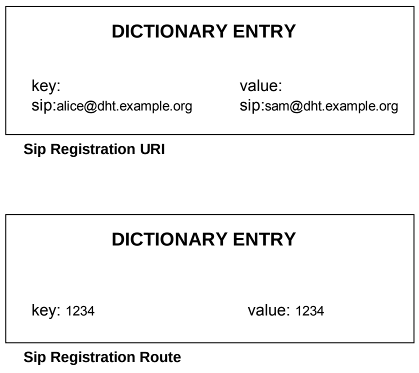


Figure 20 – SIP Registration Data types

Updating the overlay

UPDATE messages are used to keep the nodes' knowledge about the overlay up to date. The updating process and the contents of the messages included to it are completely overlay-specific i.e. they depend on the DHT algorithm used. Chord is the default DHT algorithm for RELOAD and it is mandatory to implement. Information about the update process of Chord can be found in Section 2.3.2.

Retrieving information from the overlay

FETCH and FIND messages can be used for information retrieval. FETCH message contains the resource ID, kind ID, data model and possibly a subset of the values included in that data model. FIND message is used to explore the overlay and its purpose is to determine what kind of resources can be fetched from the overlay.

Maintenance

Overlay maintenance in RELOAD is taken care of with Join, Update and Leave procedures. The message contents and when they are sent depend on the overlay algorithm that is used.

4 Simulation of P2P networks

This Chapter discusses different types of simulation tools and presents the simulation environment used in our study. The concept of churn is also introduced.

4.1 General

P2P networks, like any systems, can be simulated with a numerical, computer based simulation which imitates the behaviour of the network over time. The data generated by the simulation is then collected and analyzed and after the analysis, estimations about the performance of the true system are made.

As a consequence of the development in processing power, simulation has become a useful tool in testing and designing of large networks consisting of thousands of nodes. Building of large prototype systems is often impossible and always very expensive and time consuming. With simulation larger systems can be evaluated than when using prototypes. Simulation can be used as a tool to evaluate different design choices for systems that are not yet implemented as well as for predicting the behaviour of existing systems under varying circumstances. Simulation enables a simple way of evaluating the system's dependence on different input parameters. The arrival and departure of nodes as well as the lookup frequency can be modified to match any conditions. With prototype networks it is impossible to make these adjustments with equal accuracy.

Simulation software can be divided into general-purpose programming languages (C, C++, Java), simulation programming languages (SIMULA, GPSS, MODSIM) and simulation environments (NS2, OMNeT++). The general-purpose programming languages - obviously - are not specifically designed for simulation purposes. They are flexible to use and available in most computers but require a lot of programming work. The simulation languages offer many ready-made features for building a simulation model, and thus ease the workload of the person programming the model. Simulation environments enable building of the simulation scenario without actual programming. This way the implementation of the model can be done in a short period of time. Simulation environments usually have graphical user interfaces with animation and visualization tools. Also tools for analyzing the simulation output are included. [Ban+05] [Las07]

4.1.1 Discrete-event simulation

Discrete-event simulation, also referred as event driven simulation, is based on handling and scheduling events. An event is given a following definition in [Ban+05]: *"An instantaneous occurrence that changes the state of a system"*. An event in a P2P network simulation can be for example an arrival or departure of a peer or a message.

In discrete-event simulations the simulation time moves in discrete steps from one event to the next one. The time intervals that do not include any events are skipped over. The upcoming events are kept in a *future event list* which is processed in chronological order. As the events are handled, new events are generated in the process on the grounds of the dependencies and logic of the simulation model. The generated events are then put into the future event list to their proper position. After the first event in the list is handled it is removed from the list and the simulation clock will jump to the time instant the next event is scheduled to. [Ban+05]

Future events are scheduled as the present events are handled. For example in handling an event of a peer arrival to the network occurring at time instant *t*, a lifetime *l* for that peer is drawn in random from a given statistical distribution. Based on that lifetime value a new event is then scheduled for the departure of that peer to occur at time *t+l*. Another example from P2P network simulations is determining the intervals for the query messages sent by a peer. As the event of sending the first query message is handled, the next query message will be scheduled and put into the future event list. [Ban+05]

The components of a discrete-event simulation program are presented in [Las07]. They include

- Event scheduler

  The event scheduler maintains the future event list and it is always executed before an event. It can modify the event list and it can be called many times during the handling of one event.
- Simulation clock and time advance mechanism

  The simulation needs a global variable that represents the simulation time. In *time-driven simulations* the time is advanced in constant size increments and in *event-driven simulations* it is advanced based on the event beginning times event by event.

- The state variables of the system

  The state variables include the global variables, which together define the state of the entire system. The number of peers in the network at a given moment of time is an example of a state variable in P2P network simulation.
- Event handlers

  Event handlers process the events with event specific routines. The state values are updated and new events are scheduled by these routines
- Input routines

  The user of the simulation gives the program information about some important parameter values using the input routines.
- Report generator

  Report generator collects the data during the simulation, makes statistical analysis on it and outputs the results.
- Initialization routines

  The initial values of the state variables are set by the initialization routines. These values are usually acquired from the input routines.

4.2 Churn

The term *churn* is used to describe the process of nodes arriving to the overlay and leaving it. This process of node arrival and departure keeps the overlay in a continuous state of change. The changes in the overlay require certain maintenance operations from the overlay protocol in order to maintain the correctness of the lookups and the possible content retrievals.

There are two main characteristics that have to be set in order to determine churn rate. First there is the distribution of node arrivals, second the distribution of session time or peer uptime. Accurate churn characterization is vital in order to draw accurate conclusions about peer-to-peer systems. Modeling churn without detailed information about the arrival and departure of peers is a very challenging problem. [Stu+06]

In mobile environment churn comes in different forms than in traditional P2P applications where turning the application on and off is the only source of churn. In mobile handsets the battery may run out, the device may become unreachable because it has moved to a location where it does not have a connection to any network or it can only connect to a network in which it cannot get enough bandwidth in order to work properly.

4.3 OverSim

The simulations were carried out by OverSim [web1] overlay network simulation framework based on OMNet++ [web2] discrete event network simulator. OverSim consists of modules that are defined in a simple definition language NED. The modules are implemented in C++. Figure 21 illustrates OverSim's modular architecture. In this work the OverSim-20080919 release is used.

OverSim supports different underlying network models. In the release we use there are three models available: *Simple underlay*, *Single host underlay* and *IPv4 underlay*. In our study we use the Simple underlay which is used for large networks because of its scalability. In this underlay model packets are sent directly from one overlay node to another and have a constant delay.

OverSim has three different churn models to choose from: *Lifetime churn*, *Pareto churn* and *Random churn*. In our study Lifetime churn is used. OverSim implements DHT algorithms Chord and Kademlia that are both used in our simulations. OverSim also implements the desired routing modes and provides a generic lookup mechanism that can be used to test these different key based routing alternatives [Bau+07].

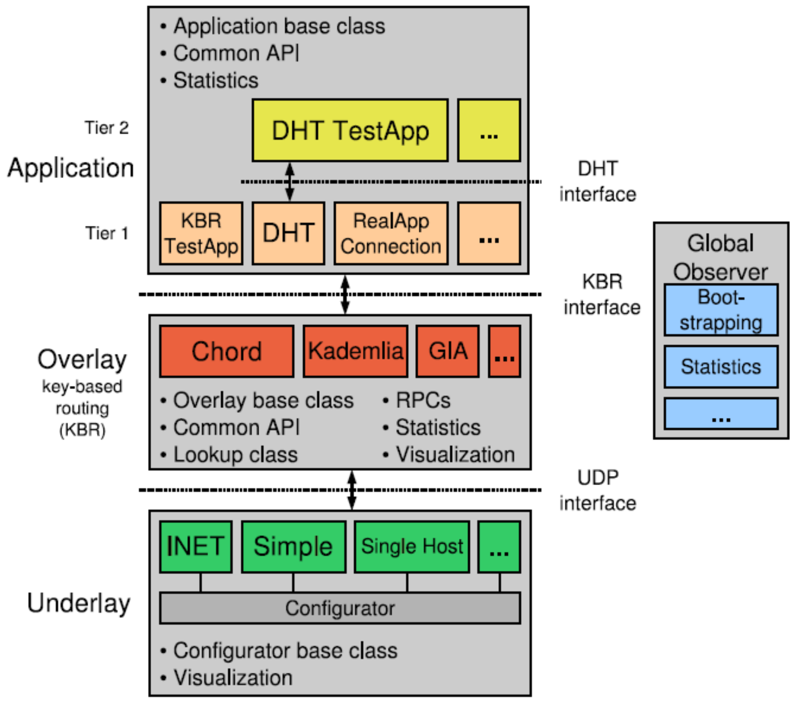


Figure 21 – OverSim architecture [Bau07]

OverSim has two configuration files that are used to specify all the relevant simulation parameters. A file called *default.ini* contains all those parameters and another file called *omnetpp.ini* contains simulation run specific parameter settings. The parameters in omnetpp.ini replace the values in default.ini if there is any overlapping between the two files. Omnetpp.ini is used for making different kinds of simulation scenario settings. These scenarios can then be run without separately configuring the default.ini file. Both configuration files are presented in Appendix E.

4.4 Distributions used in the simulations

Exponential Distribution

In this simulation the times between sending FETCH messages from a single node are drawn from the exponential distribution. The distribution is presented in Figure 22 with cumulative distribution function (cdf) in subfigure a and probability density function (pdf) in subfigure b.

$$f(t;\lambda) = \begin{cases} \lambda e^{-\lambda t}, & t \geq 0 \\ 0, & t < 0 \end{cases}$$

$$F(t;\lambda) = \begin{cases} 1 - e^{-\lambda t}, & t \geq 0 \\ 0, & t < 0 \end{cases}$$

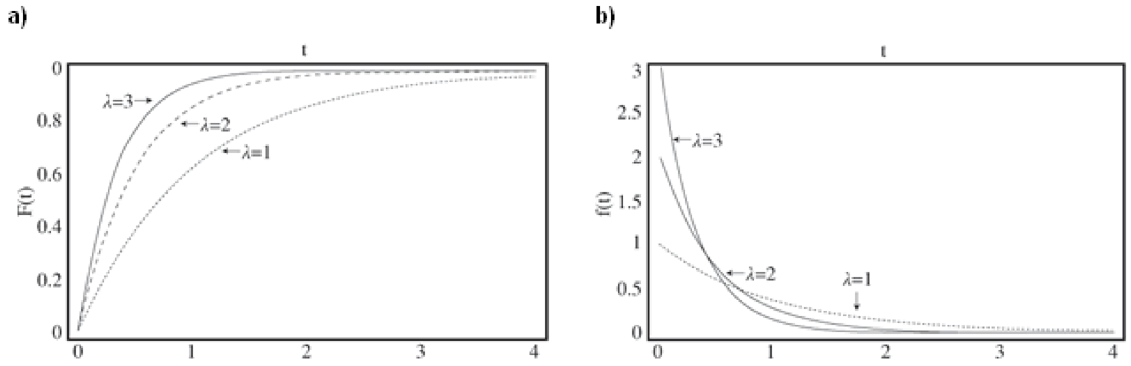


Figure 22 – cdf (a) and pdf (b) of the Exponential distribution [web3]

Weibull Distribution

Weibull distribution is an extension of the exponential distribution. It is illustrated in Figure 23. In our simulations the Weibull distribution is the same as Exponential distribution. This is the case as the shape parameter *m* has value of one. The similarity can be noticed by comparing the cdf plots.

$$f(t;\lambda,m) = \begin{cases} \frac{m}{\lambda}\left(\frac{t}{\lambda}\right)^{m-1} e^{-(t/\lambda)^m}, & t \geq 0 \\ 0, & t < 0 \end{cases}$$

$$F(t;\lambda,m) = \begin{cases} 1 - e^{-(t/\lambda)^m}, & t \geq 0 \\ 0, & t < 0 \end{cases}$$

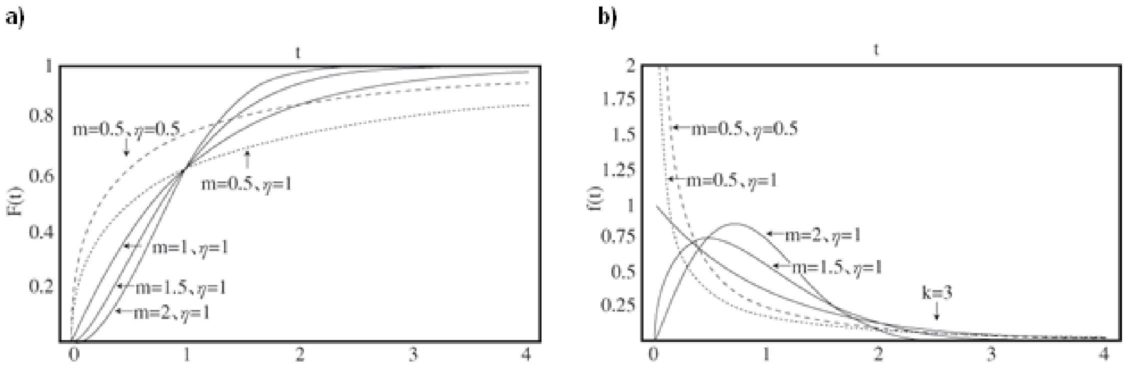


Figure 23 – cdf (a) and pdf (b) of the Weibull distribution [web3]

5 Simulation setup

This Chapter presents the preparations we made with the OverSim simulator to create the desired simulation scenarios. The preparations included modifications to the simulator code, modelling of REALOAD messages, parameter settings and statistics collection.

5.1 Simulation environment

The simulations are performed in a 32bit Ubuntu virtual machine run on a server with eight CPUs of 2,5 GHz and 4 GB RAM. Only one CPU is used for the simulations.

5.2 Modifications to the OverSim code

The standard record functions in OverSim were not enough to collect all the relevant statistics from the simulations. A new class, *MyClass* (Appendices B and C), has been written to collect these statistics and necessary additions to some other classes have been made in order to function with the new class. Another option would have been to add the statistic calculations to each relevant class. With our approach we wanted to make as few modifications to the OverSim code as possible. The most important added features are the measurement of hop count, which was already implemented in OverSim but had some problems in it and did not produce correct results. Also the calculation of the key distribution has been added as a new feature.

In order to make the lookup calculations correct the OverSim code has been examined and a new way of counting the hops has been developed and added to the code. This required minor additions to OverSim's Chord and Kademlia classes. The actual hop calculation code has been added to MyClass discussed above.

RELOAD protocol functions have been modeled with OverSim's DHTTestApp class which provides many features RELOAD uses. RELOAD operations are initiated by this class which communicates with the DHT class. The DHT class then communicates with

Chord and Kademlia classes. The relations between OverSim's classes relevant to this thesis are explained in Section 5.3.

Modeling the RELOAD protocol is not an easy task because of the somewhat general protocol description in the draft [Jen+08]. Assumptions about message sizes had to be made. The values not accurately determined in the draft were set in a way that they would be in the right scale so that the results obtained from the simulations would be credible. An example of these values is the determination of the message sizes.

Our simulations pay attention mostly on calculating the update traffic. This is why the update message sizes are especially important. The RELOAD message sizes are determined by the message sizes of the DHT algorithms (i.e. Chord as Kademlia does not really use update messages). With RELOAD those messages are complemented with a header field of 52 bytes and a signature field of 20 bytes.

Our intention was to also examine Kademlia DHT algorithm's recursive routing mode. Recursive routing was included in OverSim's implementation of Kademlia. The lookup success rates for Kademlia using recursive routing mode in our test runs were unexplainable low. It was obvious, based on those test runs, that there was something wrong with the routing mode implementation. Despite of our effort we did not manage to make Kademlia in the recursive routing mode to work correctly. Eventually a decision was made to simulate Kademlia only in the iterative routing mode.

5.3 OverSim Classes

DHTs are tested with OverSim's DHT test application which we have modified in order to equate the behavior of P2PSIPs peer protocol RELOAD.

OverSim classes modified for the simulations that are run in this study and the relations between them are presented in Figure 24. DHTTestApp and GlobalDhtTestMap classes are located in the Tier2 library that communicates with the Applications library containing the DHT and DHTDataStorage classes. In the Overlay library there are both Chord and Kademlia classes that define the logic that the DHT class executes. These two classes are subclasses of the BaseOverlay class located in the Common library. SimpleNetConfigurator class in the Underlay library defines the parameters for the network running below the overlay. Functions from MyClass are called in all of the classes presented in Figure 24 in order to collect essential information about the simulation runs.

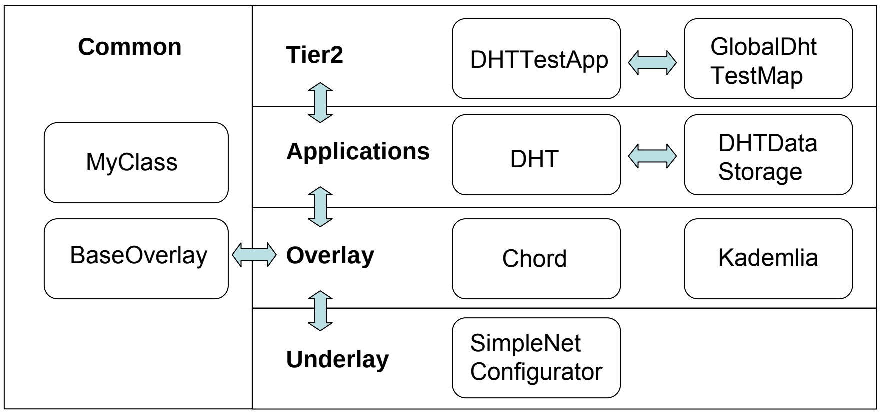


Figure 24 – OverSim libraries and modified classes

5.4 Modelling of messages

Chord's maintenance messages (FixFingers, Stabilize, Notify, New Successor Hint, Join) were modelled to correspond RELOAD's update message. Sizes for those messages are defined by adding a 52 byte header field and a 20 byte signature field to the regular OverSim Chord messages. The message sizes including the header and signature fields used in our study are presented in Table 3.

Table 3 – Maintenance message sizes

message	size (bytes)
Fix Fingers Call	105
Fix Fingers Response	158
Stabilize Call	104
Stabilize Response	130
Notify Call	105
Notify Response	340
New Successor Hint	126
Join Call	104
Join Response	339

OverSim has *FindNodeCall* and *FindNodeResponse* overlay messages, which are used for the node lookup process. FindNodeCall has a constant size of 52 bytes while FindNodeResponse contains variable amount of data depending on the DHT algorithm and the parameters the algorithm is run with. In our simulations FindNodeResponse for Chord has a size of 59 bytes and for Kademlia a size of 241 bytes. These are default sizes in OverSim and they are not modified in any way.

5.5 Fixed parameters

The relevant parameters for the simulations in this thesis are listed in Table 4. The parameters are from OverSim's *default.ini* file which is presented as a whole in Appendix E.

Table 4 – Parameters from default.ini

group	parameter	value	description
DHT	numReplica	1	number of replica for stored data records
	numGetRequests	1	number of queried replica for get requests
	ratioIdentical	0.5	ratio of identical replica needed for a valid result
DHTTestApp	testInterval	1309 1	mean interval for lookup messages from one node
	putDelay	5000	interval for key put messages
	initDelay	0.2	delay for storing the key
Chord	joinRetry	2	
	joinDelay	10	delay between join retries (s)
	stabilizeRetry	1	retries before a successor is considered failed
	stabilizeDelay	120	stabilize interval (s)
	fixfingersDelay	240	fix_fingers interval (s)
	successorListSize	8	max number of successors in the SuccessorList
	aggressiveJoinMode	true	
	extendedFingerTable	false	
	proximityRouting	false	
Kademlia	lookupRedundantNodes	8	number of next hops in each step
	ParallelPaths	1	number of parallel paths
	ParallelRpcs	1	number of nodes to ask in parallel
	lookupMerge	true	
	minSiblingTableRefreshInterval	1000	siblingTable refresh delay (s)
	minBucketRefreshInterval	200	bucket refresh delay (s)
	maxStaleCount	0	number of timeouts before node removal
	k	8	number of paths (size of k-bucket)
	s	8	network diameter O(log_{2^(b)})
	b	1	number of siblings
	pingNewSiblings	true	ping new unknown siblings?
	activePing	false	ping new neighbors?
	proximityRouting	false	
SimpleNetwork	constantDelay	50	constant delay between two peers (ms)
	jitter	0.01	average amount of jitter in %

An example setting for a run written in omnetpp.ini is shown in Table 5. In OverSim these kinds of run specific settings are collected to omnetpp.ini file which is presented as whole in Appendix E. If there are overlapping parameters in default.ini and omnetpp.ini files the parameter value written in the omnetpp.ini file will be used in the simulation run.

Table 5 – A Sample setting for a run from omnetpp.ini

parameter	value	description
description	Chord (Iterative)	description of the simulation scenario
network	SimpleNetwork	description of the underlay network
sim-time-limit	1209600	simulation time (s) (1209600s = 14d)
routingType	iterative	
lifetimeMean	69120	mean node lifetime (s)
globalFunctionsType	GlobalDHTTestMap	
useGlobalFunctions	1	are globalFunctions used? (1=true)
overlayType	ChordModules	
tier1Type	DHTModules	
tier2Type	DHTTestAppModules	
targetOverlayTerminalNum	4000	target number of active overlay terminals
testInterval	13091	mean interval for lookup messages from one node
putDelay	14400	interval for key put messages
initTime	47700	duration of the init phase (s)
initDelay	1.0	delay for storing the key (s)

5.6 Input parameters

The effect of four parameters is tested in our simulations. These parameters include the number of nodes in the network, lifetime of the nodes, time between key updates and time between FETCH-messages sent by each node. Other parameters remained constant at their default value while these four were varied one at a time. The parameter values used in the scenarios are tabulated in Table 6 with the default values written in boldface.

Table 6 - Input parameters for the simulations

Number of nodes	Node lifetime (hours)	Time between key updates (hours)	Time between FETCH messages (hours)
1000	4,8	1	0,91
2000	9,6	2	1,82
4000	**19,2**	**4**	**3,64**
6000	38,4	8	7,27
8000		12	
10000		16	
		20	
		24	

- SIMULATION TIME

  Simulation time for all scenarios is 14 days.

- NUMBER OF NODES

  This parameter describes the number of nodes participating in the overlay network. The default value (4000) is chosen to be such a value that fits the requirements of the simulation software and the available processing power. The aim is to simulate as large networks as possible. The tests with Kademlia DHT algorithm show that it would be too time taking to run the simulations with the default number of nodes at 10000. As the comparison of the two DHT algorithms is a fundamental part of this study, the default number of nodes has to be dropped to 4000. Had the simulation concerned only Chord DHT algorithm, the default number of nodes would have been at least 10000.

- MEAN NODE LIFETIME

  This parameter defines the mean value of the nodes' lifetime. Each node is assigned a random lifetime from the Weibull distribution. Node lifetimes are based on measurements in [Ver07]. The default value (19,2 hours) is set on the grounds of the mean number of power off switches in one day.

- KEY UPDATES

  Default time between key updates is chosen to be 4 hours. This value is set to be feasible with mean node lifetimes. The values are varied from one to 24 hours.

- KEY TTL

  The *time to live (TTL)* value for keys stored in nodes is set to be three times the key update value. This value is chosen to make the network able to withstand a situation where two consecutive key updates are lost.

- MEAN FETCH MESSAGE INTERVAL

  In our simulation FETCH messages are sent at exponentially distributed random intervals from each node. The parameter value is determined from usage intensity data presented in [Ver07]. From this data the information about outbound voice calls, video calls, sms- and mms messages and packet data sessions is added up. Table 7 depicts this evaluation process. The occurrence time of the next FETCH message is always scheduled at the time a node sends a FETCH message.

Table 7 – Determination of FETCH message interval

	mean number of events in a day
Outbound voice calls	2.67
Outbound video/data calls	0.10
Outbound sms messages	3.05
Outbound mms messages	0.13
packet data sessions	0.65
total	**6.6**

6.6 messages a day → one message in every 3.64 hours

5.7 Output parameters

From the simulation output there are seven parameters we are especially interested in. All the parameters are mean values over all of the nodes. Statistics are also collected about the largest value for each parameter in a single node.

- OVERALL BANDWIDTH USAGE

  Overall bandwidth usage is measured as received bytes per second in a node. Overall traffic includes all Store, Fetch, FindNode and maintenance messages.

- MESSAGE OVERHEAD FOR MAINTENANCE

  Message overhead for structure maintenance is measured as received bytes per second in a node. When the Chord-protocol is used, the maintenance messages include all Join, FixFingers, Stabilize, Notify and New Successor Hint messages. When Kademlia is used as the DHT protocol, the measurement of maintenance messages is more difficult as

Kademlia includes maintenance operations in other messages. Because of this the maintenance overhead statistics are collected only for scenarios using Chord.

- NUMBER OF LOOKUP HOPS

  The number of routing hops required for lookups was measured by calculating the number of nodes that must be visited until the node containing the requested information is found. The last hop added into the count is the one reaching that target node.
- LOOKUP MESSAGE OVERHEAD

  This parameter determines the number of messages and the volume of data transferred in a lookup process. Lookup message overhead is calculated by multiplying the number of lookup hops by the associated message sizes.
- LOOKUP DELAY

  The delay between sending a lookup and receiving the answer for it is defined by this parameter. In our simulations we use a fixed link delay for the UDP messages valued 50ms. A small jitter with a magnitude of one percent is included to prevent network-wide synchronization of message sending times. The delay used in our simulations is independent from the lookup message sizes. The delay is calculated from the instant a Fetch request message is sent to the instant a Fetch response message arrives.
- KEY DISTRIBUTION

  Key distribution is determined with the distribution of keys between the nodes. This is measured as keys per node. Also the total number of keys stored in the network is calculated.
- SUCCESS RATE

  Success rate of the FETCH-messages is the most important output parameter in our simulations. A successful FETCH message consists of a sent FETCH request and a received FETCH response. Success rate is determined as unique FETCH responses over unique FETCH requests.

5.8 Collecting the statistics

The statistics are collected using OverSim’s RECORD_STATS tool and our MyClass that includes a printStats function. MyClass prints a results.dat file that contains the statistics calculated in the class. The statistics collected with RECORD_STATS are automatically printed to a file called omnetpp.sca which is then processed with an OverSim script *overStat.pl* that prints the statistics in a more readable fashion.

MyClass collects statistics about:

- Number of hops in a lookup
- Lookup delay
- Number of keys in the network
- Maximum number of keys in a single node
- Keys per node

Statistics collected with RECORD_STATS tool include:

- Sent messages and bytes
- Received messages and bytes
- Lookup success rates
- Number of nodes joined and left
- Mean session times

Collecting statistics to two separate files is not an optimal but necessary resolution because the omnetpp.sca file reached its size limit when tests runs were executed. Dividing the statistics into two files it is possible to collect all the statistics from the simulations.

6 Results

This Chapter presents the results from the simulation scenarios. The results are illustrated with figures where the effects of the 4 input parameters (shown in Table 6) on the 7 output parameters discussed in Section 5.7 are depicted. The figures have 3 curves (excluding the maintenance traffic), one for each DHT algorithm – routing mode pair.

6.1 Overall bandwidth usage

Total traffic received per node was measured by adding up the maintenance traffic, *Store* and *Fetch* message traffic and *FindNode* message traffic (calls and responses). Figure 25 illustrates the results. Kademlia uses more than twice the bandwidth needed by Chord. The difference between the bandwidth consumption with the two DHT algorithms stays the same in all simulation scenarios.

Iterative routing mode in Chord uses more bandwidth than symmetric recursive mode. The maintenance traffic bandwidth usage results indicated that this would not be the case but when also the FindNode messages are taken into account the iterative mode demands more bandwidth.

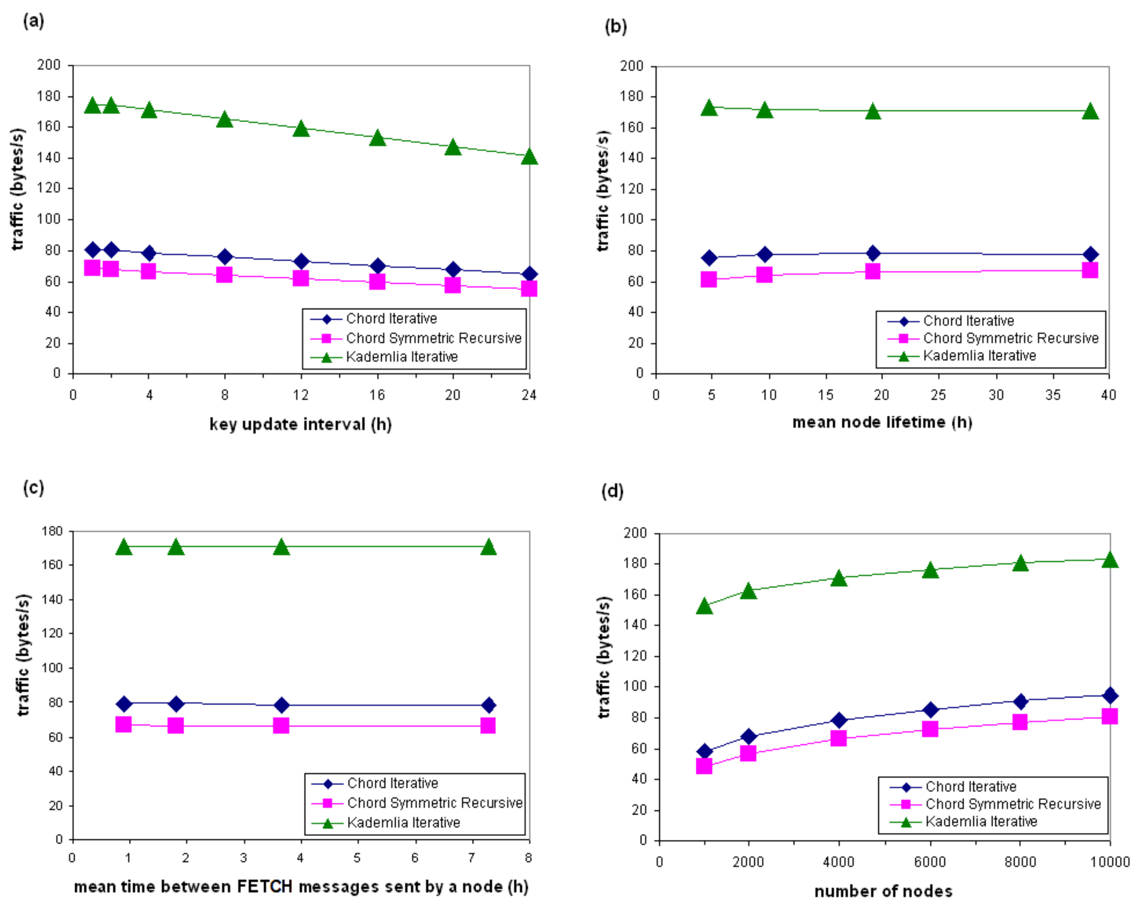


Figure 25 – Mean overall traffic received per node

6.2 Maintenance traffic bandwidth usage

The amount of maintenance traffic received per node was measured from the simulations. The results cover only the two scenarios where Chord was used as a DHT algorithm because Kademlia does not exploit separate maintenance messages and is therefore left outside of these measurements. The results are presented in Figure 26.

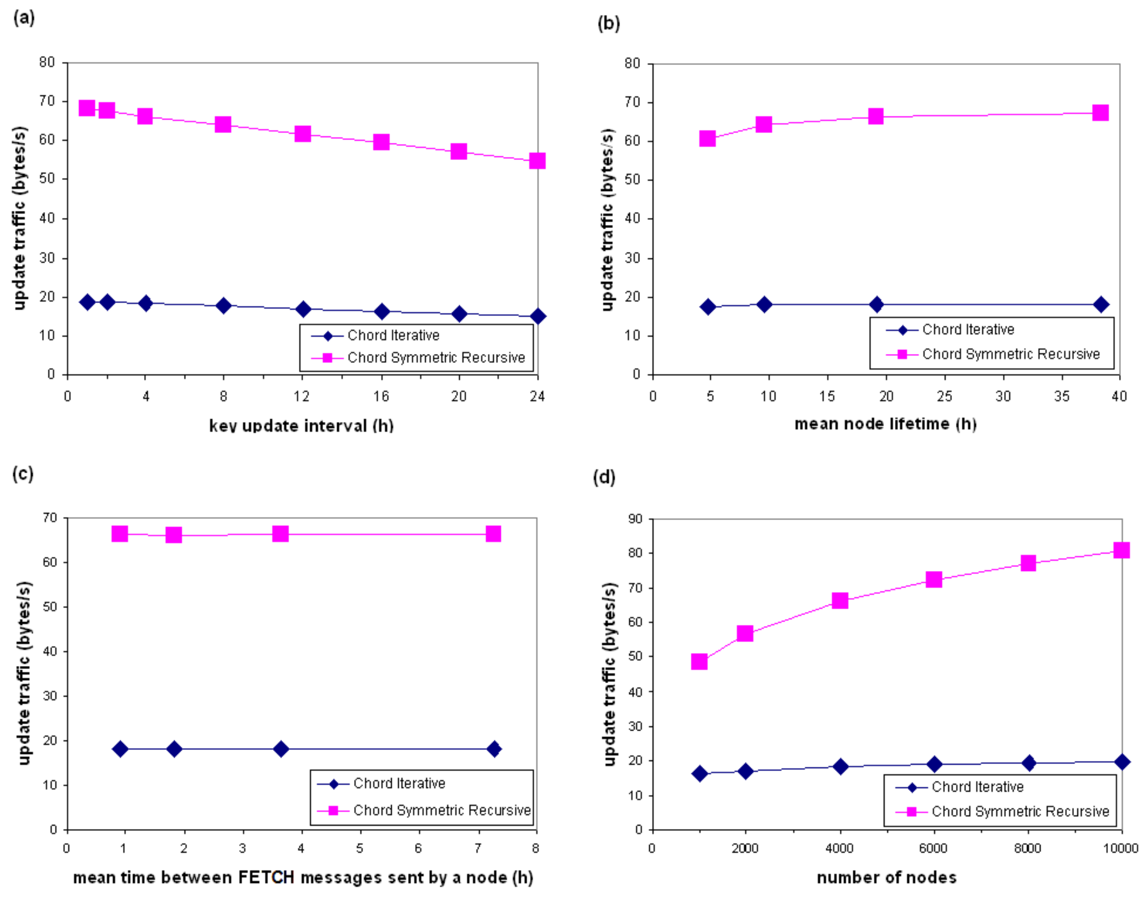


Figure 26 – Mean maintenance traffic received per node

There is a noticeable difference in the results between iterative and recursive routing modes. The reasons for this unexpected behaviour are discussed in more detail in Chapter 7. Other discoveries common to both routing modes are that:

- There is less maintenance traffic when key update interval is longer
- The maintenance traffic increases with an increasing node lifetime
- The maintenance traffic is, as expected, independent of the FETCH message interval
- A bigger network generates more maintenance traffic. The growth is in the order of O(log N). This is illustrated in Appendix F.

Figure 26d illustrates the amount of received maintenance traffic as the number of nodes in the network increases. Both iterative and recursive routing modes face an increased maintenance traffic as the number of nodes goes from 1000 to 10000. A noteworthy difference between the routing modes is the amount the maintenance traffic increases. With iterative mode there is an increase of 22% from a network of 1000 nodes to a 10000 node network. With recursive mode the increase is 67%. This is due to the difference in the number of FixFingers messages produced by the routing modes (see Section 7.1).

6.3 Number of hops for lookup

The number of routing hops to complete a lookup gives us information about the delay involved in the lookup process that is independent of the underlying topology and the actual link delays. The results are shown in Figure 27. Kademlia with iterative routing mode used less than half of the hops that were needed in the scenarios where Chord was used as the DHT protocol. The main observation from the results is that only the size of the network has an effect on the number of hops. The shapes in the Chord and Kademlia curves in Figure 27d show that they both scale logarithmically. For Chord both routing modes produce equivalent results.

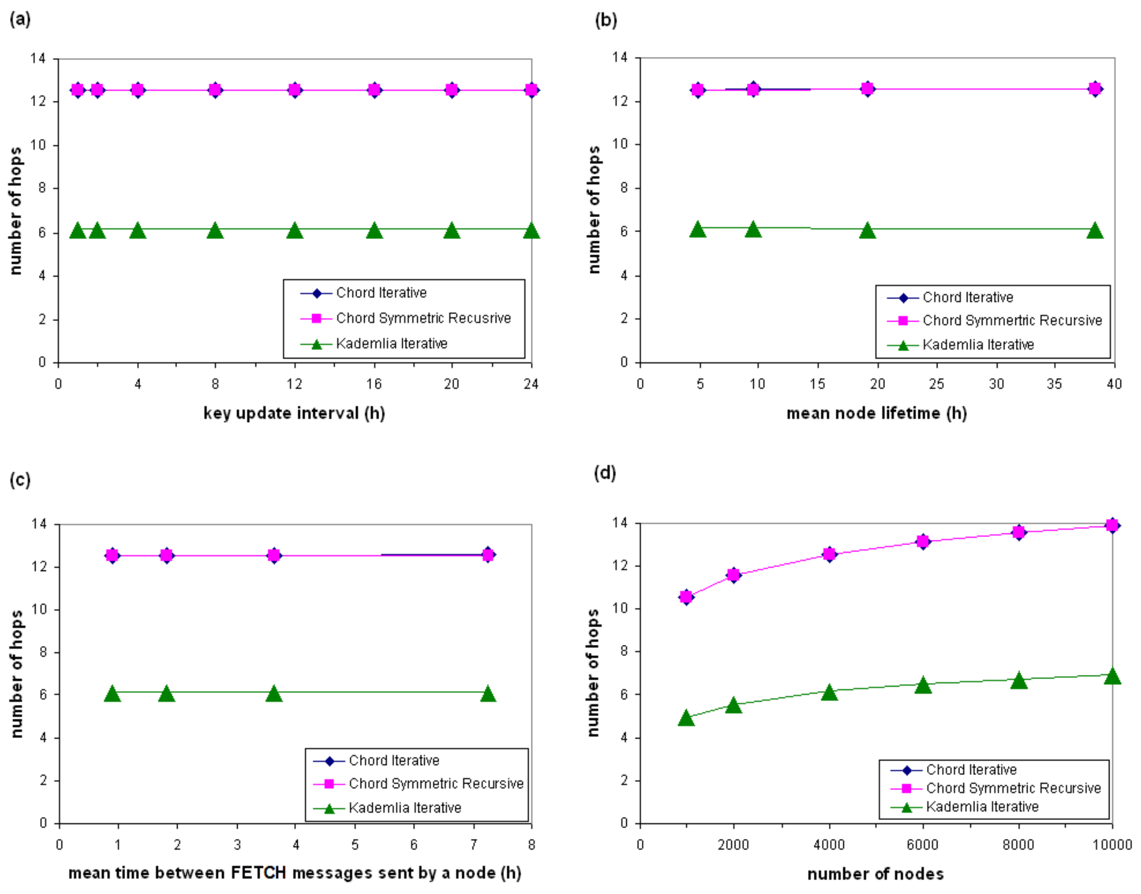


Figure 27 – Hops per lookup

6.4 Lookup message overhead

Lookup message overhead was calculated from the OverSim's overlay messages *FindNodeCall* and *FindNodeResponse*. These messages are used for locating the desired node. The motivation for this was to compare the DHT algorithms and their bandwidth usage with RELOAD. FindNodeCall has a constant size of 52 bytes while FindNodeResponse contains variable amount of data depending on the DHT algorithm and the parameters the algorithm is run with. In our simulations FindNodeResponse for Chord has a size of 59 bytes and for Kademlia a size of 241 bytes. The results are depicted in Figure 28.

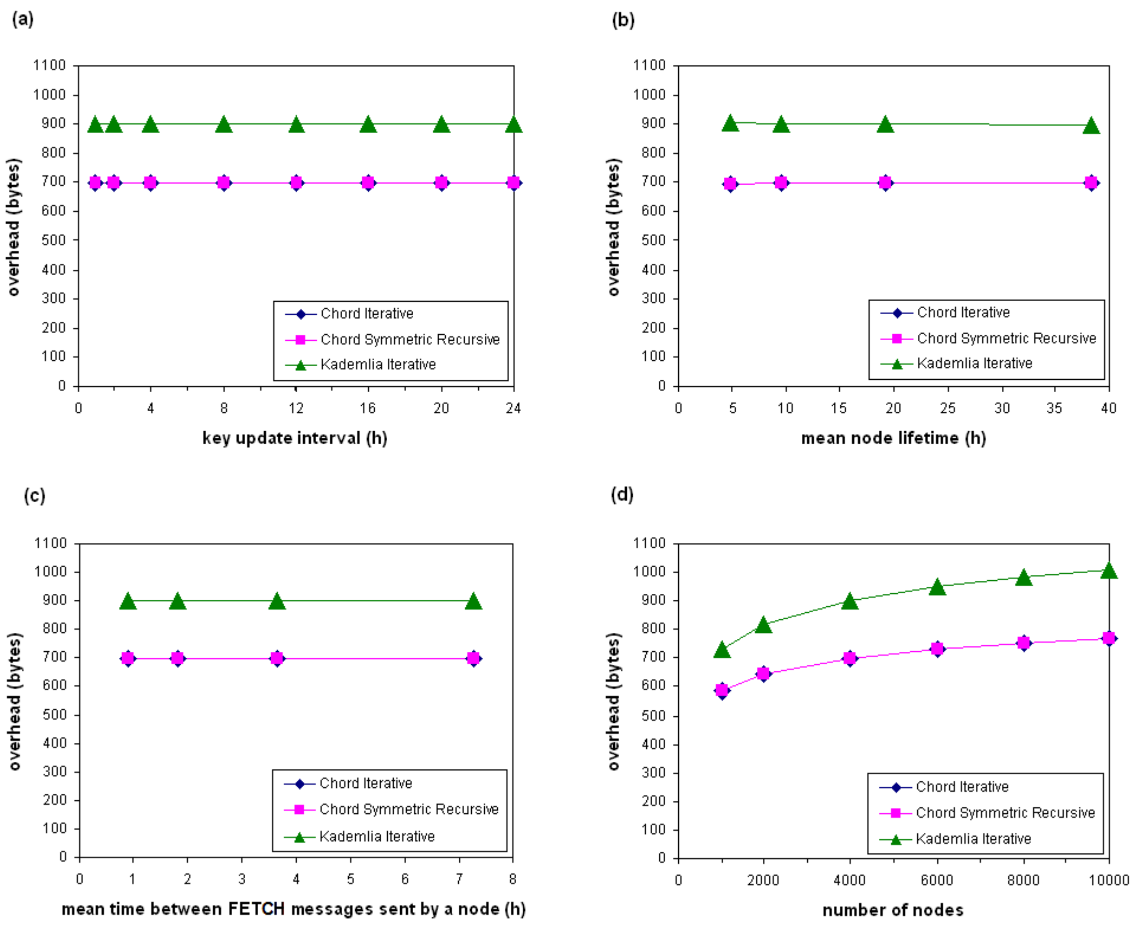


Figure 28 – Overhead caused by the lookup messages

Kademlia uses about 30% more bandwidth for lookups than Chord does. This difference grows as the number of nodes in the network increases. Combining these results with the hop count results above it can be noticed that the smaller hop count in Kademlia comes with a price in the form of larger lookup overhead.

6.5 Lookup delay

The delay of lookup was measured in seconds. Figure 29 presents the results. The results are similar with the number of hops results above. Only the number of nodes makes a difference on the delay.

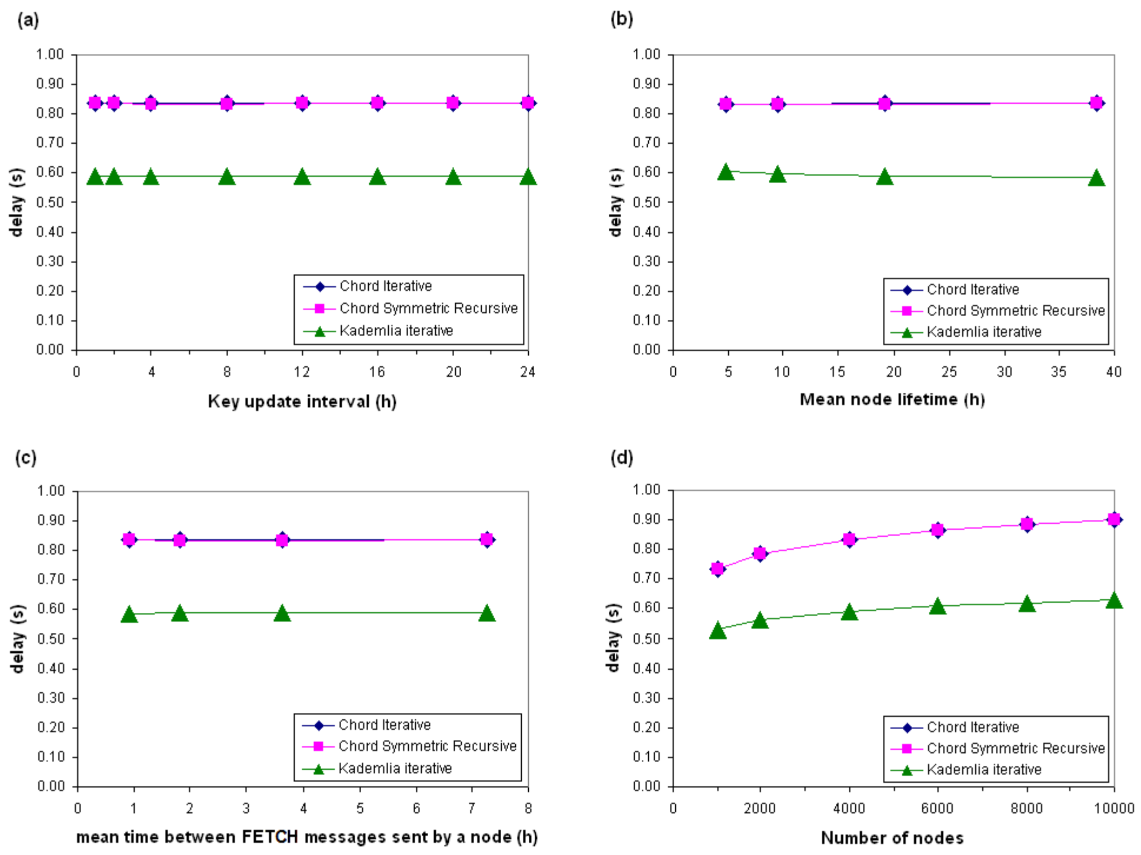


Figure 29 – Delay of the lookup process

6.6 Key distribution

The number of keys stored in a node is measured at the instant a node leaves the network. Adding up all of the nodes we have an estimate on the total number of keys stored in the network (including the duplicates) during the simulation. The keys stored in the nodes that are in the network when the simulation ends are calculated at that instant. Some keys are located in more than one node at a time because of the key lifetime and key update interval values used (see Section 7.2). The Keys per node value is calculated simply by dividing the total number of keys in the network by the number of nodes that participated in the network during the simulation.

The number of keys in the network and their distribution to nodes are illustrated in Figure 30 and Figure 31. From the figures following observations can be made:

- The longer the key update interval the larger the total number of keys and the more keys per node
- The longer the mean node lifetime the smaller the total number of keys and the less keys per node
- The larger the network the larger the total number of keys and the more keys per node.
- Kademlia stores less keys than Chord
- Recursive routing mode stores more keys than iterative routing mode when Chord is used

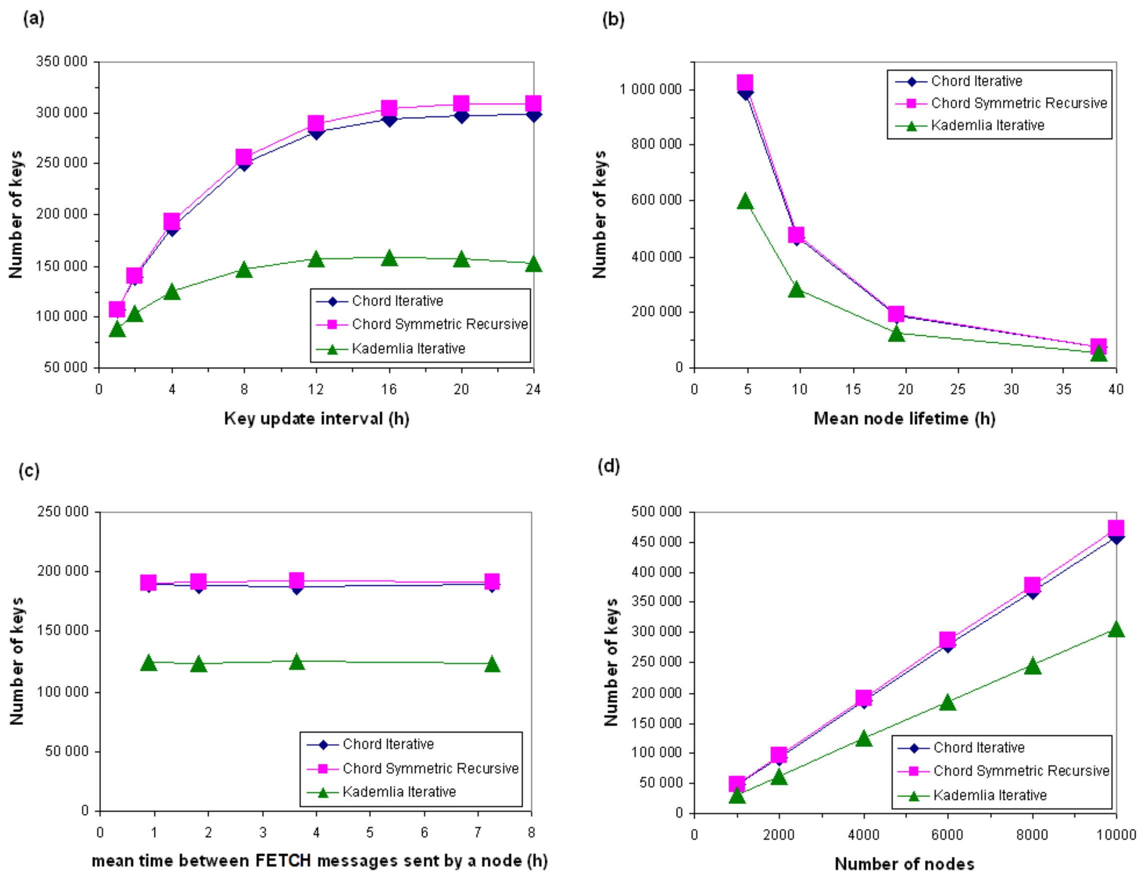


Figure 30 – Number of keys in the network

Figure 30d indicates a constant growth in total number of keys when the number of nodes in the network increases. Recursive Chord has the highest growth rate, iterative Chord growing at a slightly slower rate. Total number of keys grows significantly slower when Kademlia is used. Figure 31d also shows this constant rate of change. When the number of nodes grows, the number of keys per node stays almost fixed.

The key distribution with increasing key update intervals is depicted in Figure 31a. It can be seen that Chord and Kademlia graphs both follow the same trend although Kademlia produces less keys than Chord. The difference between Kademlia and Chord grows when the key update interval increases but the growth slows down with the larger values of key update interval.

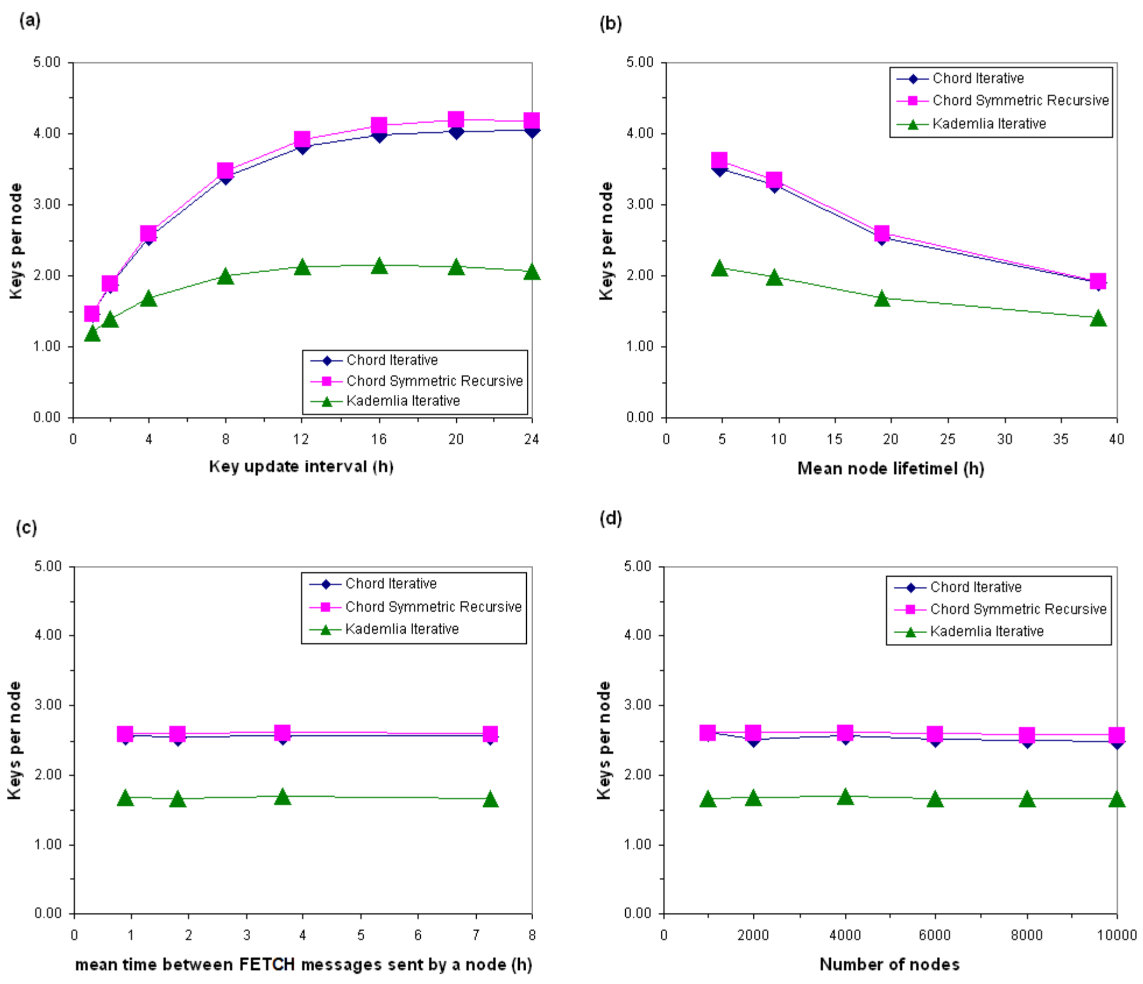


Figure 31 – Distribution of keys

The maximum values of keys per node are presented in Figure 32. The graphs follow loosely the same trends shown in Figure 31. The maximum number of keys a single node has at the instant it leaves the network grows with an increasing key update interval, decreases with an increasing mean node lifetime and grows slightly with an increasing network size. The main conclusion that can be drawn from the results is that Kademlia produces the smallest maximum values, which was presumable on the grounds of the total number of keys and their distribution.

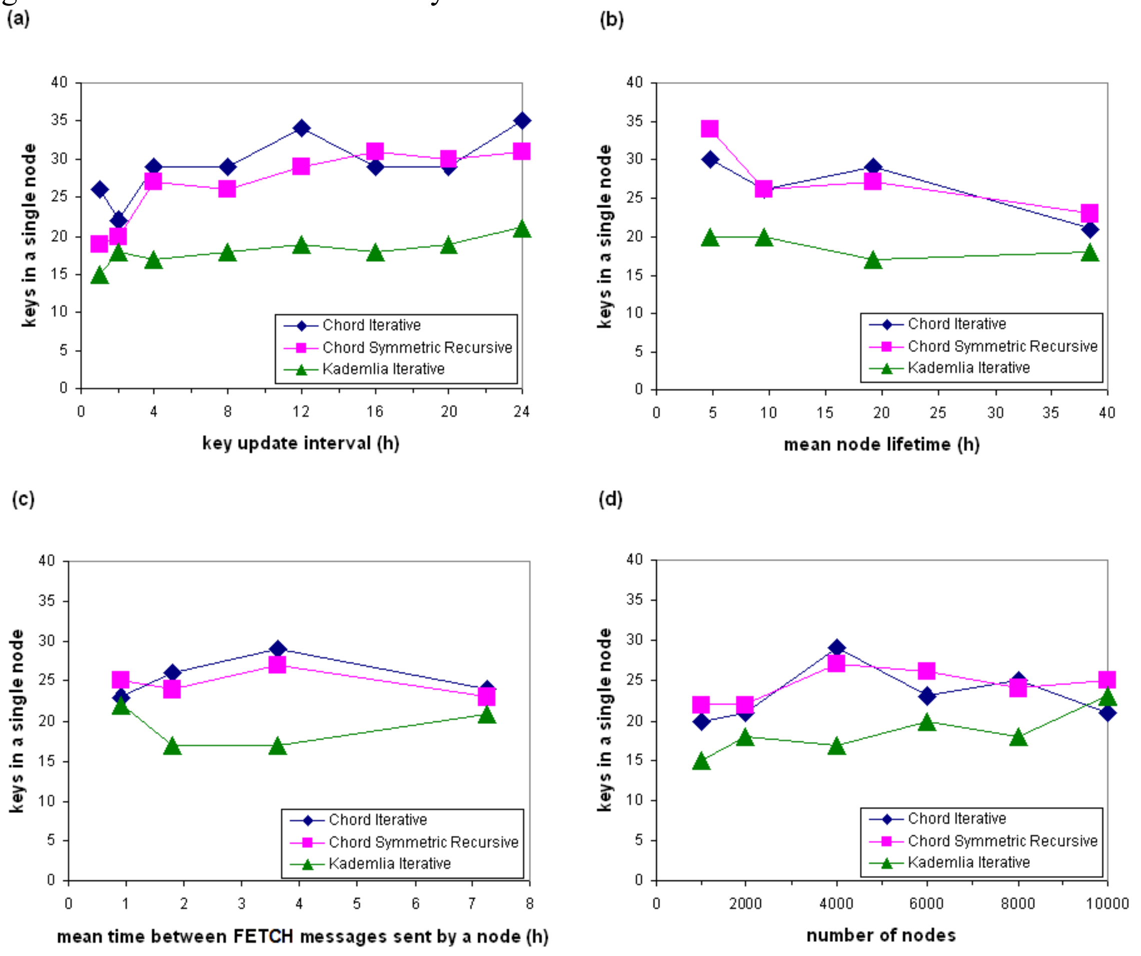


Figure 32 – Maximum number of keys stored in a single node

6.7 Lookup success rate

Figure 33 presents success rates of the lookups. From these results observations are that:

- The shorter the key update interval the higher the success rate
- The longer the mean node lifetime the higher the success rate
- The success rate decreases slowly as the network gets larger when using Chord with iterative routing mode.
- Kademlia has higher success rates than Chord
- Recursive Chord has slightly higher success rates than iterative Chord

Figure 33a demonstrates the trend of lookup success rates when the key update interval increases. Kademlia has the highest success rates and recursive Chord outperforms iterative Chord when the update interval becomes larger. In Figure 33d the success rates are plotted against network size. Kademlia maintains success rates as the number of nodes in the network increases. For recursive Chord there is a minor decrease in the success rate when the network gets bigger. For iterative Chord there is a slightly bigger decrease in the success rate than in recursive Chord's case.

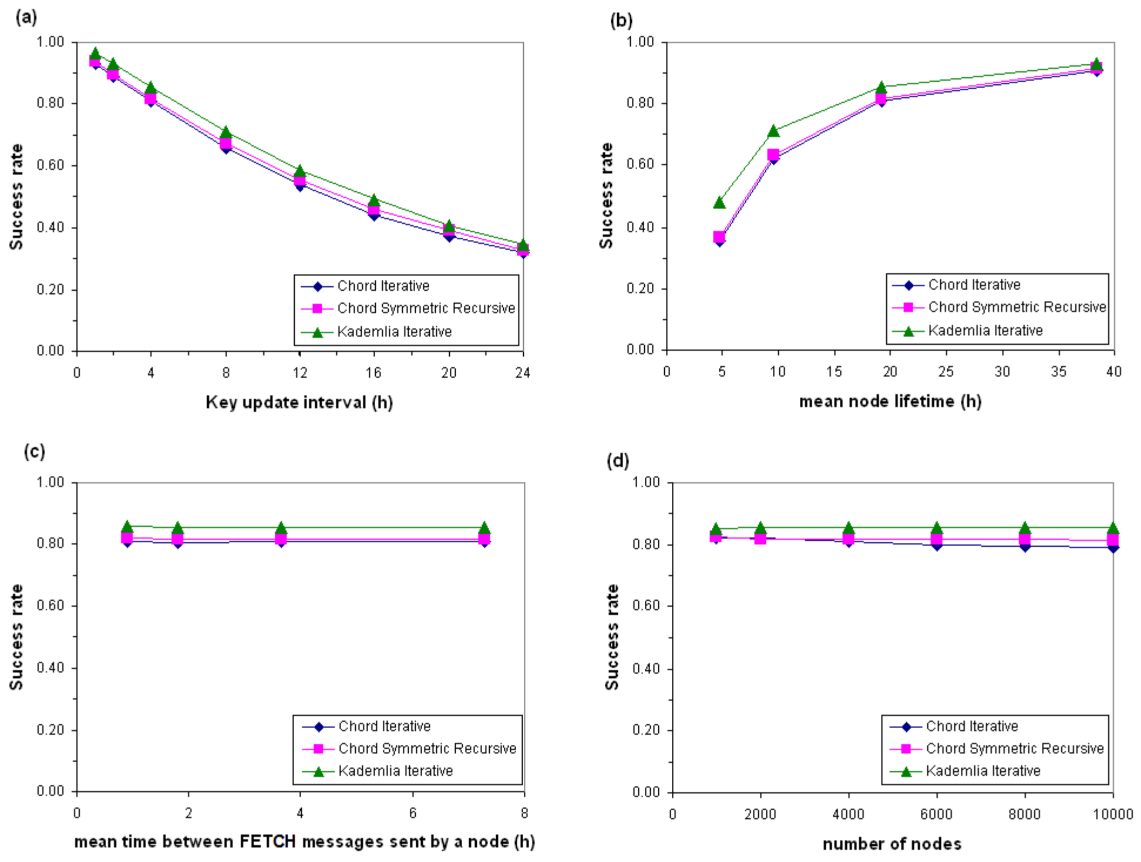


Figure 33 – Lookup success rates

Two test runs were carried out after obtaining the lookup success rate results. Chord with iterative routing mode was used. The idea was to test some shorter key update intervals and see if they give better results. The key update intervals we used were 30 minutes, 5 minutes and 1 minute. The highest lookup success rate, 96.6%, was obtained with the key update interval of 5 minutes.

7 Discussion

This Chapter presents discussion about the results and the reasons that affected the results. The results are also compared with the findings from other simulation studies. Finally the simulated system's applicability for the current mobile telephone networks is estimated on the grounds of the results.

7.1 Differences in maintenance traffic volume for Chord

A difference between the two routing modes when using Chord was observed in chapter 6.2. To better understand the large difference, the output of the OverSim simulator was studied and reasons for the difference were discovered. First the difference was narrowed to concern only the Fix Fingers messages. Studies showed that in recursive routing mode Chord produced much more Fix Fingers messages than it did when the iterative routing mode was used. After this discovery the Fix Fingers routine was more carefully examined for both routing modes.

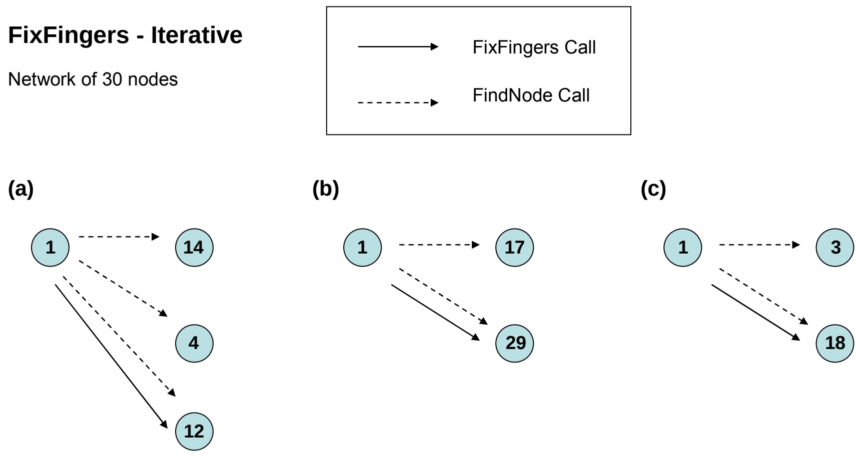


Figure 34 – FixFingers procedure with iterative Chord

To examine the Fix Fingers routine a test network of 30 nodes was set up. The progress of the routine was followed from the output shown on the OverSim's graphical user interface. The observations made during these examinations are presented in Figure 34 and Figure 35. In Figure 34 node 1 starts its Fix Fingers procedure which consists of three independent phases. Node 1 first locates the recipient using FindNode call messages. As the recipient (node 12 in Figure 34a) is found, node 1 then sends a FixFingers call to that node.

Node 1 needs to contact two nodes (4 and 14) before locating node 12 to which it then sends the FixFingers call. In the second and third phase (Figure 34b and 31c) only one node needs to be contacted before locating the target node. Altogether this means three FixFingers calls and seven FindNode calls for the procedure when iterative routing mode is used.

Figure 35 illustrates the Chord's FixFingers procedure for the recursive routing mode with the same 30 node network that was used above for iterative Chord. This time node 2 starts the FixFingers procedure. The difference with the iterative routing mode is that only FixFinger calls are used i.e. no FindNode calls are sent. Node 2 initiates the procedure which consists of 6 phases. For some reason the first three phases are identical and contain two FixFinger calls. Phases 4 and 6 contain three FixFinger calls and the procedure contains in total 14 FixFinger calls.

Although the reason for this behaviour remains unclear, this is definitely why recursive routing mode creates more FixFingers messages and thereby more maintenance messages than iterative mode. The difference grows when the number of nodes in the network increases as can be seen in Figure 26.

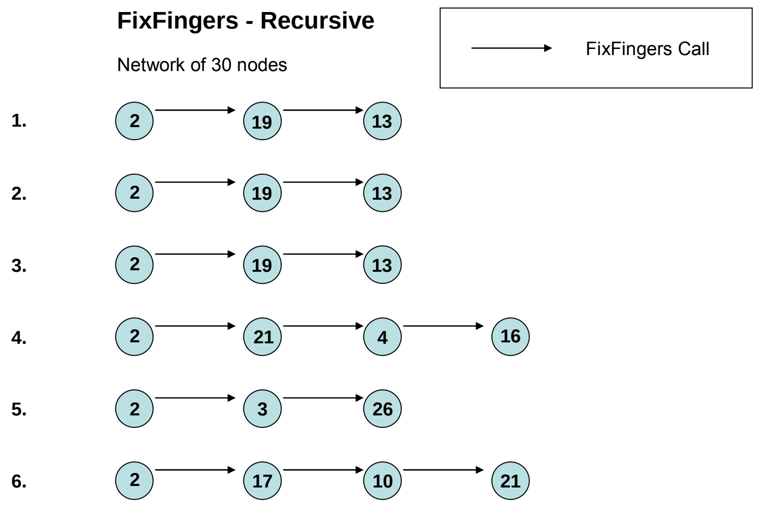
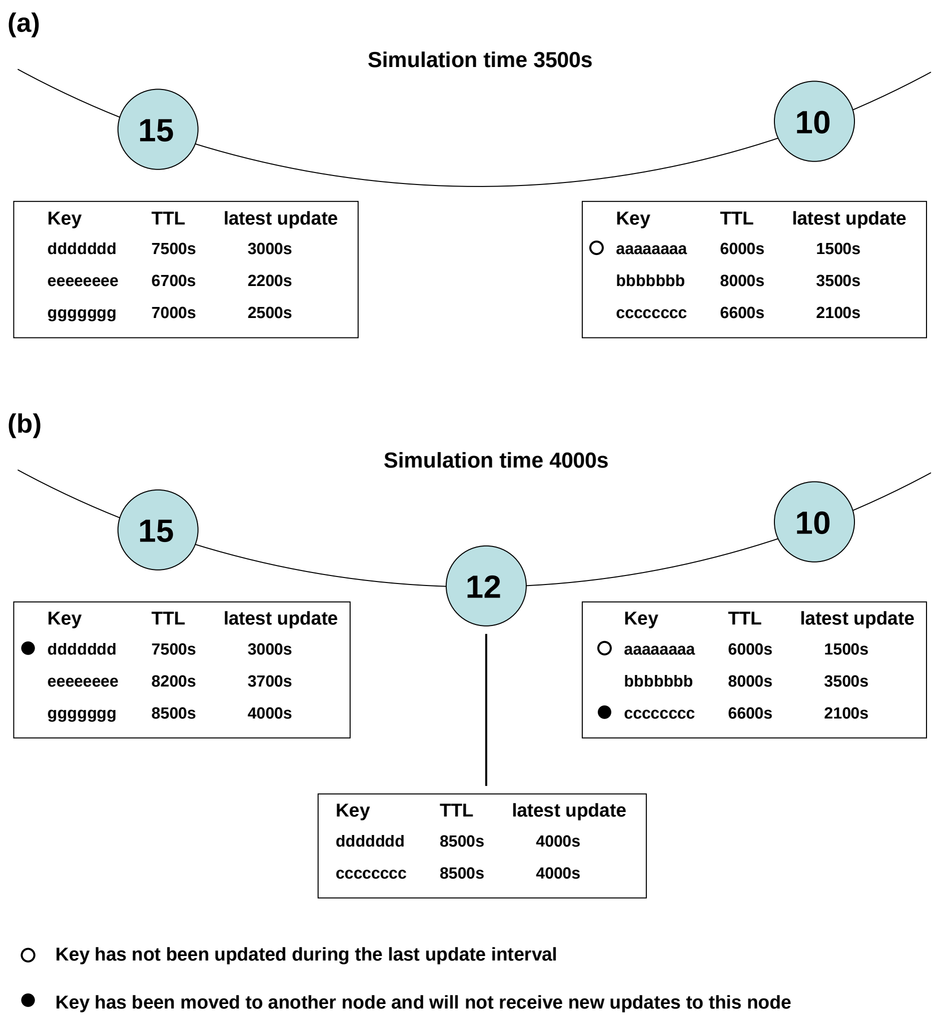


Figure 35 – FixFingers procedure with recursive Chord

7.2 Differences in key distribution

The key distribution results presented in Section 6.5 show a significant difference between Chord and Kademlia DHT algorithms. Because the difference is greater than one would have expected, some discussion about the reasons affecting the key distribution is appropriate.

The DHT module in OverSim takes care of the key storage updating as new nodes join the network. The DHT module provides the newly arrived nodes with the appropriate keys. The keys will remain also in the node previously responsible for them until the TTL timer for those keys runs out. Key updates only update the TTL value to the node which at that instant is responsible for storing that key. When nodes leave the network the keys stored in them are not transferred to another node. This means that those keys stay unreachable until the *key update timer* expires and the nodes owning the keys store them to new nodes. Figure 36 illustrates the key storage and updating.



(a) Node 15:

	Key	TTL	latest update
	ddddddd	7500s	3000s
	eeeeeeee	6700s	2200s
	ggggggg	7000s	2500s

(a) Node 10:

	Key	TTL	latest update
○	aaaaaaaa	6000s	1500s
	bbbbbbb	8000s	3500s
	cccccccc	6600s	2100s

(b) Node 15:

	Key	TTL	latest update
●	ddddddd	7500s	3000s
	eeeeeeee	8200s	3700s
	ggggggg	8500s	4000s

(b) Node 10:

	Key	TTL	latest update
○	aaaaaaaa	6000s	1500s
	bbbbbbb	8000s	3500s
●	cccccccc	6600s	2100s

(b) Node 12:

Key	TTL	latest update
ddddddd	8500s	4000s
cccccccc	8500s	4000s

○ Key has not been updated during the last update interval

● Key has been moved to another node and will not receive new updates to this node

Figure 36 – Key storage, updating and TTL values

In Figure 36a the simulation time is 3500 seconds and two nodes 10 and 15 are lying adjacent in a Chord overlay. Both nodes are storing three keys whose TTL values and the simulation times the keys were last updated are shown. Key update interval is 1500s and TTL value is 4500s. Node 10 has a key which has not been updated and will be removed at simulation time 6000s unless an update is received. Figure 36b shows the same nodes at simulation time 4000s. A new node with id 12 has just joined the network and received responsibility for two keys one previously stored in node 10 and the other in node 15. These two keys are stored in two nodes until their TTL timers run out. These keys' TTL values are updated only in node 12 and in the lookup perspective these keys reside only in node 12.

There are significant differences in keys per node values between Chord and Kademlia, which can be read in Figure 31. The scenarios using Chord have over 50% more keys per node than the scenario using Kademlia. The reason for this is an *aggressive join mode* that OverSim uses with Chord by default. With aggressive join mode Chord's *stabilize* procedure (see chapter 2.3.2) is called every time a new node joins the network. This leads up to a situation where many keys are stored into two or even more nodes simultaneously. This theory was tested by setting the aggressive join mode off and modifying the *stabilize delay* parameter in Chord. It turned out that setting stabilize delay to one second gave very similar results with the aggressive join mode. Using the default stabilize delay of 120 seconds did not work at all because many keys just disappeared from the network for an undefined reason.

The increase of keys per node depicted in Figure 31 derives from the definition of key TTL value (Chapter 5.1). The TTL value is three times the value of key update interval. As the key update interval increases it is evident that there will be more keys in the network. Keys can be stored in the network for up to three days after the node identified by the key has left the network. In these simulations no additional replicas were used i.e. number of replicas was one. This means that a key will be stored only in one node at a time or more precisely only one location for each node is known by the DHT. Taking only the TTL value into account a key can theoretically be located in three different nodes at a time. Each key update can store the key to a new node if during the update interval a new node has joined the network with an identifier closer to the key than the node the key was previously stored into. Newly joined nodes also receive keys from their predecessor node and successor node when they join the network.

Although keys might reside in many nodes, the DHT algorithm with replica value set to one only looks for them in one node. This way the duplicate keys are of no use for the lookup process. When number of replicas is increased also the algorithm changes and lookup messages are no more targeted to one node only. This key relocation shows especially in the longer update intervals, when the TTL of a key is very high and many nodes join the network during the key's lifetime. This can be observed in Figure 31a.

7.3 Lookup success rates

At first glance the lookup success ratios seem relatively low. There are, however, several reasons for these results. The most important thing affecting the lookup success rates is the number of replicas. Number of replicas corresponds to the number of nodes a key is stored to ensure the availability of a key as the network faces churn. As stated in the previous section the number of replicas was set to one. This setup highlights the drop in the lookup rates that is illustrated in Figure 33a. From the research point of view the decision not to use any replicas can be justified by the fact that the effect of other parameters on the lookup success ratio can be determined better.

The default value for the key update interval could have been chosen to be smaller to give higher lookup success rates. More accurate study on the effect of key update logic including the adjustment of the key TTL value could give higher lookup success rates.

7.4. Comparison with other simulation studies

A previous study [Hon+08] has been made by Hong & Schulzrinne from Columbia University in New York collaborating with Hilt from Bell Labs. The paper studies how much bandwidth is needed in order to maintain certain success ratios for Chord's *fixfingers* and *join* processes.

Node lifetimes used in the paper are 30min, 1h, 5h and 10h. The results from the paper show a decrease in control message traffic as the churn rate increases (i.e. node lifetime decreases). Similar behaviour can be noticed in the results of this study. Figure 26 shows that as the mean node lifetime decreases (i.e. the churn rate increases) there is a decrease in the maintenance message traffic.

In our study the churn rates are smaller than in [Hon+08]. The node lifetimes used in this study are 4.8h, 9.6h, 19.2h and 38.4h. The bandwidth results in [Hon+08] give higher traffic rates than the results in this study. This can be because of several reasons. One of them is that in [Hon+08] JOIN messages have been included in the control message count whereas in this study they have not. The parameters used in Chord are not revealed in the paper.

According to [Hon+08] the probability of success for *fixfingers* process decreases when the churn rate increases. As a successful fixfingers process involves contacting several nodes consecutively, it has similar properties with the lookup process studied in this thesis. A decrease in lookup success rates when the churn rate increases (i.e. mean node lifetime decreases) in our study is shown in Figure 33b.

7.5 Applicability for mobile telephone networks

The highest mean maintenance traffic per node measured in these simulations is 80,7 bytes/s and the highest maintenance traffic during one simulation run measured in one node is 1610 bytes/s.

Figure 37 presents extrapolation curves for the mean maintenance traffic when the number of nodes increases. The bandwidth usage grows logarithmically and scales well even with a network of ten million nodes. Overall bandwidth usage with extrapolation curves is depicted in Figure 38.

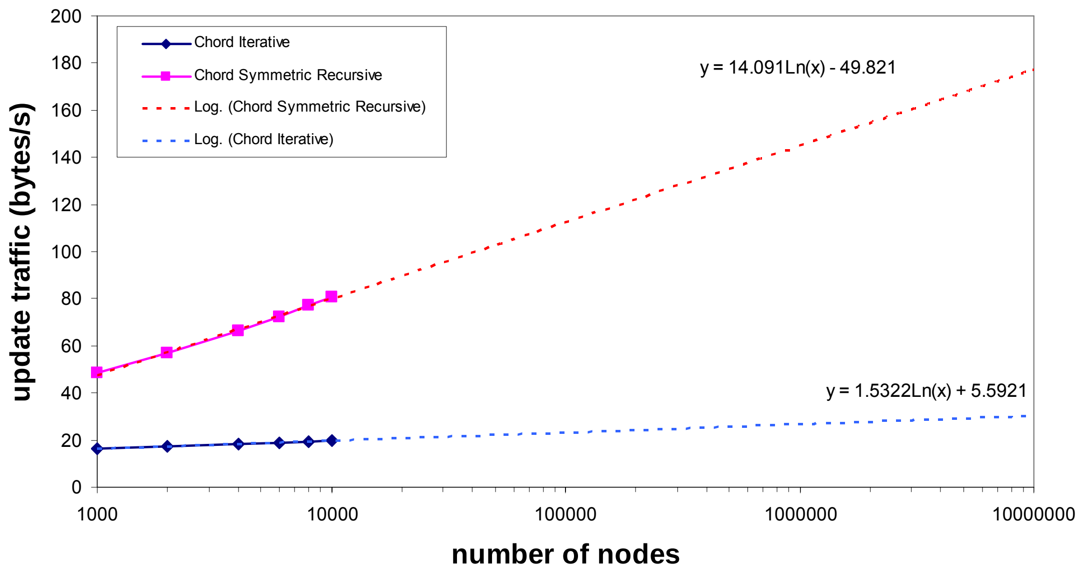


Figure 37 – Extrapolation curves for maintenance bandwidth usage

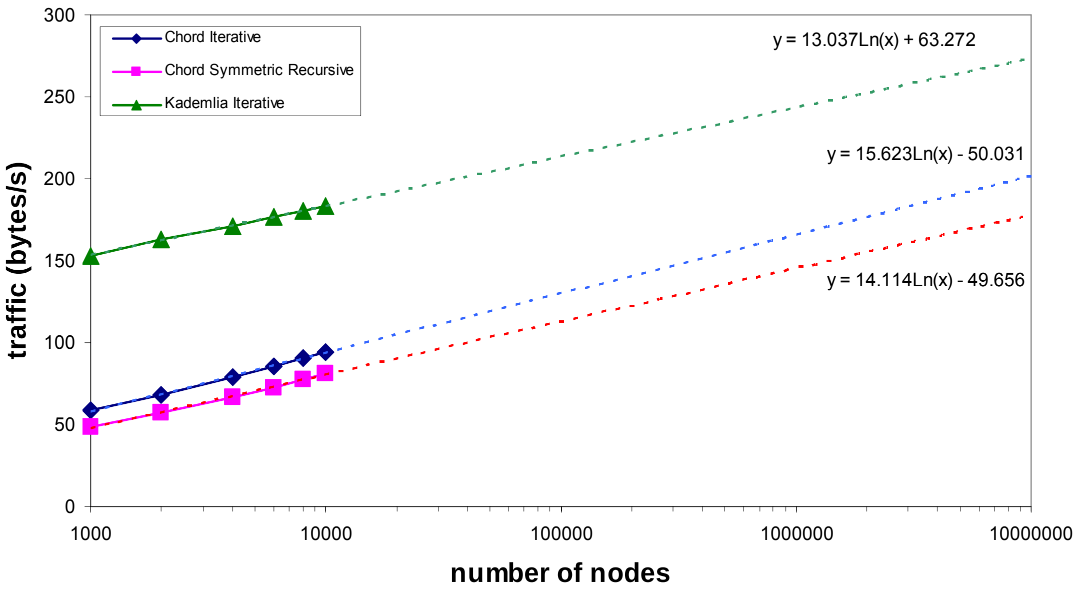


Figure 38 – Extrapolation curves for overall bandwidth usage

The bandwidth limitations in different mobile telephone networks are presented in Table 8. As we can see from the table only GSM data could have difficulties in providing enough bandwidth for P2PSIP's maintenance traffic. Networks with other techniques should work fine.

Table 8 – Theoretical maximum bandwidths for mobile telephone networks [Leh06]

GSM Data	9.6 kbps
HSCSD	57.6 kbps
GPRS	115 kbps
EDGE	384 kbps
WCDMA	2 Mbps DL / 384 kbps UL
HSDPA	3.6 Mbps DL / 384 kbps UL
HSUPA	14 Mbps DL / 5.76 Mbps UL

When we consider using P2PSIP with RELOAD in mobile telephone networks, the first thing that should be taken care of is increasing the lookup success rate to at least 95 percent. For networks using Chord this would require a key update interval of about 30 minutes. In our simulations networks using Kademlia reached a lookup success rate of 97 percent with a key update interval of one hour. If the mean node lifetime is much lower than our default 19,2 hours even a shorter key update interval can be needed. Updating the keys more frequently will increase the bandwidth usage. Another adjustment to improve the lookup success rate also increasing bandwidth usage is to make the DHT algorithm send maintenance messages more frequently. The effects of the different parameters in Chord and Kademlia can also be investigated in order to find out which method gives the best lookup success rate results without increasing the bandwidth usage too much.

Even though P2PSIP is applicable for mobile telephone networks as far as bandwidth consumption is concerned, this does not mean that there are not any other barriers. If a P2P application for example keeps the CPU of the mobile device up all the time, the battery should run out in 5 to 7 hours.

8 Conclusions

The objective of this Master's thesis was to study and simulate a mobile P2P network using a novel P2PSIP protocol which is still under development. The most important goal was to investigate the bandwidth needed for P2PSIP network's maintenance traffic in order to find out if the data rates of different mobile telephone networks are adequate to P2PSIP based overlay networks. The effect of several input parameters to the required bandwidth was tested with simulations. A special interest was the comparison between two DHT algorithms Chord and Kademlia as well as the routing mode selection between iterative and recursive styles. Simulation scenarios were built with OverSim overlay network simulator. The necessary additions were coded into the OverSim code and altogether 57 scenarios were simulated. This thesis presents the results from those simulation runs and gives guidelines for further simulations based on these results.

Kademlia DHT algorithm was simulated only in iterative routing mode because the test simulations with recursive routing gave results that were inconsistent with other test results. The reason for this inconsistency could not be solved with a reasonable amount of work so the recursive routing mode for Kademlia had to be discarded from the scope of this thesis.

8.1 Main Results

The results show that 2G mobile telephone networks can handle the bandwidth usage of a P2PSIP network operating with RELOAD peer protocol. This was proven for a network of ten thousand nodes and estimates are given that this would hold even up to network sizes of ten million nodes. Kademlia outperforms Chord when lookup delay and lookup success rate of two DHT algorithms are compared, but uses more than twice as much bandwidth. With our simulation parameters the bandwidth usage of Kademlia is, however, moderate and it will not be a problem.

The lookup success rates are, with the input parameters used in our study, in general too low for a satisfactory session establishment. With a key update interval of one hour we have lookup success rates of 93% for Chord and 97% for Kademlia. This is the absolute minimum value to use for key update interval in this kind of system. Even a key update interval of 5 minutes did not raise the lookup success rates for Chord higher than 97%. This value is too low for demanding users and applications.

8.2 Working with OverSim

The level of unambiguity of the results did not quite match the expectations. The high detail level in OverSim makes it challenging to produce a desired simulation scenario first time around. A highly delailed simulator is surely a good thing when there is unlimited time to do research but the fact is that OverSim cannot be mastered in a short period of time. One contribution this thesis has to give is to point out some features of OverSim, which should be taken into account when simulating overlay networks and DHT algorithms. The features covered in chapters 7.1 – 7.3 should be examined in more detail before making a new simulation study.

8.3 Future Research

For the future research a more advanced version of the RELOAD draft or perhaps already an RFC would make a better starting point for a study. Once there are more standardized elements in the P2PSIP and RELOAD protocols more detailed studies can be made.

To develop this study more exact input parameters should be used. This thesis gives information for the future research about how to choose the parameter range. Some parameter values could be chosen from real test networks that P2PSIP has been run on. One especially useful piece of information to get from the test network environment would be the message sizes that obviously make a great effect on the bandwidth usage.

The effect of replication would be a good thing to investigate. The reasons behind the low lookup success rates also need more research. Future research on replication could give us answers also on the lookup success rate issue.

References

[Ali+05] Alima L.O, Ghodsi, Haridi S. A framework for structured peer-to-peer overlay networks, in Global computing, vol. 3267, Lecture Notes in Computer Science: Springer Berlin/ Heidelberg, 2005, pp. 223-249

[Ban+05] Banks J, Carson J, Nelson B, Nicol D. Discrete-Event System Simulation (Fourth Edition), Prentice Hall International Series in Industrial and Systems Engineering. 2005

[Bau+07] Baumgart I, Heep B, Krause S, OverSim: A Flexible Overlay Network Simulation Framework. in Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007, Anchorage, USA, May 2007.

[Bry+08] Bryan D, Matthews P, Shim E, Willis D, Dawkins S. Concepts and Terminology for Peer to Peer SIP. draft-ietf-p2psip-concepts-02 http://tools.ietf.org/html/draft-ietf-p2psip-concepts-02

[Cam02] Camarillo G, SIP Demystified. McGraw-Hill 2002, ISBN 0-07-137340-3

[Cam+08] Camarillo C, Garcia-Martin M. The 3G IP Multimedia Subsystem (IMS) 3rd edition. Wiley 2008, ISBN 978-0-470-51662-1

[Cha+03] Chawathe Y, Ratnasamy S, BreslauL, Lanham N, Shenker S. Making gnutella-like P2P systems scalable. Proc. 2003 conference on Applications, Technologies, Architectures and Protocols for Computer Communications, August 25-29 2003, pp. 407-418.

[Coo+07] Cooper E, Johnston A, Matthews P. Bootstrap Mechanisms for P2PSIP. draft-matthews-p2psip-bootstrap-mechanisms-00 http://www.p2psip.org/drafts/draft-matthews-p2psip-bootstrap-mechanisms-00.txt

[Dab+04] Dabek F, Li J, Sit E, Robertson J, Kaashoek M, Morris R. Designing a DHT for low latency and high throughput. In Proceedings of the USENIX Symposium on Networked Systems Design and Implementation, Mar 2004.

[Gka+04] Gkantsidis C, Mihail M, Saberi A. Random Walks in Peer-to-Peer Networks. In Proceedings of IEEE INFOCOM, 2004

[God+05] Godfrey PB, Stoica I. Heterogenity and Load Balance in Distributed Hash Tables. In Proceedings of the IEEE INFOCOM, Miami, FL, Mar. 2005.

[Gum+03] Gummadi K, Gummadi R, Gribble S, Ratnasamy S, Shenker S, Stoica I. The Impact of DHT Routing Geometry on Resilience and Proximity. In Proceedings of the ACM SIGCOMM Aug 2003.

[Har+04] Harjula E., Ylianttila M., Ala-Kurikka J., Riekki J., Sauvola J. Plugand-Play Application Platform: Towards Mobile Peer-to-Peer. In: ThirdInternational Conference on Mobile and Ubiquitous Multimedia (MUM2004). College Park, MD, USA, pp. 63-69, 2004.

[Har07] Harjula E, Peer-to-Peer SIP in Mobile Middleware Intercommunication. M.Sc. thesis, Department of Electrical and Information engineering, University of Oulu, Finland, Sep 2007.

[Har+07] Harjula E, Heikkinen M, Salinas A, Hautakorpi J, Beijar N, Ou Z. DECICOM Decentralized Inter-Service Communications State-of-The-Art Study, 2007.

[Hel03] Hellerstein J, Toward Network Data Independence, ACM SIGMOD Record 32 (3) 2003.

[Hon+08] Hong S G, Hilt V and Schulzrinne H. Evaluation of Control Message Overhead of a DHT-Based P2P System. Bell Labs Technical Journal Volume 13 Issue 3, pages 79-86, 2008

[Jen+08] Jennings C, Lowekamp B, Rescorla E, Baset S, Schulzrinne H. REsource LOcation And Discovery (RELOAD) draft-ietf-p2psip-reload-00. http://www.p2psip.org/drafts/draft-ietf-p2psip-reload-00.txt , Jul 2008.

[Kov07] Kovacevic A, On Benchmarking of Peer-to-Peer Overlays, Technische Universität Darmstadt, Germany, Nov 2007. http://www.kom.tu-darmstadt.de/~sandra/Kovacevic_2.Milestone_Adaptability_ScalabilityAndStability.pdf, Retrieved on 25.5.2009.

[Kun05] Kunzmann G, Recursive or iterative routing? Hybrid!. In the proceedings of the Gesellschaft für Informatik (GI) 2005.

[Las07] Lassila P. Lecture notes of course S-38.3148 Simulation of data networks,
Department of Communications and Networking, Helsinki University of Technology, Finland, 2007.

[Leh05] Lehtinen M, NAT Traversal Techniques, Special Assignment, Department of Communications and Networking, Helsinki University of Technology, Finland, 2005.

[Leh06] Lehtinen J, Design and Implementation of Mobile Peer-to-Peer Application. M.Sc. thesis, Department of Communications and Networking, Helsinki University of Technology, Finland, 2006

[Li+05] Li J, Stribling J, Kaashoek F, Morris R, and Gil T. A Performance vs. Cost Framework for Evaluating DHT Design Tradeoffs under Churn. In IEEE INFOCOM, Miami, FL, Mar. 2005.

[May+02] Maymounkov P, Mazières D. Kademlia: A peer-to-peer Information System Based on the XOR Metric. In Proceedings of the IPTPS 2002, Boston, Mar 2002.

[Pet+03] Peterson L, Davie B. Computer Networks – A Systems Approach 3rd edition. Morgan Kaufmann 2003. ISBN 1-55860-833-8

[RFC2327] Handley M, Jacobson V. Network Working Group, Request for Comments: 2327 - SDP: Session Description Protocol. Apr 1998

[RFC 3261] Rosenberg J, Schulzrinne H, Camarillo G, Johnston A, Peterson J, Sparks R, Handley M, Schooler E. Network Working Group, Request for Comments: 3261 – SIP: Session Initiation Protocol. Jun 2002.

[RFC4981] Risson J, Moors T, Network Working Group, Request for Comments: 4981 – Survey of Research towards Robust Peer-to-Peer Networks. Sep 2007.

[Ros07] Rosenberg J. Interactive Connectivity Establishment (ICE) : A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. draft-ietf-mmusic-ice-19 http://tools.ietf.org/html/draft-ietf-mmusic-ice-19 , Oct 2007.

[Sep07] Seppänen J. Prospects of Peer-to-Peer SIP for Mobile Operators, M.Sc. thesis, Department of Communications and Networking, Helsinki University of Technology, Finland, 2007

[Son+08] Song H, Jiang X, Matuszewski M, Ekberg J-E, Laitinen P. Security requirements in Peer-to-Peer Session Initiation Protocol (P2PSIP), http://www.p2psip.org/drafts/draft-matuszewski-p2psip-security-requirements-03.txt , Jul 2008.

[Sto+01] Stoica I, Morris R, Krager D, Kaashoek M, Balakrishnan H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In Proceedings of the ACM SIGCOMM 2001, San Diego, CA, USA, Aug 2001.

[Stu+06] Stutzbach D, Reza R, Understanding Churn in Peer-to-Peer Networks. In the proceedings of Internet Measurement Conference (IMC), Oct. 2006.

[Ver07] Verkasalo H. A Cross-Country Comparison of Mobile Service and Handset Usage. Licenciate Thesis on Helsinki University of Technology, Finland, Jan 2007.

[Web1] OverSim: The Overlay Simulation Framework
http://www.oversim.org/

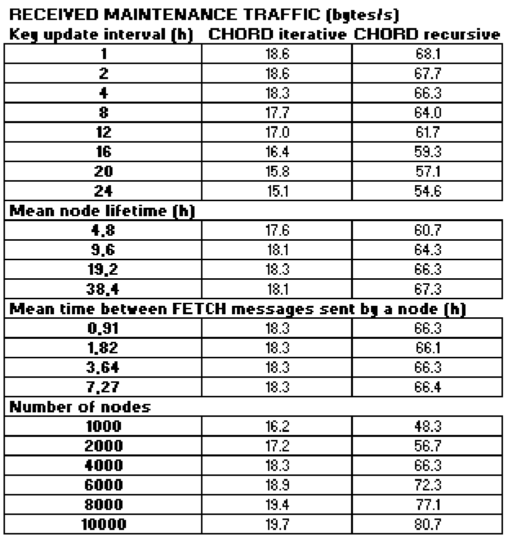
[Web2] OMNeT++ Community Site
http://www.omnetpp.org/

[Web3] SANYO Semiconductor Co.,Ltd. Quality and Reliability Handbook ver.3 Chapter 8-1-2 Probability Distribution Functions.
http://www.semiconductor-sanyo.com/reliability/main.asp?-part=8&id=M30A156 Retrieved on 15.5.2009

[Wu+06] Wu D, Tian Y, Ng K-W, Analytical Study on Improving DHT Lookup Performance under Churn. In Proceedings of the IEEE P2P'06, Cambridge UK, Sep 2006.

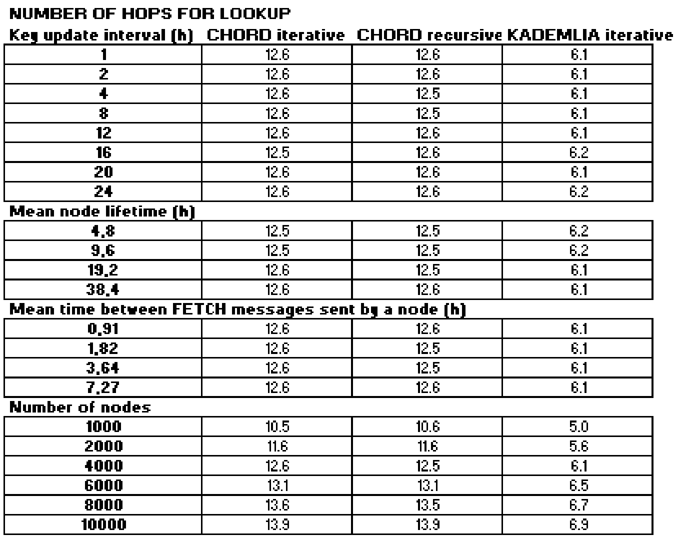
Appendices

A Tabulated Results



RECEIVED MAINTENANCE TRAFFIC (bytes/s)

Key update interval (h)	CHORD iterative	CHORD recursive
1	18.6	68.1
2	18.6	67.7
4	18.3	66.3
8	17.7	64.0
12	17.0	61.7
16	16.4	59.3
20	15.8	57.1
24	15.1	54.6
Mean node lifetime (h)		
4,8	17.6	60.7
9,6	18.1	64.3
19,2	18.3	66.3
38,4	18.1	67.3
Mean time between FETCH messages sent by a node (h)		
0,91	18.3	66.3
1,82	18.3	66.1
3,64	18.3	66.3
7,27	18.3	66.4
Number of nodes		
1000	16.2	48.3
2000	17.2	56.7
4000	18.3	66.3
6000	18.9	72.3
8000	19.4	77.1
10000	19.7	80.7



NUMBER OF HOPS FOR LOOKUP

Key update interval (h)	CHORD iterative	CHORD recursive	KADEMLIA iterative
1	12.6	12.6	6.1
2	12.6	12.6	6.1
4	12.6	12.5	6.1
8	12.6	12.5	6.1
12	12.6	12.6	6.1
16	12.5	12.6	6.2
20	12.6	12.6	6.1
24	12.6	12.6	6.2
Mean node lifetime (h)			
4,8	12.5	12.5	6.2
9,6	12.5	12.5	6.2
19,2	12.6	12.5	6.1
38,4	12.6	12.6	6.1
Mean time between FETCH messages sent by a node (h)			
0,91	12.6	12.6	6.1
1,82	12.6	12.5	6.1
3,64	12.6	12.5	6.1
7,27	12.6	12.6	6.1
Number of nodes			
1000	10.5	10.6	5.0
2000	11.6	11.6	5.6
4000	12.6	12.5	6.1
6000	13.1	13.1	6.5
8000	13.6	13.5	6.7
10000	13.9	13.9	6.9

LOOKUP OVERHEAD (bytes)

Key update interval (h)	CHORD iterative	CHORD recursive	KADEMLIA iterative
1	697.0	696.9	899.4
2	696.7	697.1	899.9
4	696.8	696.3	900.6
8	697.0	696.4	900.4
12	696.9	697.0	900.1
16	696.4	697.3	901.1
20	697.2	697.0	900.6
24	697.8	697.4	902.0
Mean node lifetime (h)			
4.8	694.1	694.2	903.1
9.6	696.2	695.3	902.2
19.2	696.8	696.3	900.6
38.4	697.5	696.9	898.0
Mean time between FETCH messages sent by a node (h)			
0.91	697.2	697.1	898.4
1.82	697.1	696.5	899.5
3.64	696.8	696.3	900.6
7.27	697.5	696.8	900.4
Number of nodes			
1000	584.6	586.1	730.2
2000	641.5	642.6	816.0
4000	696.8	696.3	900.6
6000	729.5	728.9	951.0
8000	752.3	752.0	983.6
10000	770.0	769.7	1009.9

LOOKUP DELAY (s)

Key update interval (h)	CHORD iterative	CHORD recursive	KADEMLIA iterative
1	0.83	0.83	0.59
2	0.83	0.83	0.59
4	0.83	0.83	0.59
8	0.83	0.83	0.59
12	0.83	0.83	0.59
16	0.83	0.83	0.59
20	0.83	0.83	0.59
24	0.84	0.83	0.59
Mean node lifetime (h)			
4,8	0.83	0.83	0.60
9,6	0.83	0.83	0.59
19,2	0.83	0.83	0.59
38,4	0.83	0.83	0.58
Mean time between FETCH messages sent by a node (h)			
0,91	0.83	0.83	0.59
1,82	0.83	0.83	0.59
3,64	0.83	0.83	0.59
7,27	0.84	0.83	0.59
Number of nodes			
1000	0.73	0.73	0.53
2000	0.78	0.78	0.56
4000	0.83	0.83	0.59
6000	0.86	0.86	0.61
8000	0.88	0.88	0.62
10000	0.90	0.90	0.63

TOTAL NUMBER OF KEYS

Key update interval (h)	CHORD iterative	CHORD recursive	KADEMLIA iterative
1	107 531	107 680	88 969
2	138 855	140 202	104 046
4	187 957	192 792	125 100
8	249 847	255 749	147 829
12	281 014	289 431	157 395
16	293 763	304 073	158 717
20	296 867	308 921	157 533
24	298 077	309 330	153 153
Mean node lifetime (h)			
4,8	990 851	1 025 490	600 453
9,6	470 035	479 549	285 070
19,2	187 957	192 792	125 100
38,4	74 525	75 036	54 871
Mean time between FETCH messages sent by a node (h)			
0,91	189 548	190 769	123 671
1,82	188 177	191 011	122 889
3,64	187 957	192 792	125 100
7,27	189 113	191 684	122 640
Number of nodes			
1000	48 082	48 175	31 080
2000	92 922	96 361	62 494
4000	187 957	192 792	125 100
6000	278 639	286 397	185 124
8000	368 729	377 757	245 844
10000	458 513	472 968	305 331

MEAN NUMBER OF KEYS PER NODE

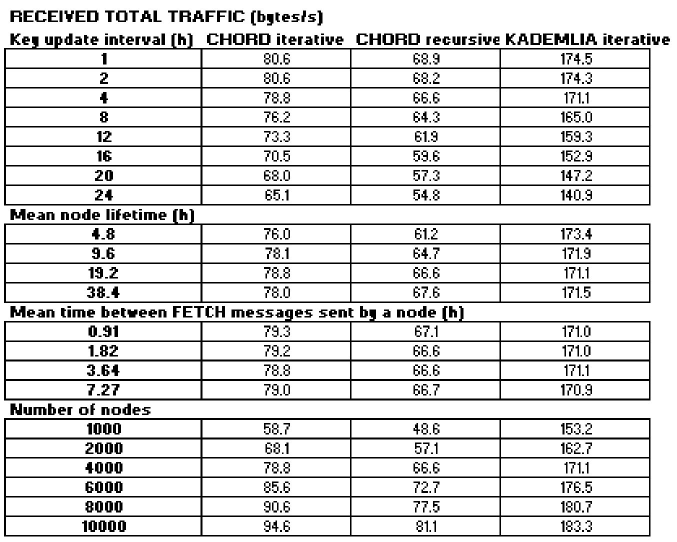
Key update interval (h)	CHORD iterative	CHORD recursive	KADEMLIA iterative
1	1.46	1.46	1.21
2	1.88	1.90	1.41
4	2.55	2.60	1.69
8	3.39	3.47	2.00
12	3.81	3.92	2.13
16	3.98	4.12	2.16
20	4.02	4.20	2.13
24	4.04	4.17	2.08
Mean node lifetime (h)			
4,8	3.50	3.62	2.13
9,6	3.27	3.33	1.98
19,2	2.55	2.60	1.69
38,4	1.91	1.92	1.41
Mean time between FETCH messages sent by a node (h)			
0,91	2.57	2.59	1.68
1,82	2.55	2.58	1.67
3,64	2.55	2.60	1.69
7,27	2.56	2.59	1.67
Number of nodes			
1000	2.61	2.60	1.67
2000	2.53	2.60	1.69
4000	2.55	2.60	1.69
6000	2.52	2.59	1.67
8000	2.50	2.57	1.67
10000	2.50	2.58	1.67

MAXIMUM NUMBER OF KEYS IN ONE NODE

Key update interval (h)	CHORD iterative	CHORD recursive	KADEMLIA iterative
1	26	19	15
2	22	20	18
4	29	27	17
8	29	26	18
12	34	29	19
16	29	31	18
20	29	30	19
24	35	31	21
Mean node lifetime (h)			
4,8	30	34	20
9,6	26	26	20
19,2	29	27	17
38,4	21	23	18
Mean time between FETCH messages sent by a node (h)			
0,91	23	25	22
1,82	26	24	17
3,64	29	27	17
7,27	24	23	21
Number of nodes			
1000	20	22	15
2000	21	22	18
4000	29	27	17
6000	23	26	20
8000	25	24	18
10000	21	25	23

LOOKUP SUCCESS RATES

Key update interval (h)	CHORD iterative	CHORD recursive	KADEMLIA iterative
1	0.93	0.94	0.97
2	0.89	0.90	0.93
4	0.81	0.82	0.85
8	0.66	0.67	0.71
12	0.54	0.55	0.58
16	0.44	0.46	0.49
20	0.38	0.39	0.41
24	0.32	0.33	0.35
Mean node lifetime (h)			
4.8	0.36	0.37	0.48
9.6	0.62	0.63	0.71
19.2	0.81	0.82	0.85
38.4	0.91	0.92	0.93
Mean time between FETCH messages sent by a node (h)			
0.91	0.81	0.82	0.86
1.82	0.81	0.82	0.86
3.64	0.81	0.82	0.85
7.27	0.81	0.82	0.85
Number of nodes			
1000	0.82	0.82	0.85
2000	0.82	0.82	0.86
4000	0.81	0.82	0.85
6000	0.80	0.82	0.86
8000	0.79	0.82	0.85
10000	0.79	0.81	0.86



RECEIVED TOTAL TRAFFIC (bytes/s)

Key update interval (h)	CHORD iterative	CHORD recursive	KADEMLIA iterative
1	80.6	68.9	174.5
2	80.6	68.2	174.3
4	78.8	66.6	171.1
8	76.2	64.3	165.0
12	73.3	61.9	159.3
16	70.5	59.6	152.9
20	68.0	57.3	147.2
24	65.1	54.8	140.9
Mean node lifetime (h)			
4.8	76.0	61.2	173.4
9.6	78.1	64.7	171.9
19.2	78.8	66.6	171.1
38.4	78.0	67.6	171.5
Mean time between FETCH messages sent by a node (h)			
0.91	79.3	67.1	171.0
1.82	79.2	66.6	171.0
3.64	78.8	66.6	171.1
7.27	79.0	66.7	170.9
Number of nodes			
1000	58.7	48.6	153.2
2000	68.1	57.1	162.7
4000	78.8	66.6	171.1
6000	85.6	72.7	176.5
8000	90.6	77.5	180.7
10000	94.6	81.1	183.3

B MyClass.h

```
// MyClass.h
#include <string>
using std::string;
#include<OverlayKey.h>
#include<NodeHandle.h>
#include <omnetpp.h>

#ifndef MYCLASS_H
#define MYCLASS_H


class MyClass {
public:
    MyClass();
    static void print();

    static std::map<OverlayKey, int> lookupHops;
    static std::map<OverlayKey, double> luDelay;

    static void addMaintenance(int join, int notify, int stabilize,
int newSuc, int fixFing);
    static void addHop(int hopCount);
    static void addHop2(int hopCount);
    static void addKeys(int keys);
    static void reset();
    static void printLookups();
    static void addLookup(OverlayKey key, double simtime);
    static void addDelay(double delay);
    static void removeLookup(OverlayKey key);
    static bool isValidLookup(OverlayKey key);
    static void addHandle(IPvXAddress ip, NodeHandle handle);
    static NodeHandle getHandle(IPvXAddress ip);

    static double hops;
    static bool iterative;
    static double finished_lookups;
    static double removed_lookups;
    static double lookup_calls;
    static int remaining_lookups ;

    static double numberOfNodes;
    static double numberOfKeys;
    static double maxKeys;
    static double minKeys;

    static double stabilizeInterval;
    static double fixFingersInterval;
    static double time;

    static string dht;
    static double lookupFreq;
    static double churnRate;
    static int terminalsAdded;
    static int terminalsRemoved;

    static int networkSize;
    static int routingTableSize;
    static string routingModel;
    static int aid;

// ####  KADEMLIA ############
    static int k;
    static int s;
    static int b;

    static double hopsPerLookup;
    static double lookupMessageOverhead;
    static double lookupDelay;
    static double keysPerNode;

};

#endif
```

C MyClass.cc

```
// MyClass.cc
#include <iostream>
using std::cout;
using std::cin;
using std::ios;
using std::cerr;
using std::endl;
using namespace std;

#include <fstream>
using std::ofstream;

#include <cstdlib>

#include <string>
using std::string;

#include "MyClass.h"

typedef std::map<OverlayKey, int> lookupHops;
typedef std::map<OverlayKey, double> luDelays;

lookupHops lookups;
luDelays delays;

double MyClass::hops = 0;
double MyClass::finished_lookups = 0;
double MyClass::time = 0;
bool MyClass::iterative = true;

double MyClass::removed_lookups = 0; // used for calculating lookups
that take too long and are removed
double MyClass::lookup_calls = 0;
int MyClass::remaining_lookups = 0; // lookups that are in progress
when simulation stops

double MyClass::numberOfNodes = 0;
double MyClass::numberOfKeys = 0;
double MyClass::maxKeys = 0;
double MyClass::minKeys = 1000;

string MyClass::dht = "empty";
double MyClass::lookupFreq = 0;
double MyClass::churnRate = 0;
int MyClass::terminalsAdded = 0;
int MyClass::terminalsRemoved = 0;

int MyClass::networkSize = 0;
double MyClass::stabilizeInterval = 0;
double MyClass::fixFingersInterval = 0;
int MyClass::routingTableSize = 0;
string MyClass::routingModel = "empty";
int MyClass::aid = 0;

// ### KADEMLIA #######
int MyClass::k = 0;
int MyClass::s = 0;
int MyClass::b = 0;

double MyClass::hopsPerLookup = 0;
double MyClass::lookupMessageOverhead = 0;
double MyClass::lookupDelay = 0;
double MyClass::keysPerNode = 0;

MyClass::MyClass() {}

// calculates the hopcount
void MyClass::addHop(int hopCount)
{
            hops = hops + hopCount;
            finished_lookups++;
            hopsPerLookup = hops / finished_lookups;
}
// calculates the number of keys in the network
// and the max/min number of keys in one node
void MyClass::addKeys(int keys)
{
            numberOfNodes++;
            numberOfKeys += keys;
            keysPerNode = numberOfKeys / numberOfNodes;

            if (keys < minKeys){
                        minKeys = keys;
            }

            if (keys > maxKeys){
                        maxKeys = keys;
            }
}
// inserts lookups and simTimes to map containers
void MyClass::addLookup(OverlayKey key, double simtime)
{
    lookups.insert(make_pair(key, 0));
            delays.insert(make_pair(key, simtime));
            lookup_calls++;
}

void MyClass::addDelay(double delay)
{
            lookupDelay += delay;
}
// removes lookups from the map container
void MyClass::removeLookup(OverlayKey key)
{
            bool skip = false;

//          checks that the key exists in the lookups map
            lookupHops::iterator it;
            it=lookups.find(key);
            if (it == lookups.end())
                        return;


//  if delay > 5 seconds the lookup is removed from the map
        double delaytmp = (simulation.simTime() - delays[key]);
        if (delaytmp > 5) {
                cout << "DELAY " << delaytmp << endl;
                delays.erase(key);
                lookups.erase(key);
                skip = true;
                ++removed_lookups;
        }

//      removes all the lookups currently having a delay > 5 seconds from the
map
        luDelays::iterator iter;
        for (iter = delays.begin(); iter != delays.end(); iter++) {
                double delaytmp =(simulation.simTime()-(iter->second));
                if (delaytmp > 5) {
                     OverlayKey temp = iter->first;
```

```
                        delays.erase(temp);
                        lookups.erase(temp);
                        ++removed_lookups;
                }
        }

if (skip == false) {
                                    // calculates the number of hops in this
lookup
            int hoptmp = (lookups[key] -1)*2;
            addHop(hoptmp);
            lookups.erase(key);

                                    // calculates the delay of this lookup
            double delaytmp = (simulation.simTime() - delays[key]);
            addDelay(delaytmp);
            delays.erase(key);
            }
}
}
// checks that there is a ongoing lookup for a specific key
bool MyClass::isValidLookup(OverlayKey key)
{
            map<OverlayKey, int, double>::iterator iter;
            bool tmp = false;
            for (iter = lookups.begin(); iter != lookups.end(); iter++) {
                        if (iter->first.compareTo(key) == 0 && time !=
                        simulation.simTime() ){
                                    tmp = true;
                                    iter->second++;
                        }
            }
            // simulation time stored to allow only one call of this function
at a time
            time = simulation.simTime();
            return tmp;

}

// resets the initial values
void MyClass::reset()
{
            MyClass::hops = 0;
            MyClass::finished_lookups = 0;

            MyClass::removed_lookups = 0;
            MyClass::lookup_calls = 0;
            MyClass::remaining_lookups = 0;

            MyClass::numberOfNodes = 0;
            MyClass::numberOfKeys = 0;
            MyClass::maxKeys = 0;
            MyClass::minKeys = 1000;

            MyClass::stabilizeInterval = 0;
           MyClass::fixFingersInterval = 0;

            MyClass::dht = "empty";
            MyClass::lookupFreq = 0;
            MyClass::churnRate = 0;
            MyClass::networkSize = 0;
            MyClass::routingTableSize = 0;
            MyClass::routingModel = "empty";
            MyClass::aid = 0;
            MyClass::iterative = true;

            MyClass::k = 0;
            MyClass::s = 0;
            MyClass::b = 0;

            MyClass::hopsPerLookup = 0;
            MyClass::lookupMessageOverhead = 0;
            MyClass::lookupDelay = 0;
            MyClass::keysPerNode = 0;

}

// prints the results to results.dat
void MyClass::print()
{
            ofstream outResultFile( "results.dat", ios::app );

    outResultFile << "DHTused " << dht << " variable" << " variable" << ' ' <<
"RoutingModel " << routingModel << ' ' << "LookupCalls "
<< lookup_calls << ' ' <<
"FinishedLookups: " << finished_lookups << ' '
<< "removedLookups " << removed_lookups << ' ' << "SuccessRate " <<
finished_lookups/(lookup_calls - remaining_lookups) << ' ' << "Added " <<
terminalsAdded
<< ' ' << "Removed " << terminalsRemoved << ' ' << "HopsPerFinishedLookup: "
<< hopsPerLookup << ' ' << "Lookup_delay: " << lookupDelay/finished_lookups <<
' ' << "Network_Size: " << networkSize << ' ' << "Stabilize_interval: " <<
stabilizeInterval  << ' ' << "FixFingers_interval " << fixFingersInterval << '
' << "Hops: " << hops << ' ' << "Keys_total " << numberOfKeys << ' ' <<
"Max_keys " << maxKeys << ' ' << "Min_keys " <<
minKeys << ' ' << "Keys_per_node: " << keysPerNode << endl;

}
```

D Additions to OverSim code

OverSim file:	`GlobalStatistics.cc`
Function:	`doFinish()`
Added code:	`MyClass::print();` `MyClass::reset();`

OverSim file:	`DHTTestApp.cc`
Function:	`handleTimerEvent(cMessage* msg)`
Added code:	`MyClass::addLookup(key, simulation.simTime());`

OverSim file:	`DHT.cc`
Function:	`handleGetResponse(DHTGetResponse* dhtMsg)`
Added code:	`MyClass::removeLookup(key);`
Function:	`finishApp()`
Added code:	`MyClass::addKeys(dataStorage->getSize());`

OverSim file:	`Chord.cc`
Function:	`findNode(…)`
Added code:	`MyClass::isValidLookup(key);`

OverSim file:	`Kademlia.cc`
Function:	`findNode(…)`
Added code:	`MyClass::isValidLookup(key);`

E OverSim parameter files

Default.ini

```
// default.ini
[General]
preload-ned-files=*.ned @../../INET-20061020-OverSim-3/nedfiles.lst
@../nedfiles.lst
# For enabeling realworld connections, choose appropriate scheduler
# UdpOut does only work with SingleHostUnderlay!
# You have to select appropriate outDeviceType in SingleHost configuration
#scheduler-class = TunOutScheduler
#scheduler-class = UdpOutScheduler
# If a realworld connection is desired, debug-on-errors has to be disabled
debug-on-errors=false
#debug-on-errors=true
perform-gc=false
network=SimpleNetwork

[ExternalApp]
# If an external app should be connected to the simulation, set app-port to
the appropriate TCP Port
# Has to be "0" if no external app is used
app-port = 0
# If bigger than zero, accept only n simultaneous app connections
appConnectionLimit = 0

[Cmdenv]
express-mode = yes

[Tkenv]
bitmap-path=../Bitmaps

[Parameters]

**.transitionTime = 10

# --- Application settings ---

# ** include *.overlayTerminal, *.overlayBackboneRouter, *.singleHost

# KBRTestApp settings
**.tier1*.kbrTestApp.testMsgInterval=60
**.tier1*.kbrTestApp.msgHandleBufSize=8
**.tier1*.kbrTestApp.lookupNodeIds=true

# i3 settings
**.tier*.i3.triggerTimeToLive = 60  # expiration time for triggers
**.tier*.i3.serverPort = 3072
**.tier*.i3.debugOutput = 1

# i3 client settings
**.tier*.serverPort = 3072
**.tier*.clientPort = 3073
**.tier*.triggerRefreshTime = 40   # time between trigger refreshings
**.tier*.serverTimeout = 100       # time to wait until server considered
unreachable
**.tier*.bootstrapTime = 20        # time to wait until i3 bootstraps
**.tier*.initTime = 30             # time to wait after i3 bootstrap to init
app
**.tier*.cacheSize = 3             # size of gateway cache
**.tier*.idStoreTime = 600         # time before discarding a stored id-
address pair
**.tier*.sampleRefreshTime = 600   # time between sample refreshings
```

```
# GIASearchApp settings
**.tier1*.giaSearchApp.messageDelay=60
**.tier1*.giaSearchApp.randomNodes=true
**.tier1*.giaSearchApp.randomKeys=20
**.tier1*.giaSearchApp.maximumKeys=100
**.tier1*.giaSearchApp.minKeyProbability=0.1
**.tier1*.giaSearchApp.maxKeyProbability=0.15
**.tier1*.giaSearchApp.maxResponses=10
**.tier1*.giaSearchApp.routeMessages = true
**.tier1*.giaSearchApp.searchMessages = false

#DHT settings
**.tier1*.dht.numReplica=1
**.tier1*.dht.numGetRequests=1
**.tier1*.dht.ratioIdentical=0.5

# DHTTestApp settings  testIntervaldef=60
**.tier2*.dhtTestApp.testInterval=13091
**.tier2*.dhtTestApp.testTtl=5000
**.tier2*.dhtTestApp.putDelay=5000
**.tier2*.dhtTestApp.initDelay=0.2

# P2PNS settings
**.tier2*.p2pns.twoStageResolution=false

#Scribe
**.tier1*.scribe.childTimeout = 60    # seconds until a node assumes that a
particular child has failed
**.tier1*.scribe.parentTimeout = 6    # seconds until a node assumes that the
parent node has failed
**.tier1*.scribe.scribePort = 1025    # port scribe listens on for direct udp
messages

# Vast / SimMud / PubSumMMOG / SimpleGameClient
**.tier*.*.areaDimension = 100
**.tier*.*.numSubspaces = 5
**.tier*.*.playerTimeout = 10
**.tier*.*.maxMoveDelay = 1
**.AOIWidth = 15.0
**.movementRate=4 # in updates per second

#SimpleGameClient FIXME: Make it independent of tier...
**.tier*.simpleGameClient.movementGenerator = "randomRoaming"
**.tier*.simpleGameClient.movementSpeed=6.0 # in m/s
**.tier*.simpleGameClient.groupSize = 1 # clients >0 per group when
GroupRoaming is active
**.tier*.simpleGameClient.useScenery = false # use obstacles on playground ?
**.globalObserver.globalFunctions*.coordinator.seed = 0 # seed for obstacle
generation ?

# generic app settings
**.tier*.*.debugOutput=true
**.tier*.*.activeNetwInitPhase=true

# --- Overlay settings ---

# ** include *.overlayTerminal, *.overlayBackboneRouter, *.singleHost

# chord settings
**.overlay*.chord.joinRetry=2
**.overlay*.chord.joinDelay=10
**.overlay*.chord.stabilizeRetry=1
**.overlay*.chord.stabilizeDelay=120
**.overlay*.chord.fixfingersDelay=240
**.overlay*.chord.successorListSize=8
**.overlay*.chord.aggressiveJoinMode=true
**.overlay*.chord.extendedFingerTable=false
**.overlay*.chord.numFingerCandidates=3
```

```
**.overlay*.chord.proximityRouting=false
**.overlay*.chord.mergeOptimizationL1 = false
**.overlay*.chord.mergeOptimizationL2 = false
**.overlay*.chord.mergeOptimizationL3 = false
**.overlay*.chord.mergeOptimizationL4 = false

# kademlia settings
**.overlay*.kademlia.lookupRedundantNodes = 8 #8
**.overlay*.kademlia.lookupParallelPaths = 1
**.overlay*.kademlia.lookupParallelRpcs = 1
**.overlay*.kademlia.lookupMerge = true
**.overlay*.kademlia.routingType = "iterative"
**.overlay*.kademlia.minSiblingTableRefreshInterval = 1000
**.overlay*.kademlia.minBucketRefreshInterval = 200
**.overlay*.kademlia.maxStaleCount = 0
**.overlay*.kademlia.k = 8 #8
**.overlay*.kademlia.s = 8 #8
**.overlay*.kademlia.b = 1
**.overlay*.kademlia.pingNewSiblings = true
**.overlay*.kademlia.activePing = false
**.overlay*.kademlia.proximityRouting = false

# generic Lookup settings
**.overlay*.*.lookupRedundantNodes = 1
**.overlay*.*.recNumRedundantNodes = 3


# pastry settings
**.overlay*.pastry.bitsPerDigit=4
**.overlay*.pastry.numberOfLeaves=16
**.overlay*.pastry.numberOfNeighbors=16
**.overlay*.pastry.joinTimeout=20
**.overlay*.pastry.readyWait=5
**.overlay*.pastry.secondStageWait=2
**.overlay*.pastry.pingTimeout=2.0
**.overlay*.pastry.pingRetries=1
**.overlay*.pastry.repairTimeout=60
**.overlay*.pastry.ringCheckInterval=60
**.overlay*.pastry.sendStateWaitAmount=.0001
**.overlay*.pastry.enableNewLeafs=true
**.overlay*.pastry.optimizeLookup=false
**.overlay*.pastry.optimisticForward=true
**.overlay*.pastry.avoidDuplicates=true
**.overlay*.pastry.partialJoinPath=false
**.overlay*.pastry.useRegularNextHop=true
**.overlay*.pastry.alwaysSendUpdate=false
**.overlay*.pastry.coordBasedRouting=false
**.overlay*.pastry.numCoordDigits=4
**.overlay*.pastry.CBRstartAtDigit = 0
**.overlay*.pastry.CBRstopAtDigit = 160
**.overlay*.pastry.useSecondStage=true;
**.overlay*.pastry.useDiscovery=false
**.overlay*.pastry.periodicMaintenance=false
**.overlay*.pastry.discoveryTimeoutAmount=.4
**.overlay*.pastry.repairTaskTimeoutAmount=500
**.overlay*.pastry.sendStateAtLeafsetRepair=true
**.overlay*.pastry.overrideOldPastry=false
**.overlay*.pastry.overrideNewPastry=false
**.overlay*.pastry.routeMsgAcks=true
**.overlay*.pastry.routingType="semi-recursive"

# bamboo settings
**.overlay*.bamboo.bitsPerDigit=4
**.overlay*.bamboo.numberOfLeaves=8
**.overlay*.bamboo.numberOfNeighbors=16
**.overlay*.bamboo.joinTimeout=20
**.overlay*.bamboo.readyWait=5
**.overlay*.bamboo.pingTimeout=2.0
```

```
**.overlay*.bamboo.pingRetries=1
**.overlay*.bamboo.repairTimeout=60
**.overlay*.bamboo.ringCheckInterval=60
**.overlay*.bamboo.sendStateWaitAmount=.0001
**.overlay*.bamboo.enableNewLeafs=true
**.overlay*.bamboo.optimizeLookup=false
**.overlay*.bamboo.optimisticForward=true
**.overlay*.bamboo.avoidDuplicates=true
**.overlay*.bamboo.partialJoinPath=false
**.overlay*.bamboo.useRegularNextHop=true
**.overlay*.bamboo.alwaysSendUpdate=false
**.overlay*.bamboo.coordBasedRouting=false
**.overlay*.bamboo.numCoordDigits=4
**.overlay*.bamboo.CBRstartAtDigit = 0
**.overlay*.bamboo.CBRstopAtDigit = 160
**.overlay*.bamboo.discoveryTimeoutAmount=.4
**.overlay*.bamboo.repairTaskTimeoutAmount=10
**.overlay*.bamboo.leafsetMaintenanceTimeoutAmount=4
**.overlay*.bamboo.globalTuningTimeoutAmount=20
**.overlay*.bamboo.routeMsgAcks=true
**.overlay*.bamboo.routingType="semi-recursive"

# koorde settings
**.overlay*.koorde.stabilizeRetry=1
**.overlay*.koorde.stabilizeDelay=20
**.overlay*.koorde.deBruijnDelay=20
**.overlay*.koorde.successorListSize=8
**.overlay*.koorde.deBruijnListSize=8
**.overlay*.koorde.shiftingBits=3
**.overlay*.koorde.joinRetry=2
**.overlay*.koorde.joinDelay=10
**.overlay*.koorde.deBruijnDelay=60
**.overlay*.koorde.aggressiveJoinMode=true
**.overlay*.koorde.useOtherLookup=true
**.overlay*.koorde.useSucList=true
**.overlay*.koorde.fixfingersDelay=10 # should try to get rid of this
parameter
**.overlay*.koorde.extendedFingerTable=false # should try to get rid of this
parameter
**.overlay*.koorde.numFingerCandidates=3 # should try to get rid of this
parameter
**.overlay*.koorde.proximityRouting=false # should try to get rid of this
parameter
**.overlay*.koorde.mergeOptimizationL1 = false
**.overlay*.koorde.mergeOptimizationL2 = false
**.overlay*.koorde.mergeOptimizationL3 = false
**.overlay*.koorde.mergeOptimizationL4 = false

# broose settings
**.overlay*.broose.bucketSize=8
**.overlay*.broose.rBucketSize=8
**.overlay*.broose.userDist=0
**.overlay*.brooseShiftingBits=2 # ugly: parameter of the compound module due
to NED limitations
**.overlay*.broose.parallelRequests=1
**.overlay*.broose.joinDelay=10
**.overlay*.broose.pingDelay=80
**.overlay*.broose.refreshTime=180
**.overlay*.broose.numberRetries=1

# gia settings
**.overlay*.gia.maxNeighbors = 50
**.overlay*.gia.minNeighbors = 10
**.overlay*.gia.maxTopAdaptionInterval = 120
**.overlay*.gia.topAdaptionAggressiveness = 256
**.overlay*.gia.maxLevelOfSatisfaction = 1.00
**.overlay*.gia.updateDelay = 60
**.overlay*.gia.maxHopCount = 10 #???
```

```
**.overlay*.gia.messageTimeout = 180
**.overlay*.gia.sendTokenTimeout = 5
**.overlay*.gia.neighborTimeout = 250
**.overlay*.gia.tokenWaitTime = 5
**.overlay*.gia.keyListDelay = 100
**.overlay*.gia.outputNodeDetails = false
**.overlay*.gia.optimizeReversePath = 0

# PubSubMMOG
**.overlay*.*.joinDelay = 1
**.overlay*.*.parentTimeout = 2
**.overlay*.*.maxChildren = 10
**.overlay*.*.maxMoveDelay = 1

# Generic settings

**.overlay*.*.debugOutput=true
**.overlay*.*.hopCountMax= 50
#**.overlay*.*.recNumRedundantNodes = 4
**.overlay*.*.collectPerHopDelay=false
IPv4Network.*.overlay*.*.drawOverlayTopology=false
SingleHostNetwork.*.overlay*.*.drawOverlayTopology=false
**.overlay*.*.drawOverlayTopology=true
**.overlay*.*.useCommonAPIforward=false
**.overlay*.*.routingType="source-routing-recursive"  #"exhaustive-iterative
semi-recursive full-recursive source-routing-recursive"
**.overlay*.*.keyLength=160
**.overlay*.*.joinOnApplicationRequest=false
**.overlay.*.localPort = 1024

# SimpleMultiOverlayHost settings
**.numOverlayModulesPerNode = 10
**.overlay[0].*.localPort=1024
**.overlay[1].*.localPort=1025
**.overlay[2].*.localPort=1026
**.overlay[3].*.localPort=1027
**.overlay[4].*.localPort=1028
**.overlay[5].*.localPort=1029
**.overlay[6].*.localPort=1030
**.overlay[7].*.localPort=1031
**.overlay[8].*.localPort=1032
**.overlay[9].*.localPort=1033

# general overlay lookup settings
#**.overlay*.*.lookupRedundantNodes = 4  ## SIIRRETTY
**.overlay*.*.lookupParallelPaths = 1
**.overlay*.*.lookupParallelRpcs = 1
**.overlay*.*.lookupSecure = false
**.overlay*.*.lookupMerge = false
**.overlay*.*.lookupUseAllParallelResponses = false
**.overlay*.*.lookupStrictParallelRpcs = false
**.overlay*.*.lookupNewRpcOnEveryTimeout = false
**.overlay*.*.lookupNewRpcOnEveryResponse = false
**.overlay*.*.lookupFinishOnFirstUnchanged = false
**.overlay*.*.lookupFailedNodeRpcs = false
**.overlay*.*.routeMsgAcks = false
**.overlay*.*.useCommonAPIforward=false

# neighbor cache settings
**.neighborCache.enableNeighborCache = false
**.neighborCache.rttExpirationTime = 100
**.neighborCache.maxSize = 50


# ---- UnderlayConfigurator settings ----

# UnderlayConfigurator module settings
*.underlayConfigurator.transitionTime = 0
```

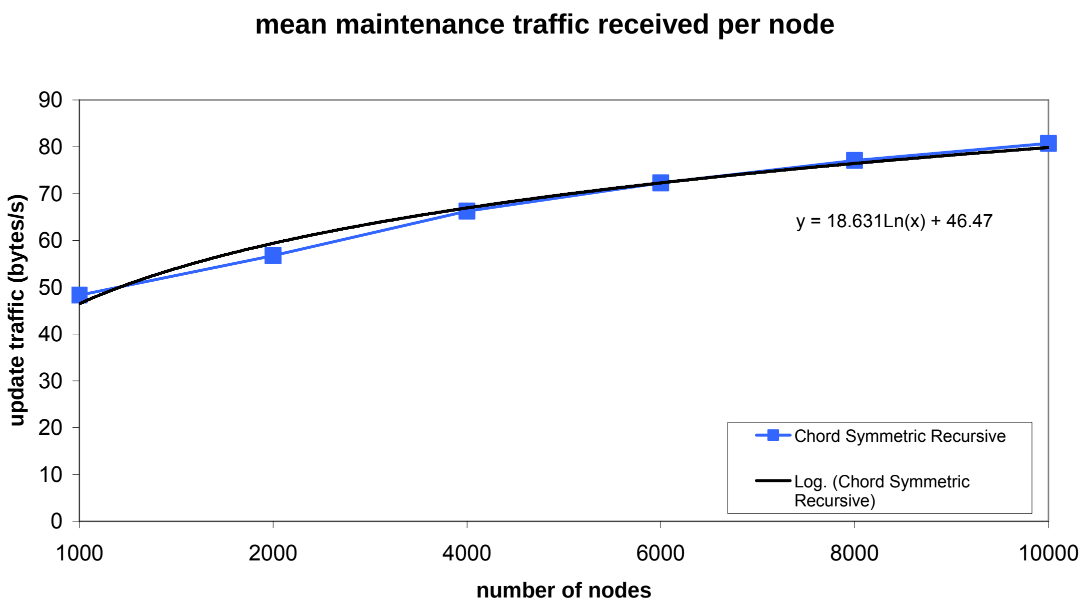
```
*.underlayConfigurator.measurementTime = -1
*.underlayConfigurator.gracefulLeaveDelay=15
*.underlayConfigurator.gracefulLeaveProbability=0.5
# disable churn for SingleHost networks
SingleHostNetwork.underlayConfigurator.churnGeneratorTypes = ""
# any combination of "NoChurn", "LifetimeChurn", "ParetoChurn" and
"RandomChurn" separated by spaces
*.underlayConfigurator.churnGeneratorTypes = "LifetimeChurn"

# ChurnGenerator configuration
*.churnGenerator*.initPhaseCreationInterval = 1
*.churnGenerator*.targetOverlayTerminalNum = 10
*.churnGenerator*.lifetimeMean = 20000.0
*.churnGenerator*.deadtimeMean = 10000.0
*.churnGenerator*.lifetimeDistName = "weibull"
*.churnGenerator*.lifetimeDistPar1 = 1.0

# RandomChurn (obsolete)
*.churnGenerator*.targetMobilityDelay=300
*.churnGenerator*.targetMobilityDelay2=20
*.churnGenerator*.churnChangeInterval=0
*.churnGenerator*.creationProbability=0.5
*.churnGenerator*.migrationProbability=0.0
*.churnGenerator*.removalProbability=0.5

# Use following channels in access networks
# see Common/channels.ned for allowed channels
# Here ** include *.underlayConfigurator, *.churnGenerator* and
*.globalObserver.globalTraceManager
**.channelTypes="ethernetline dsl"

# use globalFunctions?
*.globalObserver.globalFunctionsType = ""
*.globalObserver.useGlobalFunctions = 0

# global statistics
*.globalObserver.globalStatistics.outputMinMax = false
*.globalObserver.globalStatistics.outputStdDev = false
*.globalObserver.globalStatistics.globalStatTimerInterval = 20
*.globalObserver.globalStatistics.measureNetwInitPhase = false

# bootstrap oracle settings
*.globalObserver.bootstrapOracle.maxNumberOfKeys = 100
*.globalObserver.bootstrapOracle.keyProbability = 0.1
*.globalObserver.bootstrapOracle.maliciousNodeProbability = 0.0
*.globalObserver.bootstrapOracle.maliciousNodeChange = false
*.globalObserver.bootstrapOracle.maliciousNodeChangeStartTime = 200
*.globalObserver.bootstrapOracle.maliciousNodeChangeRate = 0.05
*.globalObserver.bootstrapOracle.maliciousNodeChangeInterval = 100
*.globalObserver.bootstrapOracle.maliciousNodeChangeStartValue = 0
*.globalObserver.bootstrapOracle.maliciousNodeChangeStopValue = 0.5

# globalObserver configuration
*.globalObserver.globalTraceManager.traceFile = ""
*.globalObserver.globalParameters.rpcUdpTimeout = 1.0
*.globalObserver.globalParameters.rpcKeyTimeout = 5.0
*.globalObserver.globalParameters.printStateToStdOut = false
*.globalObserver.globalParameters.topologyAdaptation = false

# bootstrapList configuration
**.bootstrapList.debugOutput = true
**.bootstrapList.mergeOverlayPartitions = false
**.bootstrapList.maintainList = false

# SimpleNetwork configuration
SimpleNetwork.overlayTerminal.udp.constantDelay = 50ms
SimpleNetwork.overlayTerminal.udp.useCoordinateBasedDelay = false
SimpleNetwork.overlayTerminal.udp.jitter = 0.01
SimpleNetwork.underlayConfigurator.terminalTypes = "SimpleOverlayHost"
SimpleNetwork.underlayConfigurator.fieldSize = 150
SimpleNetwork.underlayConfigurator.sendQueueLength = 0
SimpleNetwork.underlayConfigurator.fixedNodePositions = false
SimpleNetwork.underlayConfigurator.nodeCoordinateSource =
"dummy_that_doesnt_exist"
#SimpleNetwork.underlayConfigurator.nodeCoordinateSource = "nodes_2d.xml"
#SimpleNetwork.underlayConfigurator.nodeCoordinateSource = "nodes_3d.xml"

# SingleHostNetwork configuration
SingleHostNetwork.underlayConfigurator.terminalTypes = "dummy"
SingleHostNetwork.underlayConfigurator.nodeIP = ""
SingleHostNetwork.underlayConfigurator.nodeInterface = ""
SingleHostNetwork.underlayConfigurator.stunServer = ""
SingleHostNetwork.underlayConfigurator.bootstrapIP=""
SingleHostNetwork.underlayConfigurator.bootstrapPort=1024
SingleHostNetwork.zeroConfigurator.enableZeroconf = false
SingleHostNetwork.zeroConfigurator.serviceType = "_p2pbootstrap._udp"
SingleHostNetwork.zeroConfigurator.serviceName = "OverSim"
SingleHostNetwork.zeroConfigurator.overlayName = "overlay.net"
SingleHostNetwork.singleHost.outDeviceType = "UdpOutDevice"

# IPv4Network configuration
IPv4Network.outRouter*.outDeviceType = "TunOutDevice"
**.mtu = 65000
**.parser="GenericPacketParser"
**.appParser="GenericPacketParser"
**.gatewayIP = ""

# IPv4 backbone configuration
IPv4Network.underlayConfigurator.terminalTypes = "OverlayHost"
IPv4Network.backboneRouterNum=1
IPv4Network.overlayBackboneRouterNum=0
IPv4Network.accessRouterNum=2
IPv4Network.overlayAccessRouterNum=0
IPv4Network.connectivity=0.8
IPv4Network.underlayConfigurator.startIP="1.1.0.1"
IPv4Network.outRouterNum=0

# default overlay and application
# Here ** include *.globalObserver.globalTraceManager and *.churnGenerator*
**.overlayType = "ChordModules"
**.tier1Type = "KBRTestAppModules"
**.tier2Type = "TierDummy"
**.tier3Type = "TierDummy"
**.numTiers = 1

# default INET parameters

# ip settings
**.ip.procDelay=10us
**.routingFile=""
**.IPForward=true

# ARP configuration
**.arp.retryTimeout = 1
**.arp.retryCount = 3
**.arp.cacheTimeout = 100
**.networkLayer.proxyARP = true  # Host's is hardwired "false"

# NIC configuration
**.ppp[*].queueType = "DropTailQueue"
**.ppp[*].queue.frameCapacity = 10
```

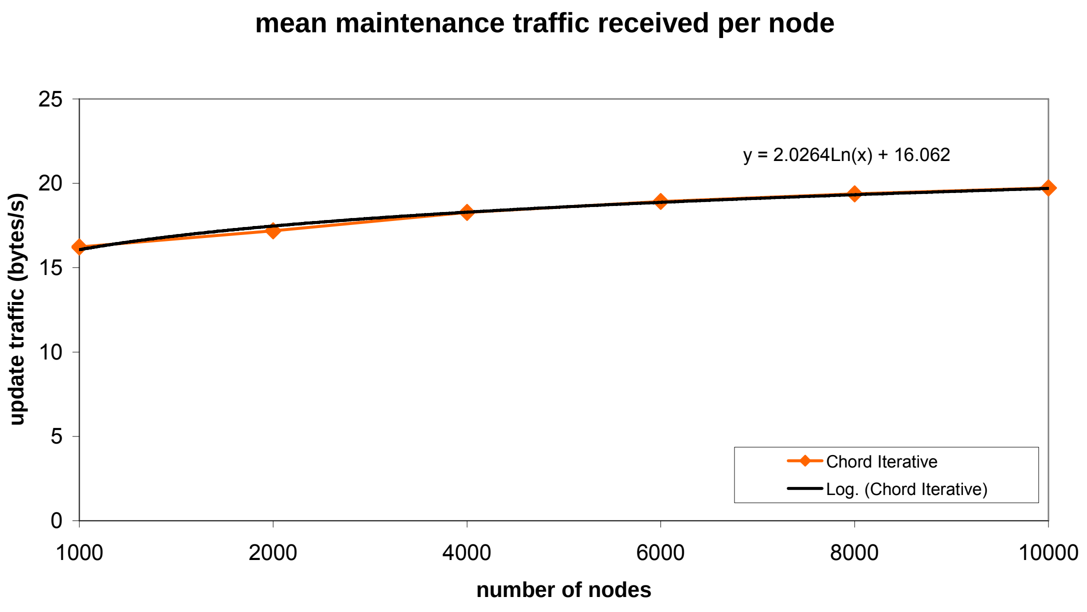
Omnetpp.ini

```
//omnetpp.ini
include ./default.ini
[Cmdenv]
//express-mode = false
status-frequency = 10000000

[Tkenv]
//**.overlay*.chord.ev-output = true
//**.ev-output = false


[Run 1]
description = "Kademlia (ITERATIVE)"
network = SimpleNetwork
sim-time-limit = 60000s
#"exhaustive-iterative semi-recursive full-recursive source-routing-recursive"
**.overlay*.*.routingType="iterative"
*.churnGenerator*.lifetimeMean = 69120.0
*.churnGenerator*.deadtimeMean = 17280.0
**.globalObserver.globalFunctionsType = "GlobalDhtTestMap"
**.globalObserver.useGlobalFunctions = 1
**.overlayType = "KademliaModules"
**.tier1Type = "DHTModules"
**.tier2Type = "DHTTestAppModules"
**.targetOverlayTerminalNum = 400
**.tier2*.dhtTestApp.testInterval=13091 ## MEAN for FETCH MESSAGES
**.tier2*.dhtTestApp.putDelay=14400 ## UPDATE TIMER
## TTL TIMER = 3*UPDATE TIMER -> DHTTestApp.cc
*.churnGenerator*.initTime = 1000
**.tier2*.dhtTestApp.initDelay=1.0
include params.ini

[Run 2]
description = "Chord (ITERATIVE)"
network = SimpleNetwork
sim-time-limit = 1209600s
#"exhaustive-iterative semi-recursive full-recursive source-routing-recursive"
**.overlay*.*.routingType="iterative"
*.churnGenerator*.lifetimeMean = 69120.0
*.churnGenerator*.deadtimeMean = 17280.0
**.globalObserver.globalFunctionsType = "GlobalDhtTestMap"
**.globalObserver.useGlobalFunctions = 1
**.overlayType = "ChordModules"
**.tier1Type = "DHTModules"
**.tier2Type = "DHTTestAppModules"
**.targetOverlayTerminalNum = 4000
**.tier2*.dhtTestApp.testInterval=13091 ## MEAN for FETCH MESSAGES
**.tier2*.dhtTestApp.putDelay=1800 ## UPDATE TIMER
## TTL TIMER = 3*UPDATE TIMER -> DHTTestApp.cc
**.tier2*.dhtTestApp.initDelay=1.0
*.churnGenerator*.initTime = 47700
#include params.ini

[Run 3]
description = "Chord (RECURSIVE)"
network = SimpleNetwork
sim-time-limit = 1209600s
#"exhaustive-iterative semi-recursive full-recursive source-routing-recursive"
**.overlay*.*.routingType="source-routing-recursive"
*.churnGenerator*.lifetimeMean = 69120.0
*.churnGenerator*.deadtimeMean = 17280.0
**.globalObserver.globalFunctionsType = "GlobalDhtTestMap"
**.globalObserver.useGlobalFunctions = 1
**.overlayType = "ChordModules"
**.tier1Type = "DHTModules"
```

```
**.tier2Type = "DHTTestAppModules"
**.targetOverlayTerminalNum = 50000
**.tier2*.dhtTestApp.testInterval=13091 ## MEAN for FETCH MESSAGES
**.tier2*.dhtTestApp.putDelay=14400 ## UPDATE TIMER
*.churnGenerator*.initTime = 47700
## TTL TIMER = 3*UPDATE TIMER -> DHTTestApp.cc
**.tier2*.dhtTestApp.initDelay=1.0
##include params.ini
```

F Trendline figures of maintenance traffic


mean maintenance traffic received per node
update traffic (bytes/s)
number of nodes
y = 18.631Ln(x) + 46.47
Chord Symmetric Recursive
Log. (Chord Symmetric Recursive)


mean maintenance traffic received per node
update traffic (bytes/s)
number of nodes
y = 2.0264Ln(x) + 16.062
Chord Iterative
Log. (Chord Iterative)