MASTER IN COMPUTER SCIENCE

P2PSIP: Towards a Decentralized, SIP-based VoIP System

Wauthy, Jean-Francois

Award date:
2007

Facultés Universitaires Notre-Dame de la Paix de Namur
Institut d'Informatique

P2PSIP: towards a decentralized, SIP-based VoIP system

Jean-François Wauthy

Mémoire présenté en vue de l'obtention
du grade de Maître en Informatique

Année académique 2006-2007

Abstract

The rise of technologies built on Peer-to-Peer networks has allowed the decentralization of many systems like file sharing or lately, with Skype, voice communications. These developments brought the SIP community to consider decentralizing the famous Session Initiation Protocol. This master thesis explores the first required steps for setting up a distributed SIP protocol (P2PSIP) as well as the problems it rises. This document also presents some implementations of a distributed Proxy/Registrar capable of handling an important amount of requests per second. These implementations show that P2PSIP is a valid option for large, decentralized deployments.

Keywords: SIP, distribution, Distributed Hash Table

Résumé

L'émergence des technologies basées sur des réseaux Peer-to-Peer a permis la décentralisation de nombreux systèmes comme les échanges de fichiers ou dernièrement, avec Skype, les communications vocales. Ces développements ont conduit la communauté SIP à envisager de décentraliser le célèbre protocole d'initiation de session. Ce mémoire explore les premières étapes nécessaires à la mise en place d'un protocole SIP distribué (P2PSIP) ainsi que les problématiques que cela soulève. Ce document présente également plusieurs implémentations d'un prototype de Proxy/Registrar distribué capables de gérer une quantité importante de requêtes par seconde. Ces implémentations montrent que P2PSIP est une solution valide pour de larges déploiements décentralisés.

Mots clés : SIP, distribution, Table de hachage distribuée

Acknowledgments

This master thesis could not have been written without the help of various people. I would like to thank the Alcatel-Lucent A5350 team to have cordially received me and to have considered me more like a colleague than an intern during my stay. In particular, I would like to thank Thomas Froment, my internship supervisor, for his precious advices, Dimitri Tombroff for guiding me during my work on the Session Registry component, Pierre De Rop for his availableness, Yannick Bourget, Eric Colaviti and Marius Lazar for having me felt as a member of the team.

I also want to acknowledge my thesis supervisors: Professor Laurent Schumacher for his feedback as well as his many corrections and suggestions and Professor Vincent Englebert for his support.

Finally, none of this could have been possible without my family and their continuous support during my complicated studies course.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The fast and still growing success of Skype showed that an (almost) fully decentralized VoIP system is technically implementable and economically viable. That brought the SIP/IETF community to wonder if they can come up with a similar successful open standard for Peer-to-Peer Internet Telephony based on SIP and by the way also decentralize standard SIP. A new working group that will deal with that project is being created and has been called P2PSIP.

The most widely used system for information distribution in a P2P environment is definitely a Distributed Hash Table (DHT). Well-known and widely used P2P applications like eMule, BitTorrent, JXTA or the Coral Content Distribution Network already use DHTs to distribute information among the participants. The P2PSIP working group, in its charter ([1]), chose to use a particular DHT algorithm while allowing additional optional algorithms in the future. For these reasons, studying and working on the P2P-SIP topic requires a proper understanding of DHTs and their internal mechanisms. The first part of this report focuses on the widely used DHTs and how they work. Then it explores the security issues of DHTs and how to possibly address them.

After having detailed the ideas discussed in the P2PSIP working group and latest publications, the problem of service discovery is discussed and a solution using a Prefix Hash Tree (PHT) is presented.

Two study cases implementations realized during the internship are then presented as well as the results of test evaluating their performances. Both implementations were designed to run on Alcatel-Lucent A5350 platform. The first implementation resulted from the design of a distributed Proxy/Registrar. In a second stage, the DHT approach was applied on the inter-cluster system of A5350 platform in order to reduce its dependency on a centralized component of the platform and to avoid blocking all the applications when restarting it.

Chapter 2

P2P-SIP

P2P-SIP is the name of the Internet Engineering Task Force (IETF) working group (WG) in charge of the standardization of a decentralized SIP architecture. The motivation for this decentralization comes from the will of IETF to avoid centralized architecture such as in SIP as currently deployed (also known as Client/Server SIP, C/S SIP, or simply SIP) and to create a Skype-like system, but with open standards.

2.1 Goals

As previously stated, the main goal of the P2P-SIP working group is get rid of the centralized servers of the common SIP architecture such as the Proxy and Registrar servers.

The charter of the IETF P2PSIP working group [1] determines the group's primary tasks as:

- producing an overview document explaining concepts, terminology, rationale, and illustrative use cases for the remaining work
- writing a proposition of standard for the P2PSIP Peer protocol (the protocol used between P2PSIP Peers)
- writing a proposition of standard for the P2PSIP Client Protocol (the protocol used between a P2PSIP Peer and a P2PSIP Client). The notions of "Peer" and "Client" will be detailed in the following section.
- writing an applicability statement that will address how the previously defined protocols, along with existing IETF protocols, can be used to produce systems to locate a user, identify appropriate resources to facilitate communications (such as media relays), and establish communications between the users, without relying on centralized servers

The ultimate goal of the working group is to standardize a service like the one Skype provides, that is to say a service capable of working in any connectivity conditions and with a good scalability.

2.2 Distribution model

The purpose of P2PSIP is to get rid of the central servers by distributing their functionalities among the participating nodes but different ways of distribution depending on which kind of nodes participate to the distribution exist. In this section, the most common distribution models are presented in order to introduce the one that will be used by P2PSIP.

2.2.1 Common distribution models

The most common distribution models are the following:

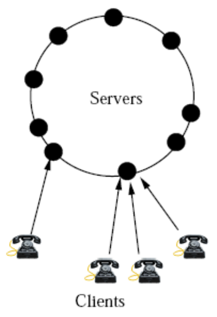


Figure 2.1: Only servers in DHT

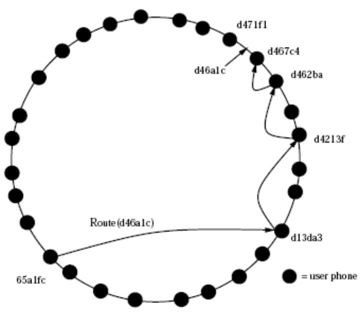


Figure 2.2: All nodes in DHT

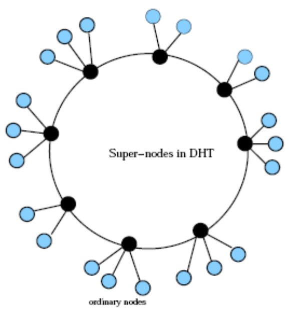


Figure 2.3: Only nodes fulfilling minimal requirements in DHT

- Only the servers participate to the distribution, this design has the advantage that it does not require changes to the clients and thus the distribution is transparent to the clients. This model is illustrated in Fig. 2.1.
- Fig. 2.2 shows the model where all the nodes participating to the system are part of the distribution framework. This model maximizes the number of nodes in the distribution mechanism but requires changes to the clients too.
- Not all nodes in the system have equivalent capabilities, some might have limited CPU performance or mobile devices can have changing connectivity conditions. The third model only uses the nodes with sufficient capabilities in the information distribution system, others use them as relay to query the system. The nodes taking part to the distribution process are also known as super-nodes. Fig. 2.3 illustrates that model.

2.2.2 Distribution model in P2PSIP

The P2PSIP charter [1] introduces a P2PSIP Peer Protocol and an optional[1] P2PSIP Client Protocol (see Section 2.3). The first protocol is used by the peers participating to the P2PSIP overlay, the second one is used by clients being outside the overlay in order to interact with the overlay via any peer. The distinction between peers participating to the overlay and clients that need to interact with a P2PSIP peer clearly hints that the P2PSIP distribution model is the

[1]optional meaning that its specification could be postponed

third one, where the nodes participating to the overlay are also known as super-nodes (like in the Kazaa P2P network).

There are different reasons why a node does not participate to the overlay like resource limitations (CPU, RAM, ...) or limited connectivity (NAT, firewall, ...). The connection to a P2PSIP behind a NAT is the main motivation for using the "super-node" distribution model but in the future, this model could ease the deployments on handheld devices with limited resources.

2.3 Main concepts

The main concepts used in P2PSIP are defined in the, so-called, "concepts draft" [2]. At the time of writing this thesis, some of them are still under discussion and are subject to change. This section describes the most important of these concepts.

P2PSIP

P2PSIP is the set of protocols that extends the Session Initiation Protocol (SIP) [3] using peer-to-peer techniques in order to resolve the targets of SIP requests. Currently, it only includes two protocols: the P2PSIP Peer Protocol used between P2PSIP Peers, and the P2PSIP Client Protocol used between a P2PSIP Client and a P2PSIP Peer.

P2PSIP Overlay

The P2PSIP overlay is the network of nodes that participates to the data distribution and that provides SIP registration, SIP request routing, and other services. The basic set of services that a P2PSIP overlay should provide is currently under discussion.

P2PSIP Peer

A P2PSIP Peer is a node participating in a P2PSIP overlay that provides storage and routing services to other nodes in the same P2PSIP overlay. A P2PSIP peer can be located behind NATs and still be fully functional. It can perform several operations like: joining and leaving the overlay, routing requests within the overlay, storing information, inserting information into the overlay and retrieving information from the overlay.

P2PSIP Client

A P2PSIP Client is a node participating in a P2PSIP overlay that provides neither routing nor route storage nor retrieval functions to that P2PSIP overlay. When a P2PSIP client wants to interact with a given P2PSIP overlay, it has to go through a P2PSIP Peer of that same overlay. Finally two characteristics of a P2PSIP client are that everything it can do, a P2PSIP peer can do it as well and that it is not necessarily a SIP User Agent Client (UAC).

P2PSIP Peer Enrollment

The enrollment is the process a P2PSIP peer follows to get an identifier and credentials for a given P2PSIP overlay. The process is done outside the P2PSIP overlay and is only needed at regular intervals or when the P2PSIP peer looses its identifier or its credentials.

2.4 On-going work

To reach the fixed objectives (see Section 2.1), much work has still to be done, especially on NAT traversal, service discovery, security issues and peer enrollment.

NAT traversal

NATs became popular because they introduced an easy solution to the shortage of IPv4 addresses. However they also introduce two kinds of problem in a P2PSIP environment.

First, NATs already introduce some issues with standard SIP itself, these problems are currently addressed by IETF. The STUN/TURN/ICE [4, 5, 6] suite of protocols and the SIP-Outbound mechanism [7] are developed to solve these issues. The P2PSIP WG will have to figure out a way to incorporate them into a P2PSIP system.

The second kind of problem is those specific to P2PSIP. The presence of NATs in P2PSIP raises the questions of how to send P2PSIP Peer Protocol or P2PSIP Client Protocol messages between two peers or clients which may be behind different NATs and of how to find STUN/TURN servers in a decentralized system. This issue has been postponed by the P2PSIP WG since the Charter now defines the P2PSIP Client Protocol as optional. It is therefore no longer a priority currently.

Searching for services

The need for a mechanism allowing localization of services in a decentralized system has already been introduced quickly in the presentation of the NAT traversal issues. However, the need for a localization mechanism is not limited to STUN/TURN servers. It could also be used for Presence applications or any other service (even some that do not exist currently). A proposition of such system is presented in Chapter 5.

Security

As more detailed in Section 3.10, DHTs face security issues. The current solution discussed by the P2PSIP working group is to use a central certification authority that would issue credentials for a peer at enrollment. These credentials would expire after a given time hence requiring the enrollment process to be done again. The peer would present these credentials when joining the system and eventually use them when discussing with other peers. This system is the best compromise between decentralization and security. The fact that the part of the system that will issue credentials will be separated from the P2PSIP overlay itself allows use cases when the party issuing credentials is unavailable or in disconnected scenarios.

This system solves the authentication problem and can harden the P2PSIP network against Sybil attacks (see Section 3.10) as long as the node is not hijacked by an external attacker.

Enrollment

As previously discussed, enrollment is tied to the security of the system. Finally, after some discussions about whether the working group should determine the format of the credentials or standardize the method of obtaining credentials, the charter states that, for starters, the existence of some enrollment process that provides a unique user name, credentials, and an initial set of bootstrap nodes if that is required by the protocols is assumed.

2.4.1 Other problems to be addressed in the future

Emergency calls

Enforcing preemption of emergency calls within a decentralized P2PSIP network will require some work in the future.

Legal interception

Given the fact that the overlay is not operated by any form of authority (except for the enrollment but it is a mechanism running separately from the overlay), the legal interception of traffic exchanged on a P2PSIP network will also be very difficult to enforce.

Spam

There is no mechanism or discussion about designing the system to avoid spamming. However, P2PSIP will probably reuse and adapt the work done on the subject in other working groups.

Chapter 3

Distributed Hash Tables

Distributed Hash Tables (DHTs) use a network overlay to partition a set of keys among participating nodes, and can efficiently route messages to the unique owner of any given key. Following the success of the first Peer-to-Peer (P2P) networks and the acknowledgment of their limitations such as centralized directory (Napster) or flooding queries (Gnutella), interests for DHTs rose.

Although their name refers to the hash table concept, not every DHT provides the same functionality as a traditional hash table. Some only provide a decentralized directory service informing the application of the localization of a key. The most commonly used DHTs are Chord, Pastry, CAN, Tapestry, Kademlia and Bamboo. They will be described in the following sections.

3.1 Chord

The Chord specifications [8] only support the mapping of a key onto a node, storing a value associated with the key is left to the application layer. It also guarantees that queries make a logarithmic number of hops.

The consistent hashing assigns each node and key a m-bit identifier using a base hash function such as SHA-1 (a node's identifier is, for example, chosen by hashing its IP, while a key identifier is produced by hashing the key itself). Identifiers are ordered in an identifier circle modulo 2^m. Key k is assigned to the first node whose identifier is equal or follows k in the identifier space. This node is called the *successor node* of key k.

Each Chord node only maintains a small amount of "routing" information about other nodes. A node knows and maintains information about:

- its *successor* on the circle
- a *finger table* whose i^{th} entry (where $1 \leq i \leq m$) contains the identity of the first node that succeeds[1] n by at least 2^{i-1}. Note that the first entry of the finger table of a node is

[1]all arithmetic operations being modulo 2^m

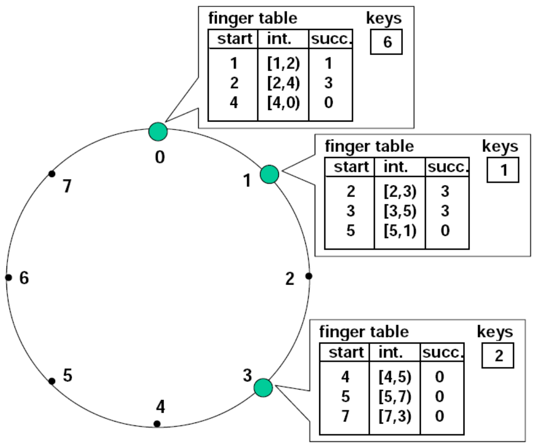


Figure 3.1: Chord ring. Nodes 0, 1 and 3 are active and store keys. They appear in each other's finger table as successor (succ.) for the interval (int.) starting at node (start).

its successor.

- a *predecessor* pointer which contains the Chord identifier and IP address of the immediate predecessor of the node; this pointer is used to simplify the join and leave mechanisms.

In a network with N nodes, each node maintains information only about $O(\log N)$ other nodes and a lookup is done in $O(\log N)$ hops. When a node joins or leaves the network, Chord must update the routing information. These operations require $O(\log^2 N)$ messages.

To preserve the Chord network integrity while nodes join and leave, it needs to preserve two invariants:

1. Each node's successor is correctly maintained.
2. For every key k, node $successor(k)$ is responsible for k.

When a node n joins the network, Chord must perform three tasks to preserve the invariants:

1. Initialize the predecessor and fingers of node n.
2. Update the fingers and predecessors of existing nodes to reflect the addition of n.
3. Notify the higher layer software so that it can transfer state (e.g. values) associated with keys that node n is now responsible for.

To deal with concurrent joins and leaves, Chord uses a "stabilization" protocol to keep nodes' successor pointers up to date. The stabilization scheme guarantees that adding nodes to the network preserves reachability of existing nodes. The stabilization process is run periodically by every node.

To ease failure recovery, each node keeps a "successor-list" of its nearest successors on the Chord ring. This successor-list is maintained by the stabilization process. It also helps higher layer software to replicate data.

Remark Chord itself does not provide any authentication, caching and replication mechanisms. The application layer is responsible for providing such mechanisms if needed.

3.2 Pastry

Each node in a Pastry [9] network has an unique 128-bit identifier (nodeId). The nodeIds have to be generated such that the resulting set of nodeIds is uniformly distributed in the nodeId space (this is usually done using a hashing function).

When presented with a message and a numeric key, a Pastry node efficiently routes the message to the node with a nodeId that is numerically closest to the key. The routing operation is expected to occur with maximum $O(\log N)$ hops. Each node also keeps track of its immediate neighbors in the nodeId space, and notifies application of new node arrivals, node failures and recoveries.

Under normal operation, Pastry can route a message to the numerically closest node to a given key in less than $\lceil \log_{2^b} N \rceil$ hops (where b is configuration parameter with typical value of 4). Delivery is guaranteed unless $\lfloor \frac{|L|}{2} \rfloor$ (where $|L|$ is a configuration parameter with a typical value of 16 or 32) nodes with adjacent nodeIds fail simultaneously.

Each node maintains:

- a *routing table* which contains $\lceil \log_{2^b} N \rceil$ rows with $2^b - 1$ entries each. The entries at row n refer to a node whose nodeId shares the present node's nodeId in the first n digits, but whose $(n+1)^{th}$ digit has one of the $2^b - 1$ possible values other than the $(n+1)^{th}$ digit in the present node's id. If no node is known with a suitable nodeId, then the routing table entry is left empty.
- a *neighborhood set* which contains the nodeIds and IP addresses of the $|M|$ nodes that are closest (according to the proximity metric) to the node. The neighborhood set is not normally used in routing messages. Typical value for $|M|$ is 2×2^b.
- a *leaf set* which is the set of nodes with the $\frac{|L|}{2}$ numerically closest larger nodeIds and the $\frac{|L|}{2}$ numerically closest smaller nodeIds. The leaf set is used during message routing. Typical value for $|L|$ is 2^b.

The routing protocol in Pastry is prefix-based and follows this procedure: given a message, the node first checks whether the key falls within the range of nodeIds covered by its leaf set. If so, the message is forwarded directly to the destination node, namely the node in the leaf set whose nodeId is closest to the key (possibly the present node). If the key is not covered by the leaf set, then the routing table is used and the message is forwarded to a node that shares a common prefix with the key by at least one more digit. In certain cases, it is possible that the appropriate entry in the routing table is empty or the associated node is not reachable, in which case the message is forwarded to a node that shares a prefix with the key at least as long as the local node, and is numerically closer to the key than the present node's id. Such a node must be in the leaf set unless the message has already arrived at the node with numerically closest nodeId. And, unless $\lfloor \frac{|L|}{2} \rfloor$ adjacent nodes in the leaf set have failed simultaneously, at least one of those nodes must be alive.

When a new node joins a Pastry network, it needs to initialize its state tables and to inform other nodes of its presence. The failure of a node is detected when a node attempts to contact the failed node and there is no response. Also, to keep the neighborhood set in a consistent state, each node tries to contact each member of the neighborhood set periodically to check whether it is alive. When a failed node is detected, the leaf set, routing table and neighborhood set containing references to the failed node have to be updated.

To handle concurrent node arrivals and departures, Pastry attaches a timestamp to each message providing state information. This way, the node receiving the message can immediately know if it has to adjust its own state based on the received information, and then eventually sends update messages with the same timestamp to the node which sent the state information; this way it can check if its state has since changed and react accordingly.

Pastry (unlike Chord) takes into account the locality aspect in its operations (mainly during routing). The network locality is a scalar value, such as the number of hops, the geographic distance, etc. between two nodes. The lower the distance between two nodes, the more desirable it will be considered by Pastry. The application running Pastry has to provide a function returning the network distance between two given nodes, allowing more flexibility in its computing but complicating the application.

3.3 Content Addressable Network (CAN)

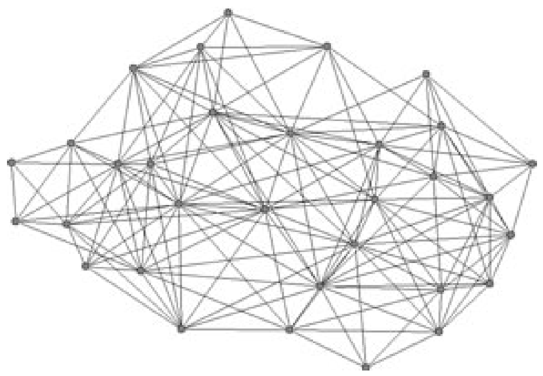


Figure 3.2: CAN geometry

A Content Addressable Network (CAN) [10] is built around a virtual multi-dimensional Cartesian coordinate space on a multi-torus. This d-dimensional coordinate space is completely logical. The entire coordinate space is dynamically partitioned among all the peers in the system such that every peer possesses its individual, distinct zone within the overall space. A CAN peer maintains a routing table that holds the IP address and virtual coordinate zone of each of its neighbor coordinates. A peer routes a message towards its destination using a simple greedy forwarding to the neighbor peer that is closest to the destination coordinates. A CAN requires an additional maintenance protocol to periodically remap the identifier space onto nodes.

3.4 Tapestry

In a Tapestry [11] network, the nodes are assigned identifiers (nodeIds) and application-specific endpoints are assigned globally unique identifiers (GUIDs). 160-bit values resulting from hashing algorithm like SHA-1 are currently used as identifiers. In order to enable different applications to coexist on top of a Tapestry network, every message contains the application-specific identifier it is related to.

To deliver messages, Tapestry uses longest prefix routing in a way similar to classless inter-domain routing (CIDR). To achieve that goal, each node maintains:

- a *neighbor list* consisting of the nodeIds and IP addresses with which the node communicates;
- a *neighbor map* with multiple levels, where each level contains links to nodes matching a prefix up to a digit position in the nodeId and also contains a number of entries equal to the nodeId's base. The primary ith entry in the jth level is the nodeId and location of the closest node that begins with $prefix(nodeId, j-1)+$"i". Its total size is $c \times \beta \times \log_\beta N$ where c is the number of neighbors, β is the base used for representing the nodeIds and N is the number of nodes in the network;
- a list of *backup links* for each entry in the neighbor maps;
- a list of *backpointers* containing references to others nodes that point at it.

There are four steps to perform when inserting a new node N into a Tapestry network:

1. Need-to-know nodes are notified of N, because N fills a null entry in their routing tables;
2. N might become the new object root for existing objects. References to those objects must be moved to N to maintain object availability;
3. The algorithms must construct a near optimal routing table for N;
4. Nodes near N are notified and may consider using N in their routing tables as an optimization.

When a server stores an object in the network, it periodically advertises this object by routing a publish message towards the root node of the published object. The root node is the node where the object is actually stored in the overlay. Each node along the publication path then stores a pointer mapping the GUID to the server. When there are replicas of an object on separate servers, each server publishes its copy. Tapestry nodes store location mapping for objects replicas in sorted order of network latency from themselves.

When a node N voluntarily leaves the network, it tells the set D of nodes in N's backpointers of its intention, along with a replacement node for each routing level from its own routing table. Each of the notified nodes sends an object republish message to both N and its replacement. Meanwhile, N routes references to locally rooted objects to their new roots and signals nodes in D when finished.

To deal with nodes unvoluntarily leaving the network (due to node and link failures or network partitions), Tapestry uses redundancy into its routing tables and object location references. To maintain availability and redundancy, nodes use periodic beacons to detect outgoing link and node failures. Such events trigger repair of the routing mesh and initiate redistribution and replication of object location references. Furthermore, the repair process is augmented by soft-state republishing of object references.

3.5 Kademlia

Kademlia [12] is designed to minimize the number of messages that nodes must send to learn about each other. Configuration information spreads automatically as side-effect of key lookups. Each node has a nodeId in a 160-bit key space. The $(key, value)$ pair is stored on the node whose ID is the closest to the key.

The original approach in Kademlia is the usage of a XOR metric (the bit-wise exclusive of their identifiers interpreted as an integer) to determine distance between points in the key space. XOR is symmetric, allowing Kademlia participants to receive lookup queries from precisely the same distribution of nodes contained in their routing tables. Systems that do not present that property do not learn useful routing information from queries they receive.

Each node keeps a list of the IP address, UDP port and nodeId for nodes of distance between 2^i and 2^{i+1} from itself for each i (where $0 \leq i < 160$). These lists are called k-buckets. Each k-bucket is kept sorted by time last seen (least-recently seen node at the head, most-recently seen at the tail). k is a system-wide parameter fixing the maximum number of nodes in the bucket. The k-buckets implement a least-recently seen eviction policy, except that live nodes are never removed from the list. By keeping the oldest live contacts around, k-buckets maximize the probability that the nodes they contain will remain online. A second benefit of k-buckets is that they provide resistance to certain DoS attacks. One cannot flush nodes' routing state by flooding the system with new nodes. Kademlia nodes will only insert the new nodes in the k-buckets when old nodes leave the system.

Node lookup consists in locating the k closest nodes to some given nodeId. Kademlia uses a recursive algorithm:

1. the lookup initiator picks α (α is a system-wide concurrency parameter) nodes from its closest non-empty k-bucket (if that bucket has fewer than α entries, it just takes the α closest nodes it knows of);

2. it sends parallel, asynchronous lookup requests to the α nodes it has chosen;

3. it sends once again requests to nodes it has learned about from previous lookup requests (this recursion can begin before all α previous requests have returned). Nodes that fail to

respond quickly are removed from consideration until and unless they do respond;

4. the lookup terminates when the initiator has queried and gotten responses from the k closest nodes it has seen.

To store a $(key, value)$ pair, a participant locates the k closest nodes to the key and sends them store requests. To ensure persistence, each node re-publishes the $(key, value)$ pairs that it has every hour. Generally, it also requires the original publishers of a $(key, value)$ pair to republish it every 24 hours. Otherwise, all $(key, value)$ pairs expire 24 hours after the original release. Finally, when a node observes a new node which is closer to some of his $(key, value)$ pairs, it replicates these pairs to the new node without removing them from its own database.

Remark When $\alpha = 1$, the lookup algorithm looks like Chord's in terms of message cost and the latency of detecting failed nodes.

3.6 Bamboo

Bamboo [13, 14] has been developed to handle high churn (the continuous process of node arrival and departure) rates and its geometry and routing algorithm are identical to Pastry; the difference lies in how Bamboo maintains the geometry as nodes join and leave the network and the network conditions vary. The motivation to use the Pastry geometry is due to its *static resilience* provided by the presence of the leaf set as shown in [15].

In contrast to Pastry, which uses the response from the root for the new node's identifier to perform a sophisticated join algorithm designed to maximize the proximity of a node's neighbors, in Bamboo, the new node does very little with the result. It simply sends an application-level ping to each node in the response, and if they respond to this ping, it adds them to its leaf set and routing table as appropriate (if they are its immediate predecessors or successors, or if they have the correct prefixes).

To maintain its leaf set over time, a Bamboo node pushes a list of the nodes in its own leaf set to some member of that set, and pulls a list of that neighbor's leaf set in response. In this way, the node learns of new nodes in its vicinity of the ring. This process can be performed periodically or in response to failures. In Pastry, nodes only push leaf sets.

As previously stated, Bamboo has been developed to handle churn. To reach that goal it supports reactive recovery, periodic recovery and can use different ways to determine timeouts (TCP-style, virtual coordinates). It also uses proximity neighbor selection (choosing among the potential neighbors for any given routing table entry according to their network latency).

3.7 Beehive

Beehive [16] is not a peer-to-peer distributed hash table but a pro-active replication framework that can, at best, provide constant lookup performance for Zipf-like query distribution [17].

It has to be deployed on top of a standard $O(\log N)$ peer-to-peer distributed hash table.

It uses a model-driven caching system which used with query distributions based on a power law can enable a DHT system to achieve a small constant lookup latency on average. That model-driven caching system can be opposed to demand-driven caching systems such as the one used in Kademlia.

It aims at the following goals:

- High performance: $O(1)$ average-case lookup performance, $O(\log N)$ worst-case lookup performance;
- High scalability: minimize background traffic, memory and disk space requirements;
- High adaptivity: adjust the performance of the system to the popularity distribution of objects.

Beehive assigns a *replication level* to each object. An object at level i is replicated on all nodes that have at least i matching prefixes with the object. Then Beehive's replication strategy finds the minimal replication level for each object such that the average lookup performance for the system is a constant C number of hops (where C is a system parameter that can be changed dynamically).

3.8 Comparison

Table 3.1 summarizes the main pros and cons of each of the DHT algorithms that were previously presented. In this table, the CAN algorithm is immediately spotted for its different lookup performance, and is therefore not interesting for a large deployment; CAN is also difficult to understand due to the complex geometry it is based on. Among the other implementations, the main differences reside in the way they handle routing, distribution and node failure or departure. Some of these DHTs have been conceived for a given use-case such as file sharing (Kademlia), high churn rates (Bamboo) or to bring new aspects into DHTs (network locality in Pastry and Tapestry).

3.9 Bootstrapping

Although the purpose of a DHT is to get a fully decentralized system, a joining node has to find a way to get in touch with at least one node of the overlay and this might often re-introduce some sort of centralization. The node joining the overlay is logically called the *joining node*, the node helping it to enter the overlay is the *admitting node* and the node contacted and that redirects the joining to the admitting node is known as the *bootstrap node*. The most widely used methods to get in touch with a bootstrap node are described in this section.

Algorithm	Lookup performance	Pros	Cons
Chord	$O(\log N)$	simplicity	rely on application for replication
Pastry	$O(\log N)$	network locality	parameters to tune
CAN	$O(dN^{\frac{1}{d}})$	[redacted]	complex underlying geometry
Tapestry	$O(\log N)$	network locality	complicated design
Kademlia	$O(\log N)$	XOR symmetric metric	no explicit key removal
Bamboo	$O(\log N)$	enhancing Pastry	features to choose

Table 3.1: Comparison of major DHT algorithms

3.9.1 Multicast address

If all the nodes willing to act as bootstrap node register to a multicast address, any joining node can contact one of them by sending a kind of ping request on the multicast address and start the bootstrapping process with the first one that responds. This approach has the advantage to preserve the decentralized characteristic of the system; however, multicast communications are not really available everywhere on the Internet.

3.9.2 DNS record

Some nodes providing the bootstrap functionality are registered in a well-known DNS (Domain Name Service) record that any joining node can use to resolve the IP addresses of these bootstrap nodes. However, this solution unfortunately reintroduces some sort of centralization in a decentralized system and requires some sort of operator. Furthermore, the time to propagate the updated DNS record can be prohibitive.

3.9.3 Cached nodes

If the joining node has already been connected to the overlay in the past, it can try to contact the nodes whose existence it was aware of when it was online. This way does not provide any guarantee of success but could save some time. It must however be used with another method as back up.

3.9.4 Well-known nodes

The addresses of some bootstrap nodes can be hardcoded in the joining node. This method is dangerous as nodes come and go sometimes quickly in a DHT. Or it requires that these nodes are almost all the time online and, most likely in this case, operated by some sort of provider.

3.9.5 DHCP

The bootstrap addresses could also be provided by the DHCP (Dynamic Host Configuration Protocol) server along with the other information it sends but the DHCP itself has to collect them somewhere, most likely using one of the previous methods.

3.9.6 Combined solution

Each of the previous solutions presents some drawbacks or has a limited application. The best solution seems to be a combination of them: using information provided by the DHCP server or using multicast when available, if not trying to use cached nodes, then a DNS record (if DNS access is available) and finally a hardcoded address.

3.10 Security in DHTs

This section concentrates on classic vulnerabilities of DHTs and quickly discuss how to prevent them. For more details on security in a P2PSIP context, refer to chapter 7.

3.10.1 DHT vulnerabilities

Authentication

DHT being decentralized and authentication not being really possible without some sort of centralized credentials server, authentication in a DHT without introducing some form of centralization is a real challenge.

Rogue nodes

Since the information stored in the DHT is distributed among the participating nodes, the node can either modify or delete on its own the information that it stores.

Sybil attack

The Sybil attack [18] consists in controlling a substantial fraction of the overlay by inserting a lot of faulty nodes operated by a single entity in the system. By controlling a large amount of nodes, the attacker can perform different kinds of attack.

Spoofing This type of attack becomes really easy when one control a large part of the overlay because the probability one will get a message whose destination is the targeted identity increases with the number of faulty nodes inserted into the system.

Man in the middle When one controls a large portion of the overlay, it can modify routing inside the overlay and therefore redirect normal nodes to faulty nodes serving modified information (this can happen with a few faulty nodes but has a lot more chances of success with many faulty nodes). Figures 3.3 and 3.4 illustrate a man in the middle attack with the two routing methods used in DHTs (iterative and recursive).

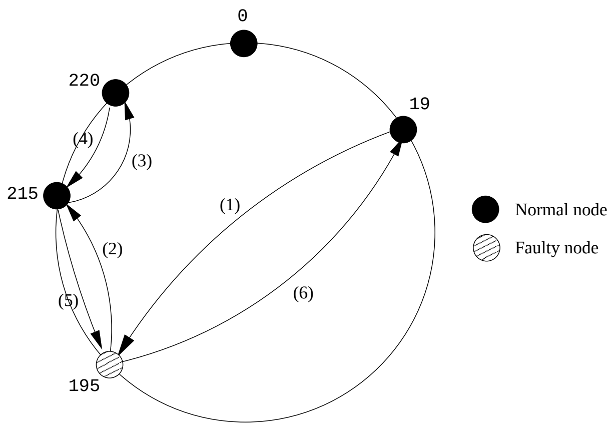


Figure 3.3: Man in the middle attack with recursive routing: (1) node 19 asks node 195 which node stores key 217; (2)→(5) node 195 normally process the query; (6) node 195 responds with the value it wants node 19 to use

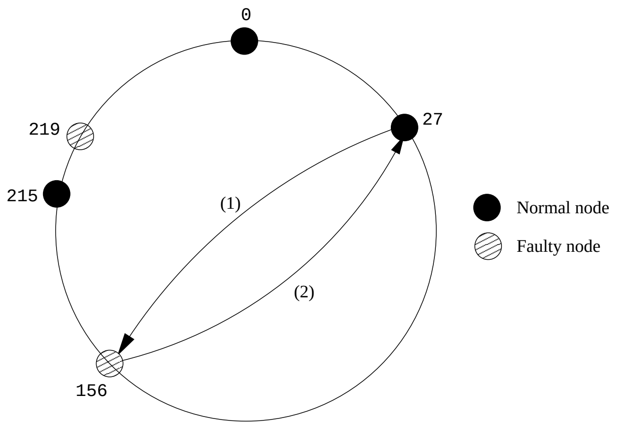


Figure 3.4: Man in the middle attack with iterative routing: (1) node 27 asks node 156 which node is more likely storing key 212; (2) node 156 answers that such node is node 219 (which is going to answer with its own value of key 212)

Denial of service By inserting a lot of successive faulty nodes into the system, the attacker can perform a Denial of Service (DoS) attack. These inserted nodes will fill all the pointers maintained by the node preceding them in the overlay cutting off the targeted nodes from the DHT.

3.10.2 Protections against attacks

Protection from rogue nodes

Encrypting exchanged data can guarantee their integrity but does not prevent a node to return an empty result or to report the data as missing from the DHT (e.g. during a lookup with recursive routing); fortunately replication techniques can add some protection against such behavior. Performing a check of the node during its enrollment within the overlay can increase the security or using a trust network for nodes known from the DHT but what if the node gets hacked or corrupted ?

Protecting from Sybil attacks

In [18], John R. Douceur states that *"without a logically centralized authority, Sybil attacks are always possible except under extreme and unrealistic assumptions of resource parity and coordination among entities"*. Protection against Sybil attacks in a fully decentralized DHT seems therefore impossible.

Chapter 4

Using a DHT inside the Session Registry

The Session Registry of the Alcatel-Lucent A5350 platform is the component providing access to data associated to a so called session across an application cluster. A basic use case of this component is when several Registrar SIP Servlets are deployed inside a common group on the platform (each of them may be running on different physical servers). A client willing to registrer itself will send a REGISTER request to the system, this request will be handled by one of the Registrar Servlets. Later, another client wants to communicate with the registered client and sends a query to the system. This query will be received by one of the Registrar Servlets, most likely not the one that handled the registration request. This Registrar will then use the Session Registry to recover the contact information previously stored during the registration process.

This chapter presents the implementation of a DHT directly inside the Session Registry in order to improve its reliability and to offer direct access to a DHT layer to applications running on the A5350 platform.

4.1 Current architecture

Currently, the central component of the system is the SessionRegistry. It is in charge of creating sessions, creating their replicate, destroying sessions, retrieving sessions and maintaining an index of all created sessions. When a session is created, it is stored on the agent that requested it, a replicate is also created on another random agent in the system.

Fig. 4.1 shows the current architecture. On this figure, the square is the SessionRegistry with its API in courier font (partially represented on the figure) and its index of sessions mapping session IDs to the agent storing that session and the agent storing the replicate. The agents are represented as circles with their name in the circle. Next to each agent, the sessions they store are listed; in bold for a master session and in italic for a replicate.

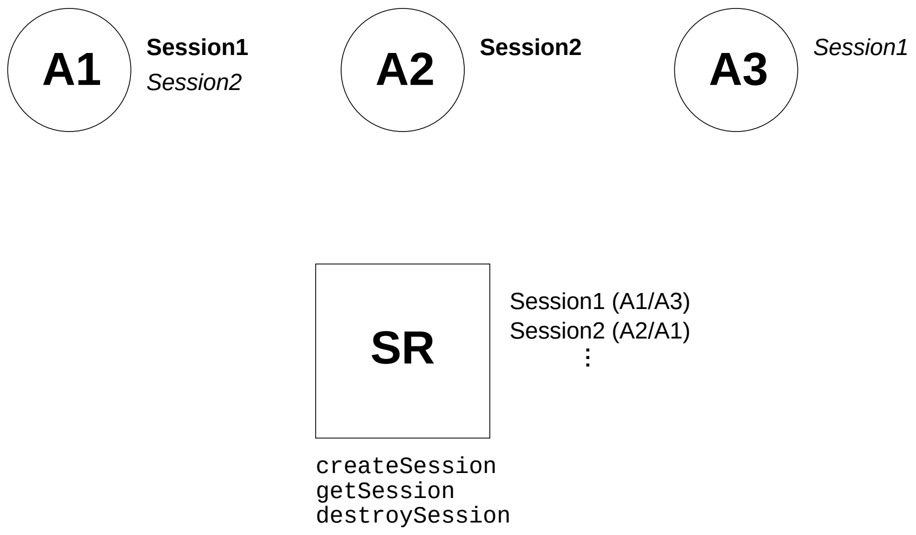


Figure 4.1: Current architecture of the Session Registry

4.2 Motivation

The current implementation of the Session Registry does not uniformly distribute the $(key, value)$ pairs among the agents and needs to refill the Registry when it is restarted. This means that one agent could store a lot more data than another and that the time to restart the Registry depends on the numbers of records it has to store; this process can therefore be very long.

The development of a DHT-based system can offer an easy work-around to these limitations. It is also an opportunity to offer direct access to a DHT interface into the SessionManager implementation that can be used by any application developed on the A5350 platform.

4.3 Design

The DHT design is based on Chord [8]. As described in Section 3.1, the agents are organized on a ring and each of them has an unique 160-bit identifier, obtained by hashing its name. Each agent's DHT layer knows about its successor and predecessor in the ring but, unlike in Chord, does not maintain any finger table.

4.3.1 Centralization

The DHT designed in this context is not decentralized because the Registry is the only component capable of providing a consistent list of all agents connected to the system at any time. Therefore, any new agent has to contact the Registry in order to insert itself in the DHT ring.

4.3.2 Data distribution

As in Chord, the $(key, value)$ pair is stored on the successor of key in the ring; the successor of a key is the closest agent whose identifier is greater or equal to the key. From an agent's point of view, an agent stores the values associated with keys strictly greater than its predecessor's identifier and lesser or equal to its own identifier.

4.3.3 Fault tolerance

Each $(key, value)$ pair stored on an agent is replicated on its successor on the DHT ring. When an agent failure is detected, the successor of the failing agent is informed of the change in the overlay topology and then knows that it is now in charge of a portion of the $(key, value)$ pairs stored in the replica it manages. The agent replicates these pairs to its own successor to maintain the replication level of the system.

This technique has a quite understandable limitation. If an agent replicates data on its n successors, it means that as long as maximum n agents fail simultaneously there is no data loss. Here the replication occurs only on the first successor, therefore only one agent can fail in order to avoid data loss.

4.3.4 Need of a `PutOrGet` primitive

In order to avoid creating the same key twice, the system has to provide a `PutOrGet` primitive that checks if a key already exists before creating or overwriting it. If the key already exists, the function returns the existing key otherwise it simply creates it.

4.4 New architecture

In the new architecture, the SessionRegistry can now provide functions to access the DHT from any application running on the platform. Also, the localization of the sessions is stored in the DHT itself and is therefore replicated. A representation of the new architecture is shown on Fig. 4.2 (it uses the same formalism as Fig. 4.1 with the session localization information in italic being replicates).

4.5 Implementation

4.5.1 Sequence diagrams

Joining the overlay

When a new agent joins the DHT ring, the following operations are performed (see Fig. 4.3):

1. The joining agent asks the Registry where it must insert itself in the ring;
2. The Registry answers with the admission information (predecessor and successor), will refuse any other admission request and will queue all `put`, `get` or `remove` operations until the admission process is either successful or aborted;
3. Once the joining agent has received its admission information, it checks if it knows about the provided successor and predecessor and if it can connect to them;

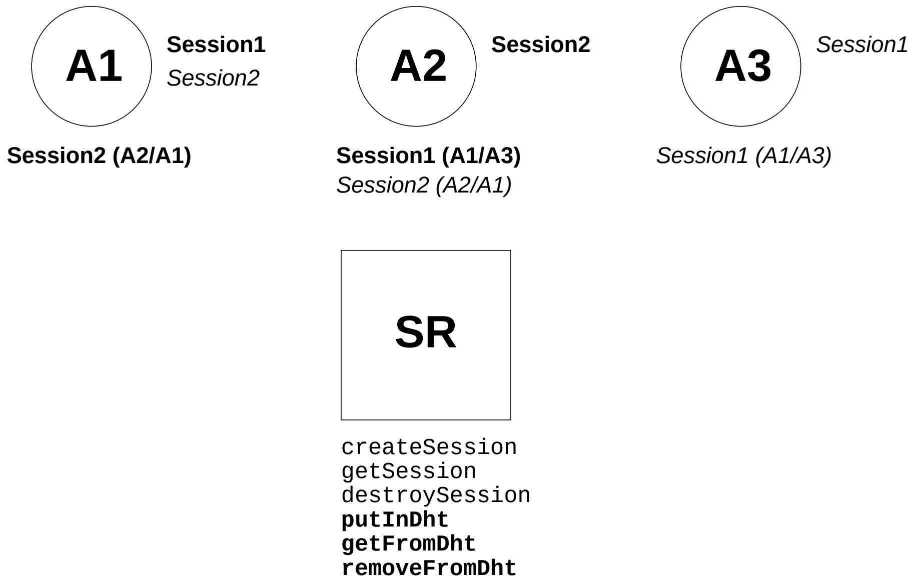


Figure 4.2: Architecture of the Session Registry with DHT support

4. The agent sends a `Join` message to its successor, the successor then handles the admission process;

5. The successor transfers the $(key, value)$ pairs that the joining agent will be managing after successful admission;

6. The joining agent confirms that it correctly received the data;

7. The successor lets the predecessor know that it has a new successor;

8. The predecessor checks that it knows about the joining agent and can connect to it;

9. The predecessor creates data replica on its soon-to-be successor;

10. The joining agent confirms the creation of the replica

11. The predecessor updates its successor pointer to the joining agent and tell its previous successor it is ready;

12. The successor migrates data it will not be handling anymore to its replica;

13. The successor removes data that will not be handled anymore by its predecessor from its replica[1];

14. The successor tells its own successor to clean its replica too;

15. The successor updates its predecessor pointer to the joining agent and sends confirmation that its part of the admission process has been completed to the joining agent;

[1]this step is not regrouped with the previous one because the *CleanReplica* is used separately in other sequences

16. The joining agent confirms to the Registry that the admission process has been successfully completed. The Registry now accepts new admission requests and processes the eventual message queue.

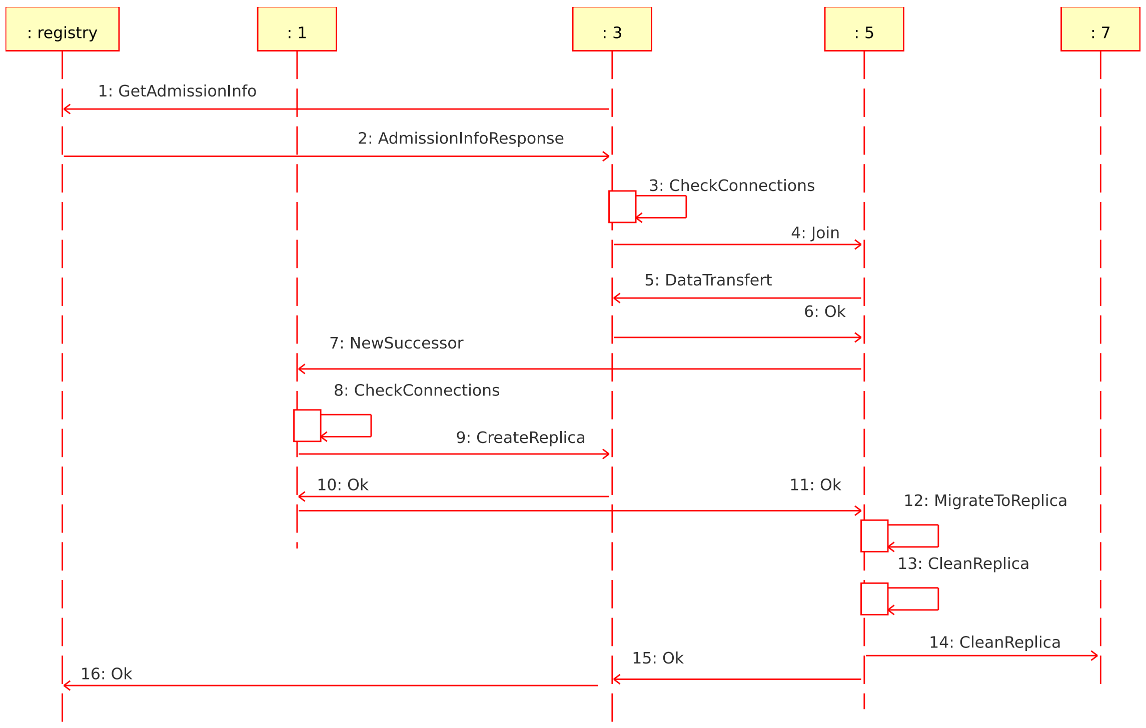


Figure 4.3: Admitting a new agent: agent 3 joins via agent 5

Fig. 4.4 presents the case where the predecessor of the joining node cannot connect to the joining node and therefore answers to the `NewSuccessor` message sent by the successor with a `Ko` message meaning that the admission has to be aborted. This message is then forwarded to the joining node and finally to the Registry. The scenario where the predecessor cannot replicate its data on the joining node is similar.

If the connections with the successor and predecessor provided by the Registry cannot be established (operation 3 of the admission process), the admission process is aborted by sending a `Ko` message to the Registry and the agent restarts all the process until it can insert itself into the ring. This situation is illustrated on Fig. 4.5.

Put operation

The put operation of the Registry DHT is a bit different from the common put. This implementation prevents an existing value from being overwritten by a put using the same key and furthermore returns the value associated with the key.

When an agent (or an application running on within the agent) executes the "`put`" operation of the DHT, the following messages are exchanged (see Fig. 4.6):

1. The agent willing to store a $(key, value)$ pair into the DHT, determines the agent it thinks should store the key and sends a `Put` message to it;

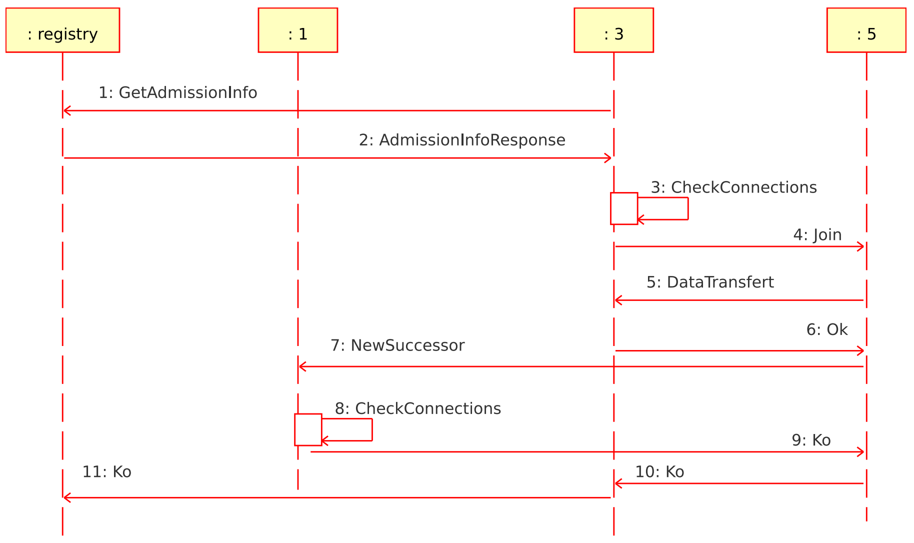


Figure 4.4: Admitting a new agent: agent 3 joins via agent 5 (double ack fails)

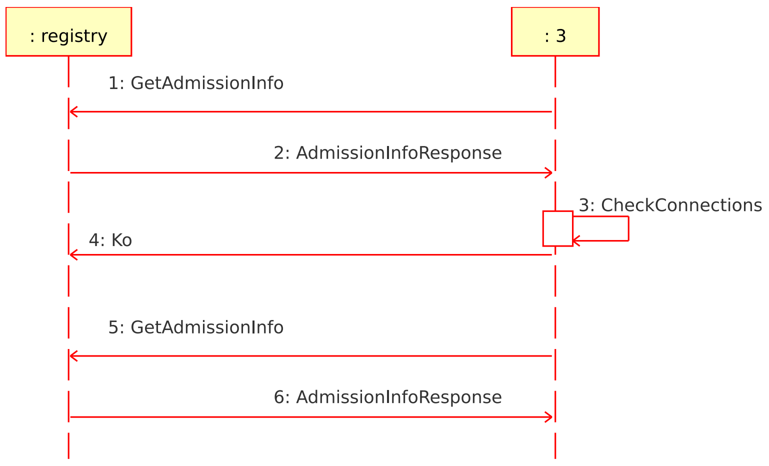


Figure 4.5: Admitting a new agent: agent 3 joins (connections check fails)

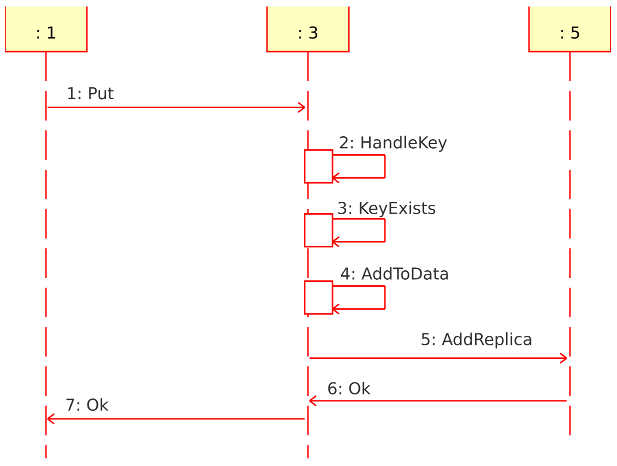


Figure 4.6: Put operation

2. The agent receiving the `Put` message checks if it actually handles the key;
3. The agent in charge of the key checks if the key already exists;
4. The key does not exist and is thus created with the data sent as value;
5. The agent in charge of the key creates a replica of the new $(key, value)$ pair on its successor;
6. The successor confirms that the replica has been created;
7. The agent informs the agent that sent the request that the put operation was successful.

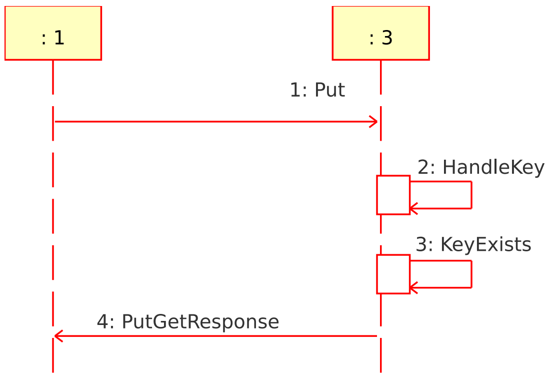


Figure 4.7: Put operation (key already exists)

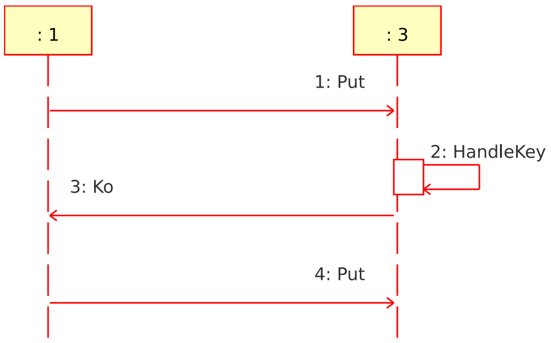


Figure 4.8: Put operation with error

If the key already exists, the agent in charge of the key directly answers with a `PutGet-Response` message with the value currently associated with the key as payload. This behavior is illustrated on Fig. 4.7.

Finally, if the agent receiving the `Put` message is not in charge of the key, it simply responds with a `Ko` message. In case of failure of an agent, the request could be sent to the correct agent but the agent itself could not be aware it is in charge of the key. This is why the agent that sent the request will try to put its data again (after a short delay), resolving the agent in charge of the key and sending a new `Put` message to that agent (see Fig. 4.8). This process will not endlessly loop because either the agent sending the request will send it to the correct agent, or the agent receiving the request will learn it is in charge of that key and then will process the request.

Remove operation

When an agent (or an application running on within the agent) executes the "remove" operation of the DHT, the following messages are exchanged (see Fig. 4.9):

1. The agent willing to remove data associated with a given key from the DHT sends a `Remove` message to the agent it thinks should store the key;
2. The agent receiving the `Remove` message checks if it actually handles the key;

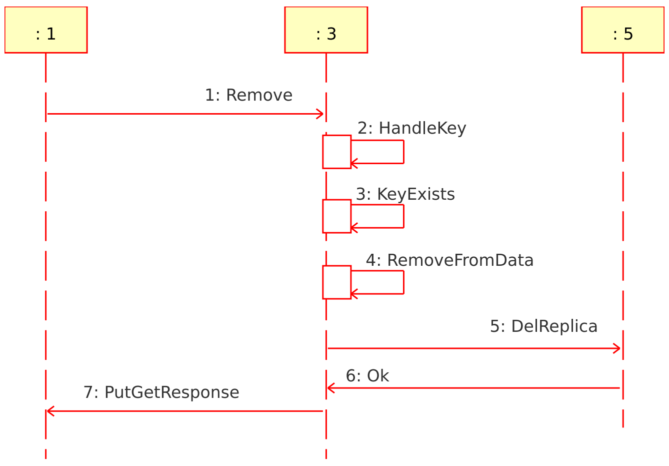


Figure 4.9: Remove operation

3. The agent in charge of the key checks if the key already exists;
4. The key exists and is thus removed from the data stored locally;
5. The agent in charge of the key tells its successor to remove the replica of the key;
6. The successor confirms that the replica has been destroyed;
7. The agent informs the agent that sent the `Remove` request that the remove operation was successful.

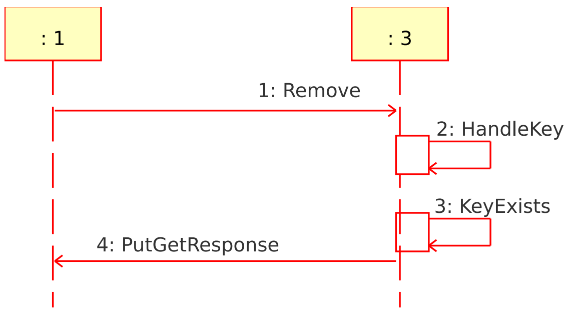


Figure 4.10: Remove operation (key does not exist)

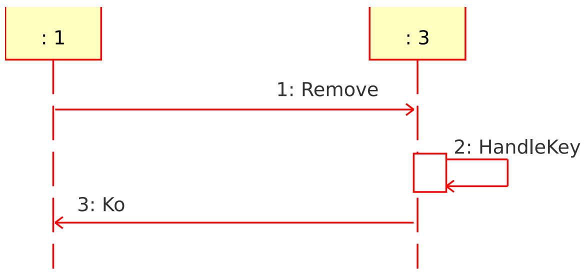


Figure 4.11: Remove operation (with error)

If the key does not exist, the agent answers normally like it should have done after a successful remove. This behavior is developed on Fig. 4.10.

Finally, if the agent receiving the `Remove` message is not in charge of the key, it simply responds with a `Ko` message. This is presented on Fig. 4.11.

Get operation

When an agent (or an application running on within the agent) executes the “`get`” operation of the DHT, the following messages are exchanged (see Fig. 4.12):

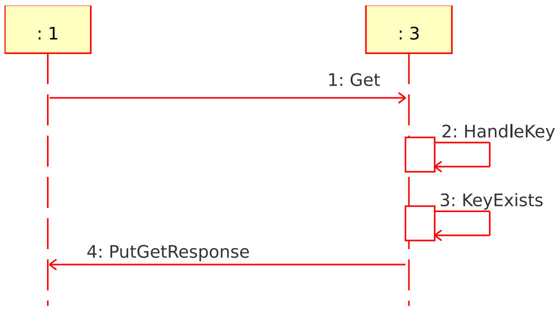


Figure 4.12: Get operation

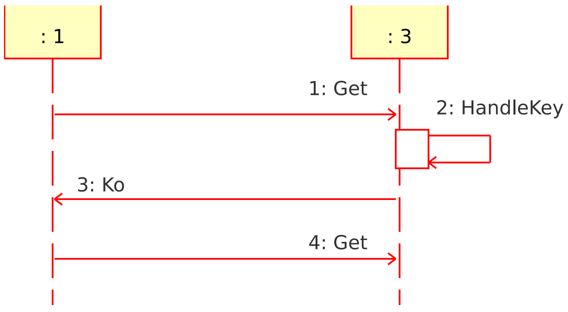


Figure 4.13: Get operation (with error)

1. The agent willing to retrieve a $(key, value)$ pair from the DHT sends a `Get` message to the agent it thinks is in charge of the key;
2. The agent receiving the `Get` message checks if it actually handles the key;
3. The agent in charge of the key checks if the key already exists;
4. The key exists and the valued associated with it is sent back in a `PutGetResponse` to the agent that sent the request.

If the agent receiving the `Get` request does not handle the requested key, it answers with a `Ko` message. Similarly to the *Put* operation, the agent which sent the request will try again after a short delay (Fig. 4.13). This new request could be sent to the same agent as the first time but either the request will be sent to another agent or the agent receiving the request will know it is now in charge of the key.

Failure or departure of an agent

When an agent fails or leaves the system, the registry is notified and sends to the successor of the leaving agent information about its new predecessor. In order to get back to a consistent state of the DHT, the following messages are exchanged (see Fig. 4.14):

1. The Registry learns an agent failed (or left) and thus sends a `NewPredecessor` request to the successor of the leaving agent; this message contains info about its new predecessor on the DHT ring;
2. The successor checks if it knows and can connect to the agent specified in the `New-Predecessor` message;
3. The successor lets the predecessor know that it has a new successor;
4. The predecessor checks that it knows about the successor and can connect to it;
5. The predecessor creates data replica on its soon-to-be successor;
6. The successor confirms the creation of the replica
7. The predecessor updates its successor pointer to the successor and tells it that it is ready;

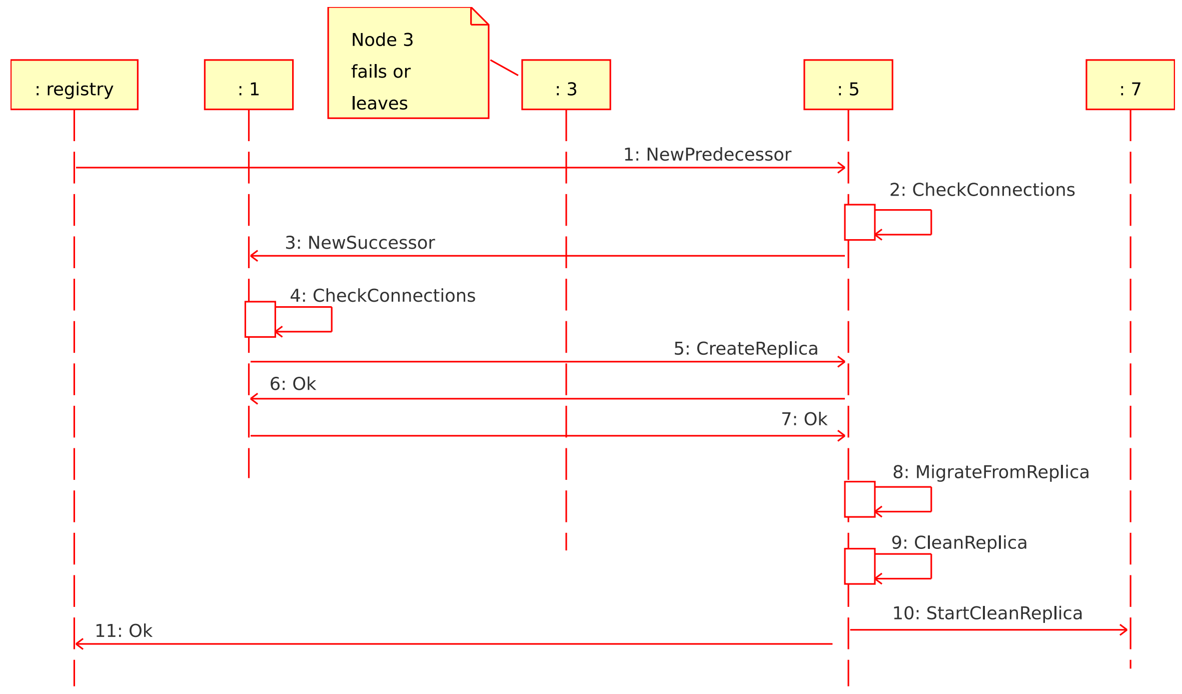


Figure 4.14: Failure or departure of an agent

8. The successor migrates data it will not be handling anymore to its replica;
9. The successor transfers data that it now handles from its replica;
10. The successor removes data that will not be handled anymore by its predecessor from its replica;
11. The successor tells its own successor to clean its replica too;
12. The successor updates its predecessor pointer to the predecessor and sends confirmation to the Registry that the ring has been fixed.

If this operation fails, the only solution is to wait for a few seconds and try again. In the meantime, other DHT operations must be added to a waiting queue because the registry's view of the DHT ring is not consistent.

4.5.2 Class diagram

Fig. 4.15 shows the class diagram of the DHT-based SessionRegistry and how it hooks up with the current implementation.

4.6 Limitations

4.6.1 Message delivery

In this implementation, the assumption is made that every message is delivered in a finite time or the socket where it was sent to gets closed. Therefore, there is no timeout triggered if the

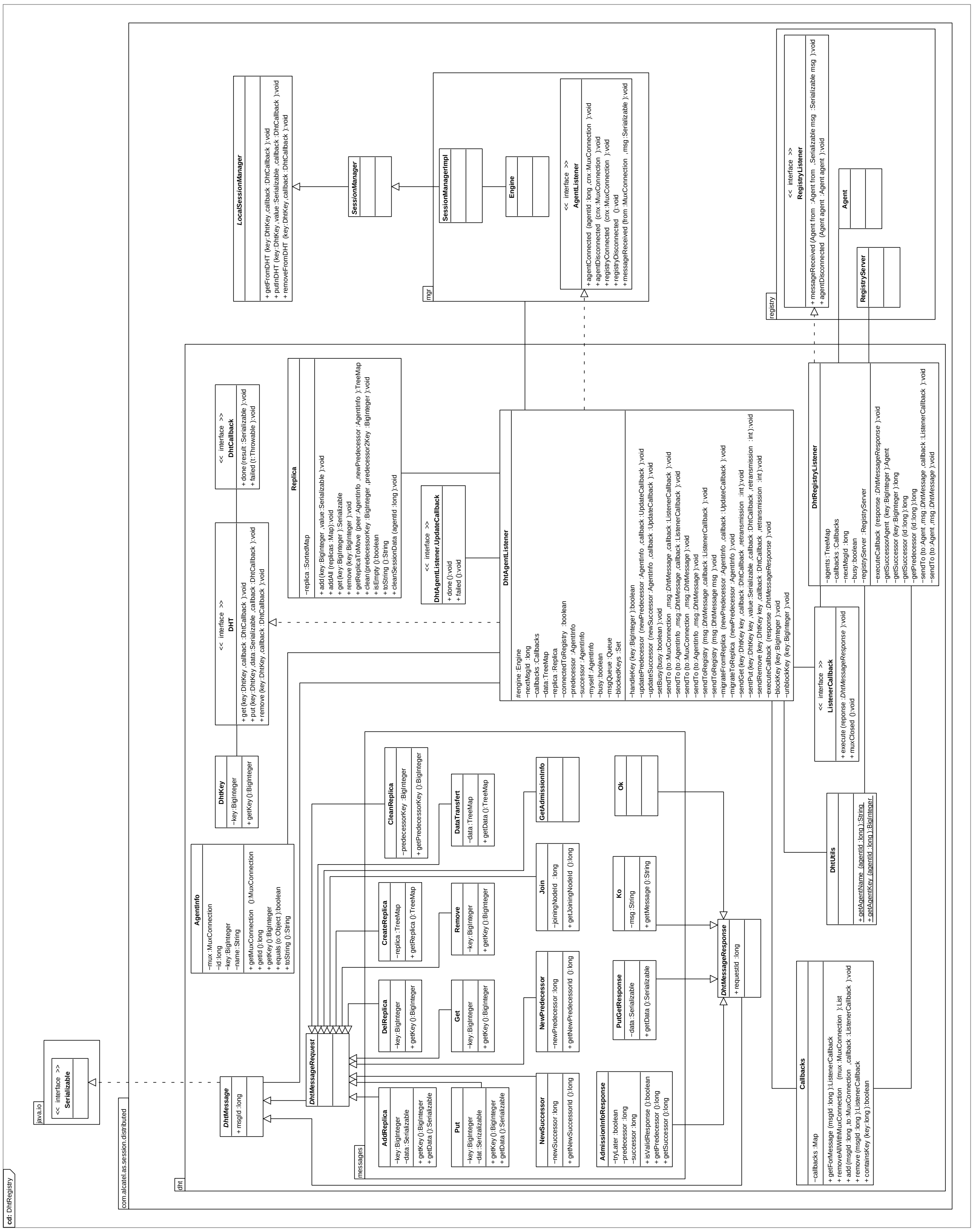


Figure 4.15: Class diagram of the DHT layer

message is not received after a fixed delay. On the other hand, if the socket used to communicate with another agent is closed, the agent detects it and reacts accordingly.

4.6.2 Agent memory

The $(key, value)$ pairs being stored on the DHT nodes, it consumes memory of the agents and no longer of the registry. This situation must be taken into account when deploying and configuring the agents.

4.7 Performance

4.7.1 Environment

Performance tests were performed on 5 dual Intel Xeon 3.4Ghz. They were all connected on a private Gigabit Ethernet network. The Session Registry and SipStack components were deployed on one server with default parameters except for the multicast discovery support that has been activated in the Session Registry. For the tests, a maximum of 2 agents were deployed on each of the remaining servers with 512 megabytes of dedicated memory per agent; they were deployed in order to have one CPU per agent. Finally, in order to obtain consistent results, the agents had fixed identifier on the DHT ring. These identifiers were chosen in order to have a uniformed distribution of the agents on the ring.

4.7.2 Description of the test

In order to estimate the performance of the SessionAPI using a DHT layer to store the information about the location of a session (this SessionAPI is then called SessionDhtAPI in the present report), a simple test was run. This test, embedded in a SIP Servlet and triggered by sending an empty `INFO` SIP request to it, consists of 4 steps:

Warm up In order to get the Java Virtual Machine (VM) in steady-state conditions, 3,500 sessions are synchronously created. This is required to avoid having the Java just-in-time compiler eating CPU cycles during the test;

Insertion 7,500 new sessions are asynchronously created and an attribute is set on each of them;

Retrieval Once the 7,500 sessions have been created, the test asynchronously retrieves them;

Suppression Finally, once all the sessions have been successfully retrieved, the test destroys them all.

The next section presents the results of this test run with 2, 4 or 8 running agents using the standard SessionAPI and the SessionDhtAPI.

4.7.3 Results

Creating a session

The high delay (30 seconds) when creating the sessions with only 2 running agents shown on Fig. 4.16 is easily explained by the fact that they are the only 2 nodes of the DHT and therefore the successor of each other. This means that insertion of any data on the first agent will result with the creation of a duplicate on the other one and insertion on the second agent means duplicate creation on the first one. This situation puts an heavy load on both agents resulting in poor performance. Anyway, deploying a DHT with only 2 nodes is somehow a nonsense.

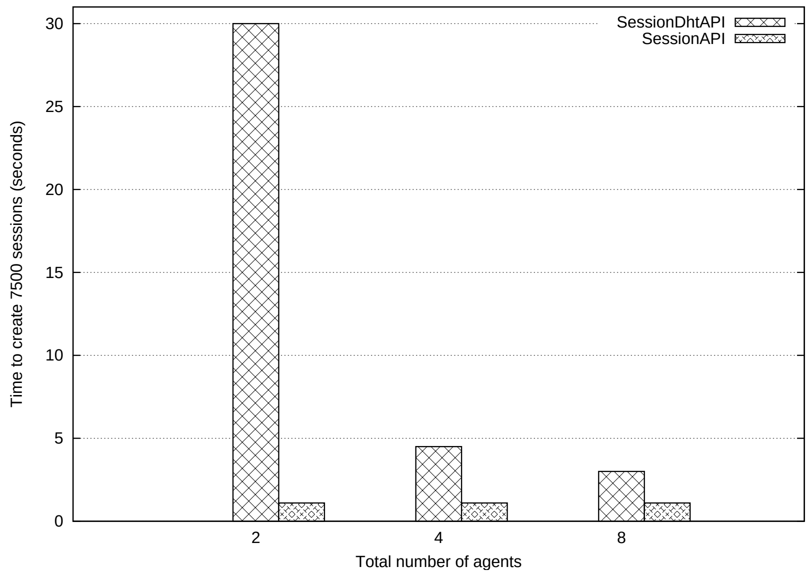


Figure 4.16: Performance of the `createSession` operation

When 4 agents are present, the timing for the creation of the sessions gets reasonable and with 8 agents timing is even better, this is due to a better distribution of the load among the running agents.

The performance for the session creation with the DHT layer is 3 to 4 times slower than using the standard SessionAPI. These performance are however acceptable given the fact that each session location information is duplicated in the DHT.

Retrieving a session

As shown on Fig. 4.17, retrieving a session is not influenced by the number of agents in the DHT. The slightly higher latency between the standard SessionAPI and the SessionDhtAPI is due to the overhead of computing the key and exchanging messages to get the data from the DHT.

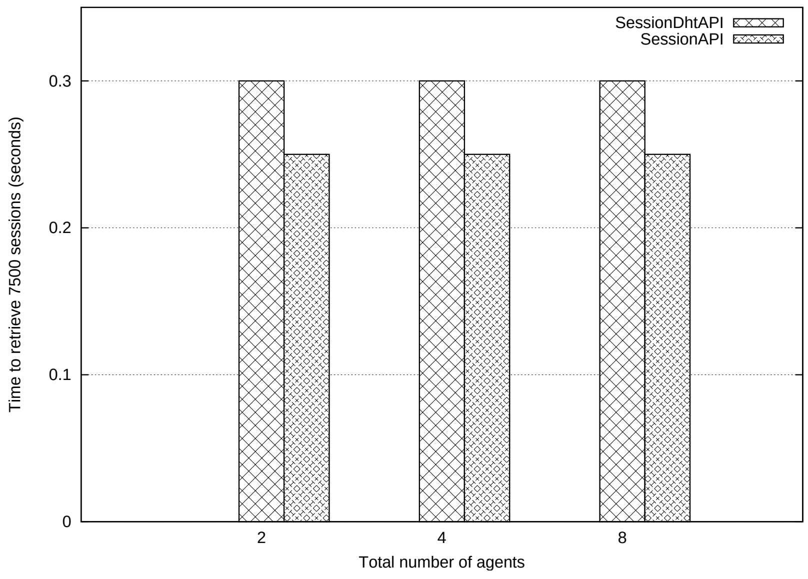


Figure 4.17: Performance of the `getSession` operation

Destroying a session

Fig. 4.18 shows, again, that in a deployment with only 2 agents, the fact that they are the successor of each other increases their load and leads to higher latency. When 4 or more agents are deployed, the latency to remove data from the DHT is almost constant. The difference of timing between the SessionAPI and the SessionDhtAPI is due to the computing of the key, the exchanges of messages between the agents to remove the data from the root node and to remove the duplicate.

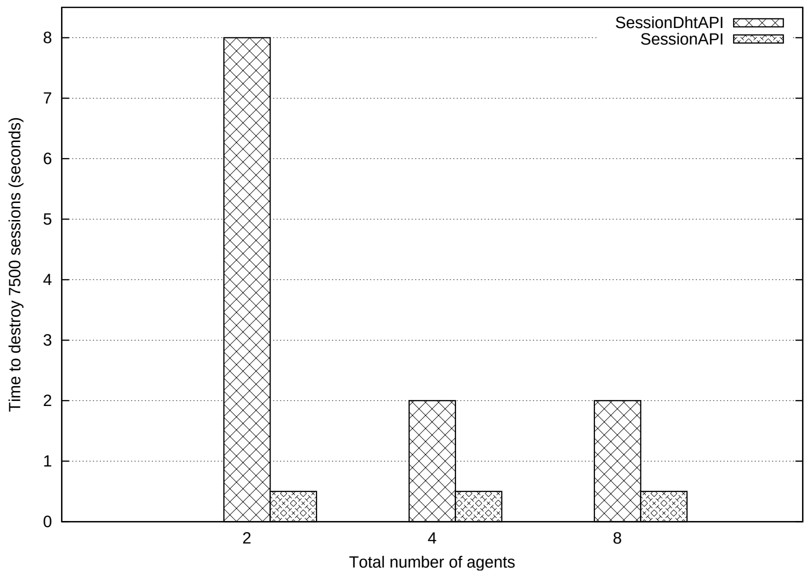


Figure 4.18: Performance of the `destroySession` operation

4.7.4 Conclusion

The performance tests have shown that the use of a DHT for the SessionRegistry is a viable solution (as long as it is used with a deployment of more than 2 agents). The DHT-based SessionRegistry addresses the main shortcomings of the “basic” implementation. Indeed, after a crash of the SessionRegistry component, the system does not have to rebuild the index of the localization of each session anymore because this index does not exist anymore. Also, this index is now also replicated increasing the fault tolerance of the whole system. However, this DHT-based SessionRegistry should be used cautiously since it introduces higher latencies and consumes memory in the same space as the agents.

Chapter 5

Service discovery in a DHT environment

The importance of discovering and locating services within a P2PSIP network has already been introduced in Section 2.4. This chapter begins with a presentation of the problem in the context of STUN. Then, since P2PSIP is using a DHT as distribution system, the problem can be reduced to finding services information in a DHT environment. Therefore, the next section describes two naive and simple solutions of indexing techniques on top of a DHT; these two solutions were also pointed out and briefly discussed by the P2PSIP working group. Finally, a proposal of a service discovery system based on the usage of a Prefix Hash Tree [19] (PHT) is presented.

5.1 STUN and P2PSIP

A STUN server allows a client to discover its connectivity. It enables the client to find out its public address, the type of NAT (or multiple NATs) it eventually lies behind and the Internet side UDP port associated by the NAT with a particular local UDP port. In the end, STUN enables a peer to start NAT traversal mechanisms if needed.

5.1.1 STUN servers in a P2PSIP environment

In classic P2PSIP environment, many user agent clients (UACs) that participate to the overlay will be running on standard home or office computers located behind different kinds of NAT. This situation will require NAT traversal mechanisms in order to be able to set up a connection with another client (that might itself be behind some sort of NAT too).

While most of these peers can directly provide services like proxying or registration using a pre-configured NAT traversal technique (like port forwarding), running a STUN server on a NATted peer is not an option. In classic C/S SIP, a STUN server can be running inside a Wide Area Network (WAN) to provide connectivity information to clients located in NATted Local Area Networks (LAN) managed by the WAN and willing to communicate with another client inside the same WAN. This scenario also assumes that the WAN does not allow such communications to the outside.

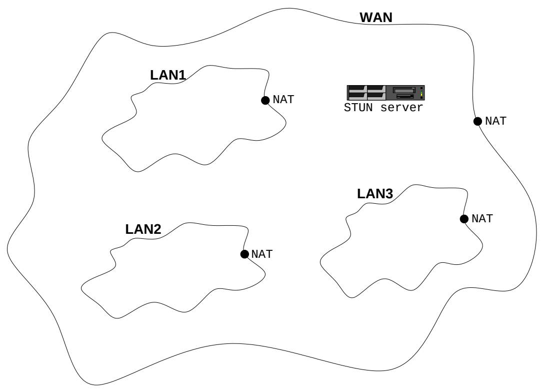


Figure 5.1: WAN network with 3 sub-LAN behind NATs using a global STUN server

In the previous illustration, clients in the various LANs know about the network localization of the STUN server. However in a global P2PSIP scenario, nothing is known or can be assumed about a peer connectivity, preventing such use of a STUN server. On the other hand, one major advantage of a P2PSIP overlay with many peers is that many of them are potential STUN servers. But in order to become a STUN server, it must check that it is not behind a NAT.

5.1.2 Bootstrap STUN server

To ensure it is not behind a NAT or some limited connection, a peer must be able to contact at least one STUN server. The address of this STUN server can be provided by the admitting peer, which will need a way to know that information, or by some kind of service localization mechanism. If a peer becomes a STUN server, it must publish this information using the same service localization mechanism.

Finally, if we look at a starting P2PSIP overlay, it is obvious that some bootstrap STUN server, probably collocated with the bootstrap node, must be available and registered in the service localization system too.

5.1.3 Limitation

The major limitation of such system is that only the peers with open connection to the global Internet will become STUN servers for the overlay. This restriction can limit the amount of STUN servers available within the overlay.

If two peers, each behind different NATs but behind a common NAT at the next hop, want to communicate they will have to contact one of the overlay STUN servers located outside any NAT although it might exist a peer inside their top-level NAT which could be able to provide them STUN services. This is illustrated on Fig. 5.2 where the dotted lines represent the connections to the outside STUN server and the dashed lines the connections that would happen if the 'local' STUN server was used.

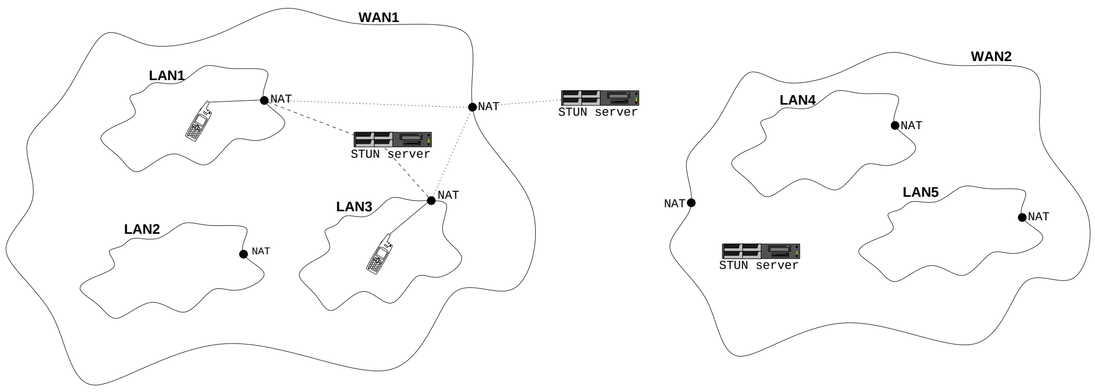


Figure 5.2: WAN networks using a third-party STUN server

5.2 Naive solutions

This section presents two naive indexing techniques in a P2PSIP environment. For each “solution”, the reason why it is not acceptable for a full deployment is briefly discussed.

5.2.1 Centralized server

The trivial solution to the service localization problem is to use an unique server (at least seen as unique by the user) running on a well known address on which localization information about the available services is stored. Unfortunately this solution is not acceptable because it is re-introducing a centralized server in an environment trying to get rid of any form of centralization and is a single point of failure.

5.2.2 Storing the service information in the DHT

Another solution is to store information about the services directly into the DHT either storing all the information under a single key or using a special prefix for keys containing the information.

Using a single key

All the service information could be associated to a single reserved key and stored into the DHT. This approach is however not practical for several reasons, the most obvious ones being:

- the quantity of information can be very important, especially if the overlay is big;
- it introduces a single point of failure into the system (if the DHT does not provide any replication mechanism);
- the load on the node in charge of the reserved key will be high.

Using a reserved prefix

This solution is pretty simple, it uses a well-known prefix and adds an index number to it (e.g. *STUN1*, *TURN14*, . . .). This requires reserving one prefix for each kind of service that needs to be stored in the DHT and storing a specially reserved key (e.g. *STUNINDEX*) per service that can keep track of the indexes that are in use (or just information about the available range).

Unfortunately as a DHT only supports exact lookup queries, the client should then perform many lookups in the DHT to retrieve the service information. Many of these queries may fail before starting to receive useful information. Also, as in the single key mechanism, a lot of responsibilities are given to a single node which may fail or be targeted by different kinds of attack.

5.3 Indexing techniques in DHTs

Different techniques are available to index data on top of a DHT, the most common are Skip Graphs and Prefix Hash Tree.

5.3.1 Skip Graphs

Skip Graphs [20] are a distributed data structure based on skip lists [21]. They have been designed to allow searches in peer-to-peer systems and queries based on key ordering. Skip Graphs exploit the underlying tree structure of most DHTs (Chord, CAN, and similar DHTs look like a balanced tree based on the near-uniform distribution of the output of the hash function). Skip Graphs use this tree structure to provide tree functionality and then add balancing and load distribution. Therefore, they require a DHT whose distribution structure can be linked to a tree structure and they have to access to lower levels of the DHT implementation in order to be implemented.

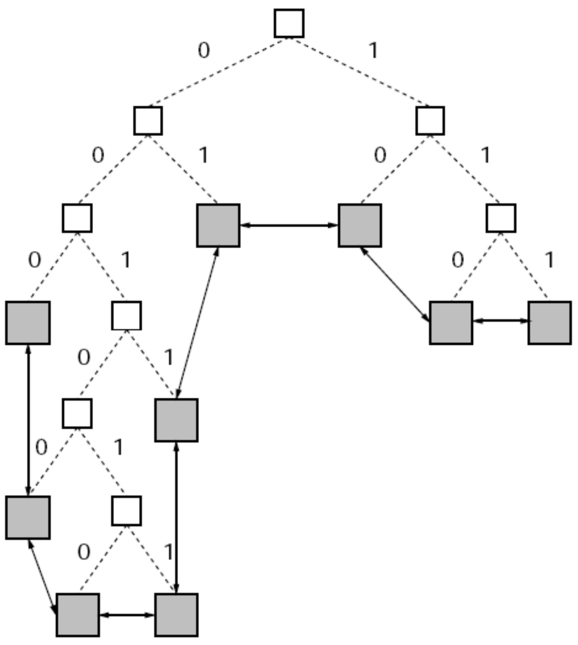
Skip Graphs and derived techniques like [22] assume knowledge of or require changes to the DHT topology or routing behavior. However the P2PSIP architecture, as stated in the working group charter [1], is going to support multiple DHT implementations; then any indexing system willing to be deployed in a P2PSIP environment has to be built entirely on top of the classic high-level DHT interface (`put`, `get`, `remove`). This constraint therefore excludes techniques like Skip Graphs.

5.3.2 Prefix Hash Tree

A Prefix Hash Tree [19, 23] (PHT) is a binary trie[1] of binary strings of length D. A PHT has 2 parameters: D, the length of the binary strings of the indexed domain and B, the maximum number of keys stored in an internal node.

Each node of the trie is labeled with a prefix that is defined recursively: given a node with label l, its left and right child nodes are labeled $l0$ and $l1$ respectively. The root is labeled with the attribute being indexed, and downstream nodes are labeled as above. The following properties are invariant in a PHT:

[1]ordered tree data structure that is used to store an associative array where the keys are strings, more detailed information can be found in [24]



Leaf nodes	Keys
000*	000001
	000100
	000100
00100*	001001
001010*	001010
	001010
	001010
001011*	001011
	001011
0011*	
01*	010000
	010101
10*	100010
	101011
	101111
110*	110000
	110010
	110011
	110110
111*	111000
	111010

Figure 5.3: Prefix Hash Tree (from [19]) – leaf nodes are grayed

Universal prefix Each node has either 0 or 2 children.

Key storage A key K is stored at a leaf node whose label is a prefix of K.

Split Each leaf node stores at most B keys.

Merge Each internal node contains at least $(B + 1)$ keys in its sub-tree.

Threaded leaves Each leaf node maintains a pointer to the leaf nodes on its immediate left and and immediate right respectively.

A PHT is built over a DHT and its internal nodes are distributed among the DHT nodes by hashing the prefix labels of PHT nodes over the DHT identifier space and using them as key for storage into the DHT. For more detailed information, please refer to [19].

Fig. 5.3 shows a sample of PHT containing 20 6-bit keys with $B = 4$. The table on the right lists the 20 keys and the leaf nodes they are stored in. Some keys appear multiple times in the table, the reason for that is not explained in the paper. However, in this context, we can ignore duplicate entries because they would represent the exact same service providers.

Limitations

Unfortunately, PHTs have some limitations. They only support 1-dimensional range queries (multi-dimensional range queries can be implemented using PHTs but require the use of complex projections). The second limitation of PHTs is the fact that the indexed data are binary strings, limiting the format of the indexed data.

5.4 Solution using a Prefix Hash Tree

Prefix Hash Tree [19] can be deployed on top of any DHT and then fits well in the P2PSIP constraints; it has also efficient algorithms ($O(\log N)$) that can be enhanced when the distribution has known properties (can run in $O(\log \log N)$).

Another system that could provide a service localization system within a P2PSIP overlay based on a DHT can be implemented using a Prefix Hash Tree that indexes specially crafted binary strings (these strings are described in Section 5.4.1).

5.4.1 Indexed data

The information about a service has to be represented as binary string. As URIs are unlimited conventional strings, they cannot be used and a special data format needs to be created. Furthermore this binary string has to be formatted so that a prefix search or a range search can target a subset (specific service type, IP range, ...) of services stored in the PHT.

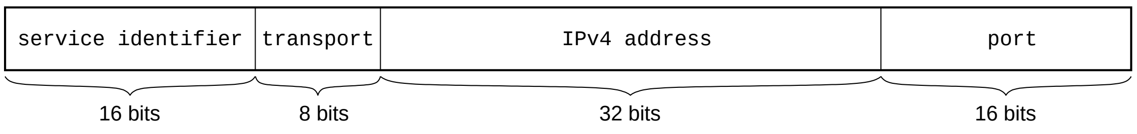


Figure 5.4: Format of the 72-bit binary strings indexed in the PHT

Fig. 5.4 shows the format of the 72-bit binary string that should be used to allow prefix and range searches such as previously described. The 72-bit binary string is formed by a 16-bit service identifier (this identifier has to be unique in the DHT and should be standardized or should use existing standardization) followed by a 8-bit transport field (using a value defined in [25]), a 32-bit IPv4 address and a 16-bit port number.

This format can easily be adapted to support IPv6 addresses by using a 128-bit field for the IP address and using the native IPv6 support for IPv4 addresses to continue storing IPv4 addresses in the PHT.

5.4.2 Storing service information

To store the service information, the service provider has just to insert its information into the PHT using the 72-bit binary string format described in 5.4.1.

5.4.3 Retrieving service information

Any node can perform search for services in the PHT using one of the algorithms described in [19]. Using prefix searches allows a node to lookup for any service providers based on their capabilities, the protocol they use, their IP address, and so on. Range searches allow a node to look for any service providers in a specific IP range, or running on a specific port range, or providing a range of services and so on.

As a P2PSIP network could contain a large amount of service providers, it is recommended to stop searching service providers when a predefined number of providers have been found.

5.4.4 Maintaining service information

As some internal nodes of a PHT can be lost if the node where they are stored leaves the overlay, each service provider should periodically refresh its record in the PHT.

5.5 Remarks

Patent

The work presented in the last section of this chapter is the subject of a pending patent.

P2PSIP charter excludes non-URIs localization

At the time of the writing of the work presented in this chapter, there were discussions about what kind of localization would be accepted by the working group charter; at that time no consensus had been reached and some were in favor of accepting other forms than URIs.

Finally, the charter submitted to the Internet Engineering Steering Group (IESG) states that *"arbitrary search of attributes is out of scope, but locating resources based on their URIs is in scope"*. This decision definitely excludes the usage of a PHT, except if a decent way of indexing URIs is found.

Chapter 6

P2PSIP Proxy/Registrar prototypes

This chapter presents the generic design of a distributed Proxy/Registrar as a SIP Servlet [26] and details the different implementations that have been realized. This distributed Proxy/Registrar can be plugged to any DHT providing a `put`, `get`, `remove` asynchronous interface.

It is based on a distribution model where only the servers participate to the distribution (see Section 2.2.1 and Fig. 2.1). In fact, this choice of distribution model does not contradict the model on which P2PSIP is based (the "super-node" model, see Fig. 2.3) since the model used in this chapter can be seen as a "super-node" model in which all servers have become super-node. To complete the parallelism with the P2PSIP model, the protocol used by the DHT implementation to manage the distribution can be considered as the P2PSIP Peer protocol and SIP as the P2PSIP Client protocol. However, this comparison is not fully compliant with the WG Charter [1] since SIP is not a subset of the protocol used by the DHT implementation.

Finally, at the end of this chapter, performance tests of these Servlets and their results are presented and discussed.

6.1 P2PSIP Servlet

The P2PSIP Servlet is a SIP Servlet [26] providing the functionality of a distributed Proxy/-Registrar. In such a distributed deployment, the client shall not be aware that the Proxy/Registrar is distributed. It has just to send its requests to one of the participating servers.

The proposed implementation is an abstract class that needs to be derived to support a specific DHT implementation. It has been required that it would support Java and be asynchronous, so as to work on the A5350 platform. Given these requirements, a thorough evaluation of the existing implementations of the major DHT algorithms in view of their insertion into the P2PSIP Servlet class lead to the implementation and test of three instances. Those instances are mono-threaded and rely exclusively on asynchronous calls and callbacks.

6.1.1 Class diagram

Fig. 6.1 presents the class diagram of the P2PSIP Servlet. Some implementations (ChordSipServlet, BambooSipServlet and SipDHT) are also visible on the diagram as well as some classes used to interact with the implementations (P2PSipCallback and P2PSipPayload). They will be discussed in the following.

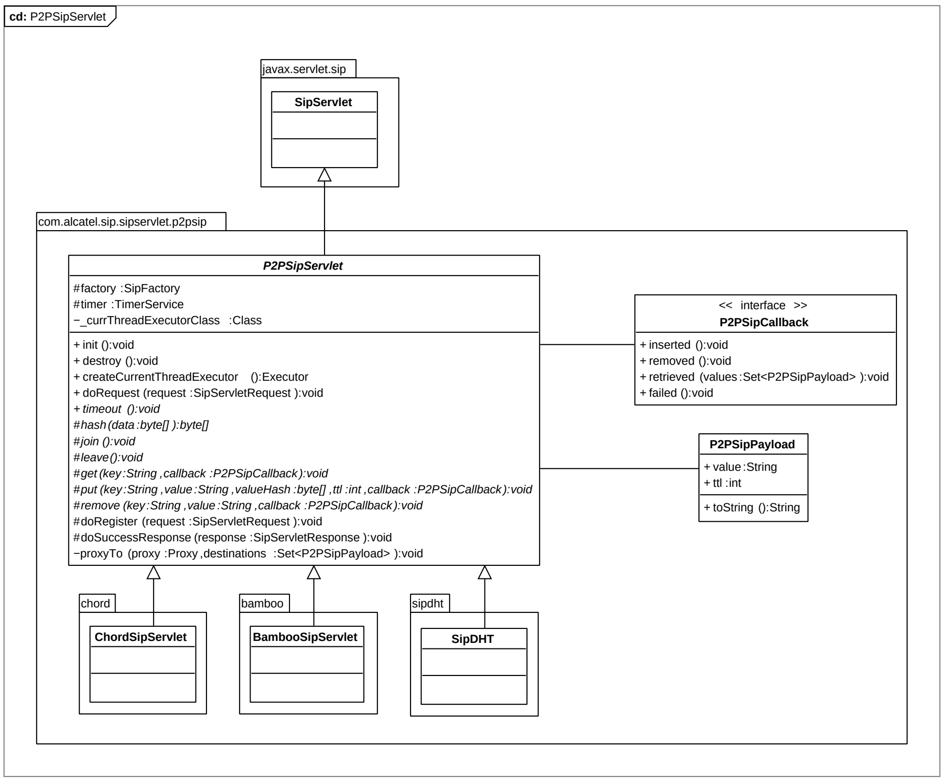

cd: P2PSipServlet
javax.servlet.sip
SipServlet
com.alcatel.sip.sipservlet.p2psip
P2PSipServlet
#factory :SipFactory
#timer :TimerService
−_currThreadExecutorClass :Class
+ init ():void
+ destroy ():void
+ createCurrentThreadExecutor ():Executor
+ doRequest (request :SipServletRequest):void
+ timeout ():void
#hash (data :byte[]):byte[]
#join ():void
#leave():void
#get (key:String ,callback :P2PSipCallback):void
#put (key:String ,value:String ,valueHash :byte[] ,ttl :int ,callback :P2PSipCallback):void
#remove (key:String ,value :String ,callback :P2PSipCallback):void
#doRegister (request :SipServletRequest):void
#doSuccessResponse (response :SipServletResponse):void
−proxyTo (proxy :Proxy ,destinations :Set<P2PSipPayload>):void
<< interface >>
P2PSipCallback
+ inserted ():void
+ removed ():void
+ retrieved (values :Set<P2PSipPayload>):void
+ failed ():void
P2PSipPayload
+ value :String
+ ttl :int
+ toString ():String
chord
ChordSipServlet
bamboo
BambooSipServlet
sipdht
SipDHT

Figure 6.1: P2PSIP SERVLET class diagram

6.1.2 Deriving P2PSIP Servlet

In order to provide support for a specific DHT, the derived class must implement the following abstract methods:

`hash`

The `hash` method is used to produce a hash of a byte array, this method is to be implemented by the DHT abstraction layer because some DHT may have some hashing usage limitations (like Bamboo which only supports SHA).

Parameters

- `data`: the array of bytes to hash

Returns An array of bytes containing the hash.

`timeout`

The `timeout` method is invoked when a ServletTimer is fired (this method can be empty if the P2PSIP Servlet implementation does not use any ServletTimer).

`join`

The `join` method is called when the P2PSIP Servlet wants to join the DHT overlay (this actually happens during the call to the `init` method of the P2PSIP Servlet).

This method can also throw a `ServletException` as it is called by the `init` method of the P2PSIP Servlet and if the node cannot join the overlay the P2PSIP Servlet must fail its initialization.

`leave`

The `leave` method is called when the P2PSIP Servlet is destroyed and therefore wants to properly leave the DHT overlay.

`get`

The `get` method is used to retrieve data from the DHT. This method is asynchronous and thus returns immediately. The callback provided as parameter is called when the data has been retrieved from the DHT or the system failed to retrieve the data.

Parameters

- `key`: a String representing the key identifying the requested data in the DHT;
- `callback`: a P2PSipCallback that will be called when the retrieval process is done.

`put`

The `put` method is used to store data into the DHT. This method is asynchronous and thus returns immediately, the callback provided as parameter is called when the data has been stored into the DHT or the system failed to store the data.

Parameters

- `key`: A String representing the key identifying the data to store into the DHT;
- `value`: A String containing the data to store into the DHT;
- `valueHash`: An array of bytes containing a hashed representation of `value`;
- `ttl`: An integer fixing the time-to-live of the data in the DHT;
- `callback`: a P2PSipCallback that will be called when the storing process is done.

`remove`

The `remove` method is used to remove data from the DHT. This method is asynchronous and thus returns immediately, the callback provided as parameter is called when the data has been removed from the DHT or the system failed to remove the data.

Parameters

- `key`: A String representing the key identifying the data to remove from the DHT;
- `value`: A String containing the data to remove from the DHT;
- `callback`: a P2PSipCallback that will be called when the removal process is done.

The value to remove is passed as second parameter to deal with multivalued keys.

6.1.3 Asynchronous calls and callbacks

As previously pointed out, the P2PSIP Servlet implementation uses asynchronous calls. The main reason for that is the design of the A5350 platform requiring that a SIP Servlet does not perform any blocking call as every SIP Servlet in the container is in fact running in a single thread (performing a blocking call would result in blocking all the SIP Servlets).

Each method that needs to send back information about its execution must therefore have knowledge of a callback object. Such callback object for the abstract methods of a P2PSIP Servlet must implement the P2PSipCallback interface and be given as argument to the asynchronous methods of P2PSIP Servlet.

6.1.4 Threading in the DHT layer

The P2PSIP Servlet and all other SIP Servlets are running in a single thread but in order to allow the usage of threads in the DHT layer (like Bamboo and OpenChord do), the `create-CurrentThreadExecutor` method is used. The DHT layer can use it to tell the SIP Servlet thread to execute some task by invoking the `execute` of the `java.util.concurrent.-Executor` object that the method returns.

6.1.5 Limitation

Currently, the main limitation is the security. The P2PSIP Servlet and its different implementations so far do not take security into account. They assume that every participant of the DHT is trustworthy.

6.2 ChordSipServlet

The ChordSipServlet is a P2PSIP Servlet that relies on a OpenChord DHT.

6.2.1 OpenChord

OpenChord is a Chord [8] implementation realized by the *Distributed and Mobile Systems Group* of the Otto-Friedrich-Universität of Bamberg and released under the GPL license. It provides both a synchronous and an asynchronous Application Programming Interface (API), an application layer handling storage of the data and is written to support generic keys and data.

6.2.2 Class diagram

Fig. 6.2 shows the class diagram of the ChordSipServlet.

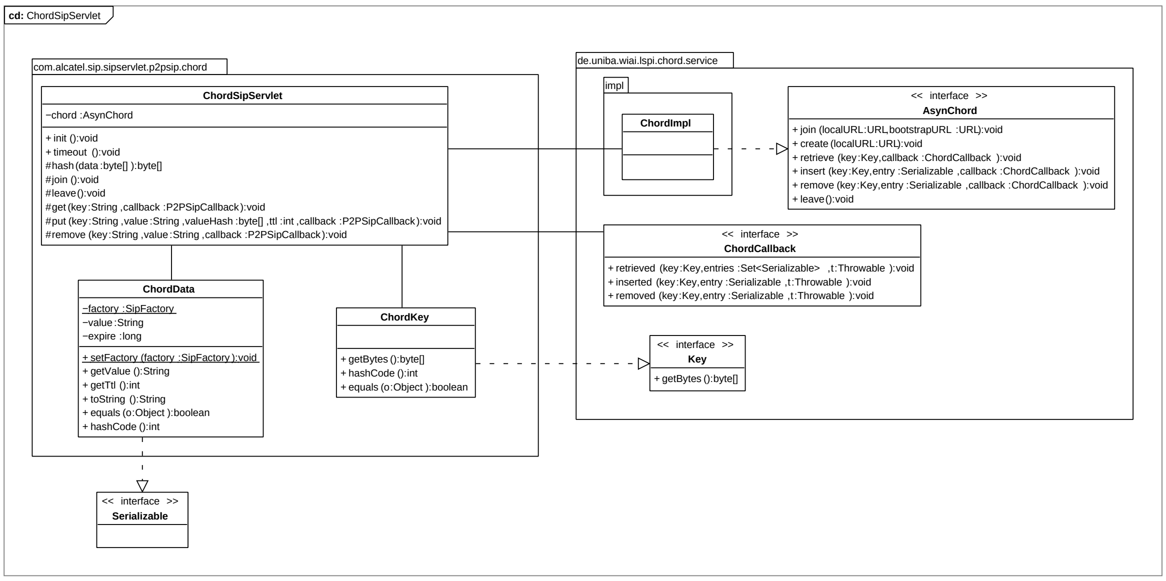

cd: ChordSipServlet
com.alcatel.sip.sipservlet.p2psip.chord
ChordSipServlet
−chord :AsynChord
+ init ():void
+ timeout ():void
#hash (data :byte[]):byte[]
#join ():void
#leave ():void
#get (key:String ,callback :P2PSipCallback):void
#put (key:String ,value:String ,valueHash :byte[] ,ttl :int ,callback :P2PSipCallback):void
#remove (key:String ,value:String ,callback :P2PSipCallback):void
ChordData
−factory :SipFactory
−value :String
−expire :long
+ setFactory (factory :SipFactory):void
+ getValue ():String
+ getTtl ():int
+ toString ():String
+ equals (o:Object):boolean
+ hashCode ():int
ChordKey
+ getBytes ():byte[]
+ hashCode ():int
+ equals (o:Object):boolean
<< interface >>
Serializable
de.uniba.wiai.lspi.chord.service
impl
ChordImpl
<< interface >>
AsynChord
+ join (localURL:URL,bootstrapURL :URL):void
+ create (localURL:URL):void
+ retrieve (key:Key,callback :ChordCallback):void
+ insert (key:Key,entry :Serializable ,callback :ChordCallback):void
+ remove (key:Key,entry :Serializable ,callback :ChordCallback):void
+ leave ():void
<< interface >>
ChordCallback
+ retrieved (key:Key,entries :Set<Serializable> ,t:Throwable):void
+ inserted (key:Key,entry :Serializable ,t:Throwable):void
+ removed (key:Key,entry :Serializable ,t:Throwable):void
<< interface >>
Key
+ getBytes ():byte[]

Figure 6.2: CHORDSIPSERVLET class diagram

6.2.3 Using OpenChord

Deploying and using OpenChord is quite easy. First, the data that are going to be stored into the OpenChord DHT have to be defined. This is done by creating a Serializable class and overriding the `equals` and `hashCode` methods. The last part allows the user to determine his own way to compare data inside the DHT (i.e. the implementation only cares about the stored value and discards the TTL in those comparison operations).

Then, the format of the key that will be used to identify the data has to be defined. Again, this is done by implementing the `de.uniba.wiai.lspi.chord.service.Key` interface and overriding the `equals` and `hashCode` methods.

Finally, ChordImpl object has to be instantiated in order to allow the application to call its (in the context of an usage on the A5350 platform) asynchronous methods to access the DHT.

6.2.4 OpenChord callbacks

As the asynchronous API of OpenChord is used, callbacks are required. These callbacks are defined by OpenChord, they have to implement the `ChordCallback` interface. Their usage is similar to the usage of the callbacks in P2PSIPServlet except that they do not define a `failed` method; the call is considered as successful if the Throwable `t` (the last parameter of each method) is `null`, otherwise `t` provides more info about the error.

6.2.5 TTL handling

OpenChord does not provide any TTL support in its implementation, all data stored into the OpenChord DHT are there until the user explicitly removes them.

To deal with that issue, the remaining TTL is checked each time a value is retrieved from the DHT. If its TTL is equal to or less than zero it is removed from the DHT and not sent to the upper layer. The obvious drawback of such implementation is that an expired value will remain stored if nobody tries to retrieve it from the DHT. An additional mechanism automatically dealing with the removal of expired entries could easily be implemented by having these entries cleaned by each node at periodic intervals.

6.3 BambooSipServlet

The BambooSipServlet is a P2PSIP Servlet that relies on Bamboo.

6.3.1 Bamboo

Bamboo [13] is a DHT supporting low-latency under very high churn rates and reliable, high-performance storage with low get latencies. It was developed by Sean C. Rhea during his PhD thesis.

Bamboo is designed as a event-driven, single-threaded application and uses Staged, Event-Driven Architecture (SEDA). As described in [27], SEDA architecture splits the application into stages communicating through special objects called *events*. The Network, Router, StorageManager, DataManager, Vivaldi, Rpc are therefore stages (as it is visible on the class diagram since they derive from StandardStage, see Fig. 6.3). The ASyncCore interface and its implementation, the ASyncCoreImpl class, manage the communication between the stages. The DustDevil class is used to configure and initialize the several stages before starting the main loop.

6.3.2 Modifications made to Bamboo

Supporting TTL up to `MAXINT`

The vanilla version of Bamboo only supports TTL up to a week. In Bamboo, data are stored in different buckets regarding their TTL; by default, it defines 3 buckets: one for TTL less or equal to one hour, one for TTL less or equal to one day and one for TTL less or equal to one

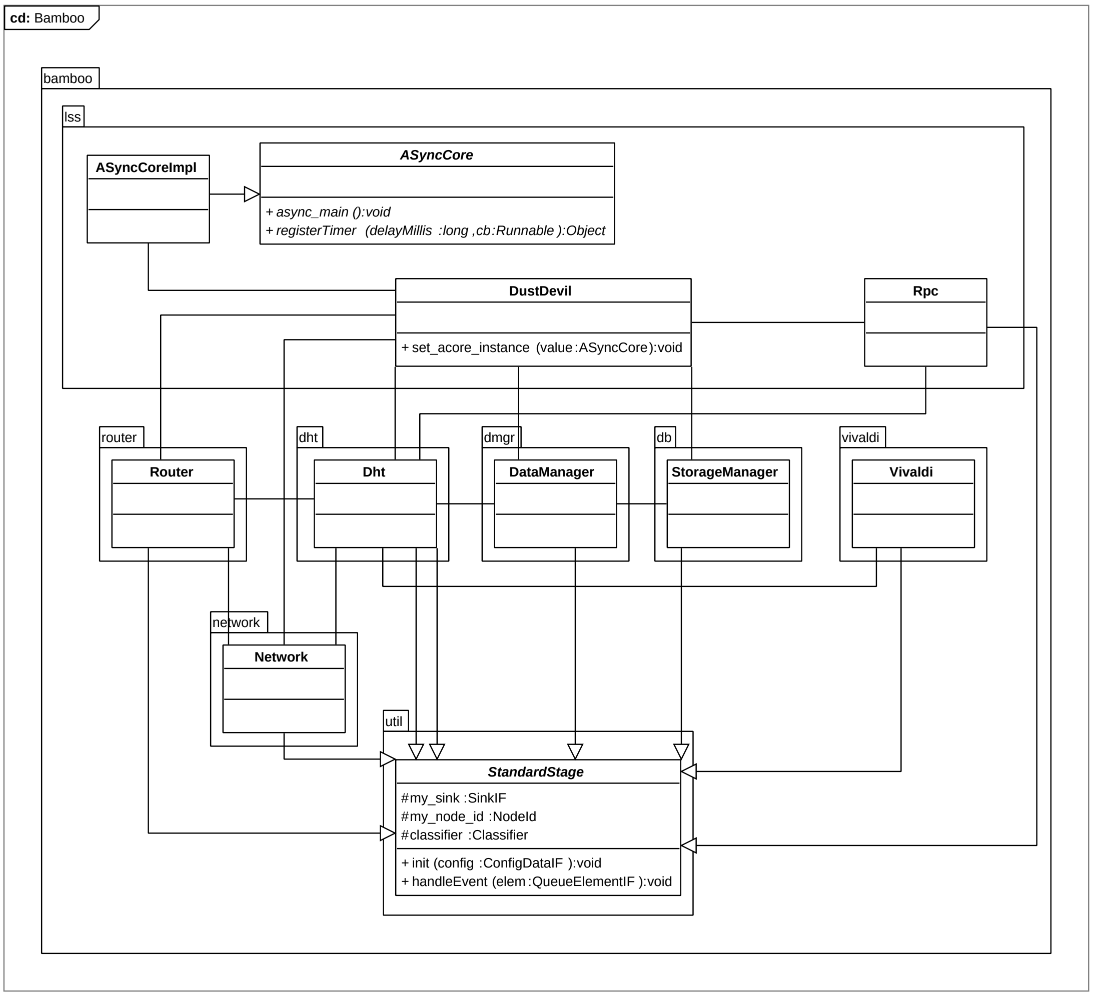


Figure 6.3: BAMBOO layer class diagram

week. In order to support bigger TTL, three more buckets are defined: one for TTL less or equal to a month, one for TTL less or equal to a year and one for other values of TTL (less or equal to `MAXINT`). The definition of these new buckets is a bit rough and could require some tuning to cope with real life scenarios and the fair share and fill rate limitations enforced by Bamboo.

Using customized *value hash*

In order to store contact address into the Bamboo DHT, support for customized *value hash* had to be added. By default, Bamboo recomputes the hash of the data when storing it into the DHT and when the user wants to remove data from the DHT, the hash provided must match the hash of the stored value. When a contact address is stored into the DHT, a string with various, and maybe unpredictable, parameters, is stored. To still be able to remove that data from the DHT (e.g. when the user unregisters a contact address) and since the removal process requires the correct hash of the stored value, Bamboo was changed to avoid having the *value hash* recomputed when added in the DHT, allowing the use of the hash of the stripped contact URI as *value hash*.

Allowing reduction of the TTL

Bamboo does not allow changing the TTL of a stored data to a lower value. This was changed in order to allow any TTL update.

6.3.3 BambooSipServlet

Initializing Bamboo

Since starting the Bamboo main loop is a blocking call, a special thread is started in the `join` method of the P2PSIP Servlet to avoid blocking the SIP container. To start Bamboo, this thread initializes a DustDevil instance to configure the several stages that are needed to provide the Bamboo DHT functionality and the customized Bamboo Stage (P2PSipStage) that will act as a gateway between the P2PSIP Servlet instance and Bamboo. Finally, this thread starts the Bamboo main loop.

BambooSipServlet class diagram

Fig. 6.4 presents the BambooSipServlet class diagram.

Communication between P2PSIP Servlet and Bamboo

As previously pointed out (cfr. Section 6.3.1), communication in Bamboo is achieved using events. The BambooSipServlet creates the event object corresponding to the wanted operation (`GetReq` to query the DHT and `PutReq` to store or remove data from the DHT). This event is then sent to the P2PSipStage instance and processed by the Bamboo layer.

Interactions with Bamboo are asynchronous, therefore when the BambooSipServlet instance wants to interact with Bamboo it creates a new BambooCallback object that will be used to handle the result of the operation and invoke the appropriate P2PSipCallback method. This

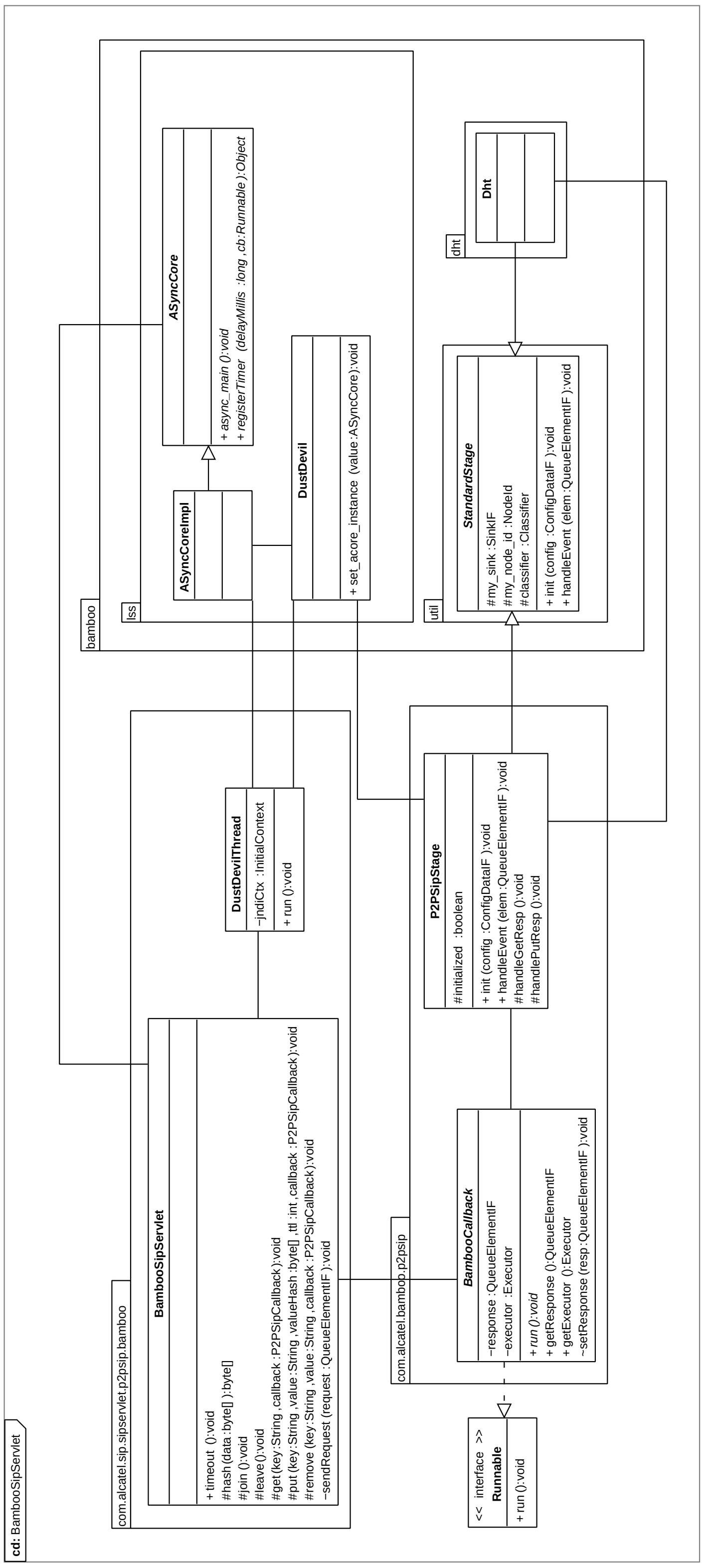

cd: BambooSipServlet
com.alcatel.sip.sipservlet.p2psip.bamboo
BambooSipServlet
+ timeout ():void
#hash (data :byte[]):byte[]
#join ():void
#leave():void
#get(key:String ,callback :P2PSipCallback):void
#put (key:String ,value:String ,valueHash :byte[] ,ttl :int ,callback :P2PSipCallback):void
#remove (key:String ,value:String ,callback :P2PSipCallback):void
~sendRequest (request :QueueElementIF):void
DustDevilThread
~jndiCtx :InitialContext
+ run ():void
bamboo
lss
ASyncCore
+ async_main ():void
+ registerTimer (delayMillis :long ,cb:Runnable):Object
ASyncCoreImpl
DustDevil
+ set_acore_instance (value:ASyncCore):void
dht
Dht
util
StandardStage
#my_sink :SinkIF
#my_node_id :NodeId
#classifier :Classifier
+ init (config :ConfigDataIF):void
+ handleEvent (elem:QueueElementIF):void
com.alcatel.bamboo.p2psip
P2PSipStage
#initialized :boolean
+ init (config :ConfigDataIF):void
+ handleEvent (elem:QueueElementIF):void
#handleGetResp ():void
#handlePutResp ():void
BambooCallback
~response :QueueElementIF
~executor :Executor
+ run ():void
+ getResponse ():QueueElementIF
+ getExecutor ():Executor
~setResponse (resp:QueueElementIF):void
<< interface >>
Runnable
+ run ():void

Figure 6.4: BAMBOOSIPSERVLET class diagram

BambooCallback object is passed as user data in the event object previously mentioned.

When the Bamboo layer has processed the event, it *answers* with another event (`GetResp` for a query event and `PutResp` for a storage or removal event). The answer is handled by the P2PSipStage instance which has subscribed to these kinds of events during its initialization phase. It then invokes the BambooCallback passed as user data which in turn invokes the P2PSipCallback, after having processed the result of the event, as described before.

6.4 SipDHT

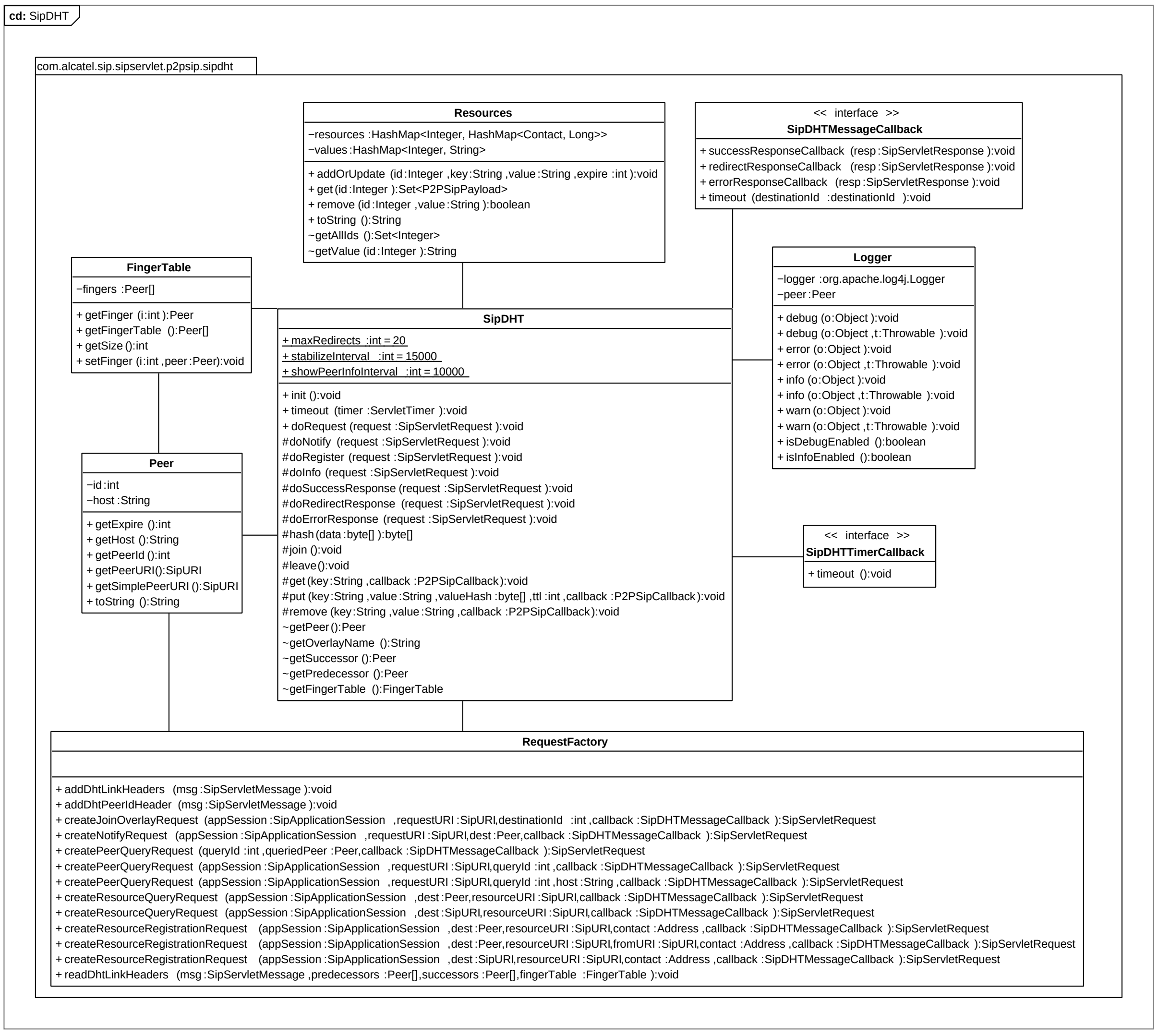

cd: SipDHT
com.alcatel.sip.sipservlet.p2psip.sipdht
Resources
−resources :HashMap<Integer, HashMap<Contact, Long>>
−values :HashMap<Integer, String>
+ addOrUpdate (id :Integer ,key :String ,value :String ,expire :int):void
+ get (id :Integer):Set<P2PSipPayload>
+ remove (id :Integer ,value :String):boolean
+ toString ():String
~getAllIds ():Set<Integer>
~getValue (id :Integer):String
<< interface >>
SipDHTMessageCallback
+ successResponseCallback (resp :SipServletResponse):void
+ redirectResponseCallback (resp :SipServletResponse):void
+ errorResponseCallback (resp :SipServletResponse):void
+ timeout (destinationId :destinationId):void
FingerTable
−fingers :Peer[]
+ getFinger (i:int):Peer
+ getFingerTable ():Peer[]
+ getSize ():int
+ setFinger (i:int ,peer :Peer):void
Logger
−logger :org.apache.log4j.Logger
−peer :Peer
+ debug (o:Object):void
+ debug (o:Object ,t:Throwable):void
+ error (o:Object):void
+ error (o:Object ,t:Throwable):void
+ info (o:Object):void
+ info (o:Object ,t:Throwable):void
+ warn (o:Object):void
+ warn (o:Object ,t:Throwable):void
+ isDebugEnabled ():boolean
+ isInfoEnabled ():boolean
SipDHT
+ maxRedirects :int = 20
+ stabilizeInterval :int = 15000
+ showPeerInfoInterval :int = 10000
+ init ():void
+ timeout (timer :ServletTimer):void
+ doRequest (request :SipServletRequest):void
#doNotify (request :SipServletRequest):void
#doRegister (request :SipServletRequest):void
#doInfo (request :SipServletRequest):void
#doSuccessResponse (request :SipServletRequest):void
#doRedirectResponse (request :SipServletRequest):void
#doErrorResponse (request :SipServletRequest):void
#hash (data :byte[]):byte[]
#join ():void
#leave ():void
#get (key :String ,callback :P2PSipCallback):void
#put (key :String ,value :String ,valueHash :byte[] ,ttl :int ,callback :P2PSipCallback):void
#remove (key :String ,value :String ,callback :P2PSipCallback):void
~getPeer ():Peer
~getOverlayName ():String
~getSuccessor ():Peer
~getPredecessor ():Peer
~getFingerTable ():FingerTable
Peer
−id :int
−host :String
+ getExpire ():int
+ getHost ():String
+ getPeerId ():int
+ getPeerURI():SipURI
+ getSimplePeerURI ():SipURI
+ toString ():String
<< interface >>
SipDHTTimerCallback
+ timeout ():void
RequestFactory
+ addDhtLinkHeaders (msg :SipServletMessage):void
+ addDhtPeerIdHeader (msg :SipServletMessage):void
+ createJoinOverlayRequest (appSession :SipApplicationSession ,requestURI :SipURI,destinationId :int ,callback :SipDHTMessageCallback):SipServletRequest
+ createNotifyRequest (appSession :SipApplicationSession ,requestURI :SipURI,dest :Peer,callback :SipDHTMessageCallback):SipServletRequest
+ createPeerQueryRequest (queryId :int ,queriedPeer :Peer,callback :SipDHTMessageCallback):SipServletRequest
+ createPeerQueryRequest (appSession :SipApplicationSession ,requestURI :SipURI,queryId :int ,callback :SipDHTMessageCallback):SipServletRequest
+ createPeerQueryRequest (appSession :SipApplicationSession ,requestURI :SipURI,queryId :int ,host :String ,callback :SipDHTMessageCallback):SipServletRequest
+ createResourceQueryRequest (appSession :SipApplicationSession ,dest :Peer,resourceURI :SipURI,callback :SipDHTMessageCallback):SipServletRequest
+ createResourceQueryRequest (appSession :SipApplicationSession ,dest :SipURI,resourceURI :SipURI,callback :SipDHTMessageCallback):SipServletRequest
+ createResourceRegistrationRequest (appSession :SipApplicationSession ,dest :Peer,resourceURI :SipURI,contact :Address ,callback :SipDHTMessageCallback):SipServletRequest
+ createResourceRegistrationRequest (appSession :SipApplicationSession ,dest :Peer,resourceURI :SipURI,fromURI :SipURI,contact :Address ,callback :SipDHTMessageCallback):SipServletRequest
+ createResourceRegistrationRequest (appSession :SipApplicationSession ,dest :SipURI,resourceURI :SipURI,contact :Address ,callback :SipDHTMessageCallback):SipServletRequest
+ readDhtLinkHeaders (msg :SipServletMessage ,predecessors :Peer[],successors :Peer[],fingerTable :FingerTable):void

Figure 6.5: SIPDHT class diagram

The SipDHT is a Chord DHT using SIP messages to communicate with the other peers in the overlay. It is based on David A. Bryan's draft [28]. This draft is still experimental and probably still presents some issues especially when handling peer failure.

The SipDHT uses both a SIP-based protocol for interactions with clients and for maintaining the DHT. In order to use the same SIP stack for both protocols and to handle easily and asynchronously SIP messages, the portion of the code managing the DHT has directly been written in the main SipDHT class. This way, the SipDHT implementation does not need any special mechanism for the interactions between the DHT and P2PSIP Servlet layers. The class diagram of the SipDHT is shown on Fig. 6.5.

6.4.1 Differences from Bryan's draft

Using `NOTIFY` requests

As pointed out on the P2PSIP mailing-list[1], the behavior of a stabilizing peer receiving another response than a `200 OK` to a `REGISTER` request acting like a Chord *notify* is still an open issue. In order to circumvent this issue, the SipDHT uses `NOTIFY` requests to perform the Chord *notify* procedure. A peer always handles and responds to correctly formed `NOTIFY` requests with a `200 OK` response.

Finding a fallback peer

When a peer finds that a peer failed to answer to a request, the peer updates its pointers that were using the failing peer (Section 9.8 of [28]). However, if it can not find a new peer to be used, it uses a *fallback* peer in a way similar to the one described in Section 9.3.1 of [28]: the first valid peer among the successors, the predecessor and the fingers is used.

Limitations due to implementation

In the current implementation some parts of [28] are not present:

- a peer does not handle the TTL information it has about the peers it had contacts with;
- the peers do not check the Peer-ID header validity in the received messages;
- each peer only supports one successor and does not replicate any data.

6.5 Other Java DHTs

This section describes other Java implementations of major DHT algorithms that have been considered to be used in order to create other P2PSIP Servlet implementations but were eventually disregarded.

[1]`https://lists.cs.columbia.edu/pipermail/p2p-sip/2006-November/001638.html`

6.5.1 FreePastry

FreePastry is an implementation of the Pastry DHT [9], it is developed by Microsoft Research and several universities in the USA. It provides an asynchronous API and therefore was worth investigating. More info can be found on their website: `http://freepastry.org/`. However, while starting to put together a PastrySipServlet, it appeared that FreePastry does not provide the functionalities needed to support the P2PSIP Servlet interface; namely:

No value removal support

Natively, FreePastry does not provide a way to easily remove a $(key, value)$ pair from the DHT. The authors justify this by security issues that removal support brings up. It can however be implemented by the end user/developer.

No multiple inserts support

If the user tries to insert values in the DHT with the same key, it fails. This appears to be a real problem when refreshing the TTL of a contact info stored in the DHT.

6.5.2 Tapestry

Tapestry is an algorithm developed by the CURRENT Lab for secure and reliable networking of the University of California, Santa Barbara that provides a decentralized directory service (like the Chord algorithm). According to [11], *"Tapestry is highly resilient under dynamic conditions, providing a near optimal success rate for requests under high churn rates, and quickly recovering from massive membership events in under a minute."*

It features several mechanisms to provide fault tolerance against hostile nodes:

- A Public Key Infrastructure (PKI) is used while creating node identifiers (to prevent Sybil attacks);
- Message Authentication Codes (MACs) are used to maintain integrity of overlay traffic;
- Monitoring system for maintaining neighbor links (reduce packet loss/improve message delivery in the overlay).

The team behind Tapestry is now concentrating its efforts on a new variant, developed in C, of the Tapestry protocol called *Chimera*.

Unfortunately, no suitable Java implementation could be found in order to put together a P2PSIP Servlet implementation using based on Tapestry.

6.5.3 JDHT

JDHT [29] is a simple Java based DHT which implements `java.util.Map`. However, the JDHT API is synchronous and would require additional efforts to make it usable within an asynchronous P2PSIP Servlet.

6.6 P2PProxyRegistrar

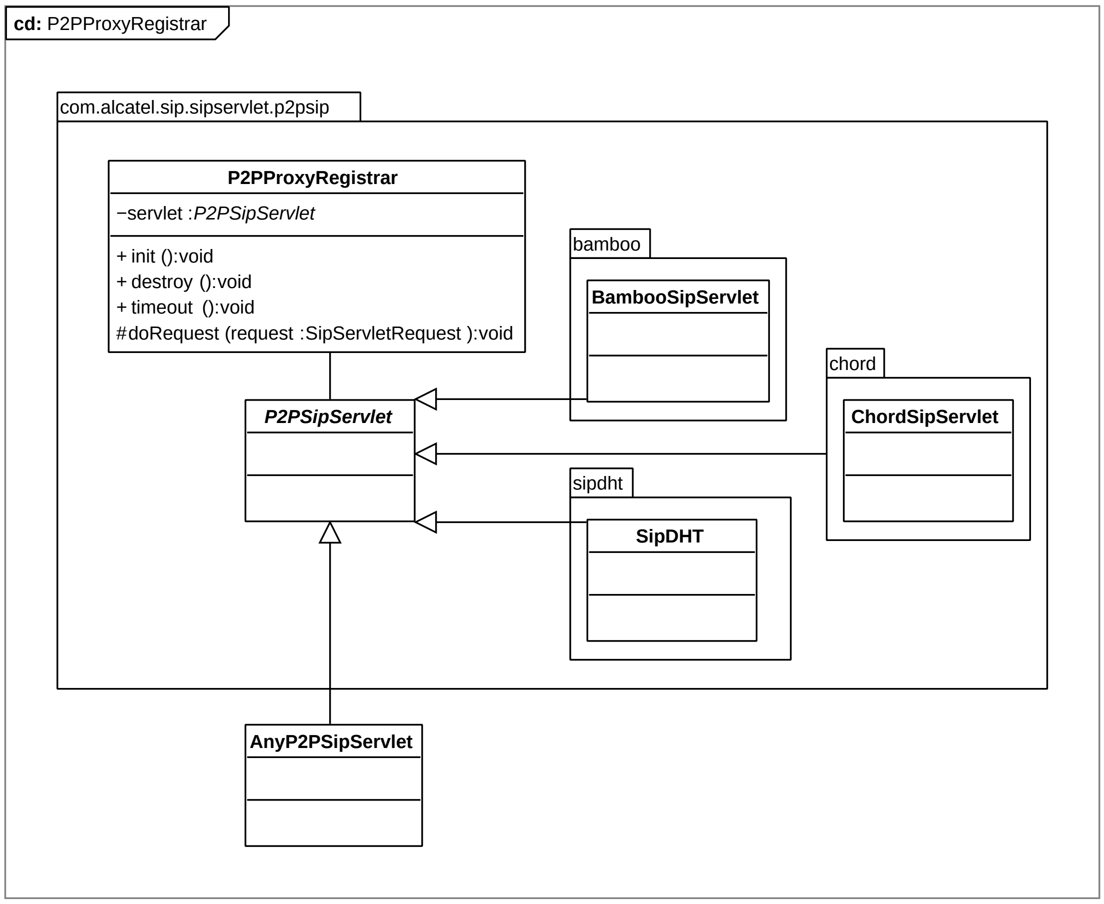


Figure 6.6: P2PPROXYREGISTRAR class diagram

The P2PProxyRegistrar SIP Servlet has been put together to ease the deployment on the A5350 platform of any of the previous implementations of the P2PSIP Servlet. It actually re-groups all the implementations within a single SIP Servlet and allows the user to choose the implementation to use in the instance.

It can support any P2PSIP Servlet implementation simply by adding the full qualified class name to the list of available implementations in a configuration file of the P2PProxyRegistrar SIP Servlet.

6.7 Suggested adaptations in the A5350

In order to support P2PSIP, a SIP Servlet should be able to listen on another port than the SIP port in order to communicate on an overlay channel and on a SIP channel simultaneously. Another, and probably more elegant, way to enable such a dual-channel communications system could be to create a P2PSipStack based upon the existing SipStack once a P2PSIP Peer protocol has been standardized.

6.8 Performance

This section presents the results of the different tests performed on the previously described implementations.

6.8.1 Testing environment

Several tests were performed, in order to determine the maximum load that the P2PSIP Servlet could support. Each test was based on two scenarios:

- a simple registration scenario: a *SIPp* [30] process simulates an UAC that successfully registers (without authentication) to the Registrar. This Registrar is either a specific node or the P2PSIP overlay. Fig. 6.7 details the callflow which is based on the one presented in subsection 2.1 of [31]. This scenario was run using a list of 150,000 distinct AORs.

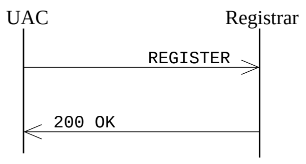


Figure 6.7: Registration scenario flow diagram

- a basic call scenario: a *SIPp* process simulates a call from an user to another user simulated by another *SIPp* process. This call is proxied through the distributed Proxy/Registrar. The callflow detailed on Fig. 6.8 is taken from subsection 4.2 of [32]. In this scenario, the caller was calling the same callee every time and the call was immediately terminated by the caller.

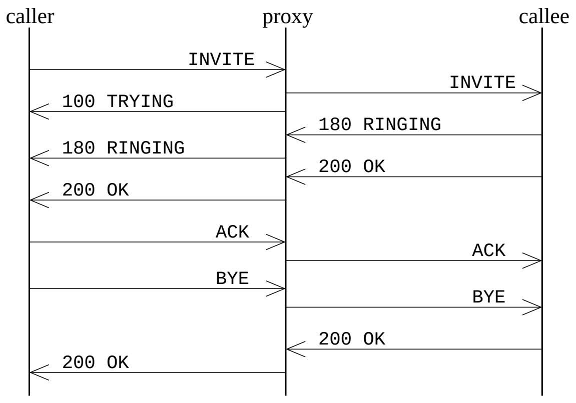


Figure 6.8: Call scenario flow diagram

Each test was performed on three configurations:

- Single server, up to several nodes
- Up to five servers, up to two nodes per server, no load balancing
- Up to five servers, up to two nodes per server, load balancing between active nodes

In the tests, unless otherwise mentioned, the P2PSIP Servlets were deployed on dual 3.06Ghz Intel Xeon server equipped with 3.6Gb of RAM; the SIP traffic was generated using *SIPp* from an other server equipped with two 3.4Ghz Intel Xeon CPUs and 3.6Gb of RAM. Also, all test servers were connected together on a private gigabit Ethernet LAN. The overlay was initialized previous to the tests and remained stable during their execution (no churn).

6.8.2 Deployment on a single server

In this test, the P2PSIP Servlets were deployed on the Alcatel-Lucent Proxy Platform in a dedicated group and on a single host. Also, a total of 2Gb of RAM has been dedicated to the deployed P2PSIP Servlets in each scenario and the requests were automatically load balanced on the nodes.

This test also compares the two implementations of P2PSIP Servlet with the default Proxy/-Registrar SIP Servlet that comes with the Proxy Platform.

Registration scenario

The results of the registration scenario test are presented on Fig. 6.9. The ChordSipServlet showed promising results, being able to deal with 1,050 REGISTER requests per second when a single node was deployed. Unfortunately, this value dropped to 270 REGISTER requests per second when a second node was added. This performance loss was likely due to the exchange of DHT maintenance messages between the nodes. It even dropped lower when more nodes were added but the decrease of performance was then mainly due to the overload of CPUs.

The BambooSipServlet showed poor performance (160 REGISTER requests per second) even decreasing (100 and 40 REGISTER requests per second) when adding nodes. However, this was not due to the CPUs being loaded. Actually the CPU load level of the nodes remained low but the processes were wasting time waiting for I/O accesses.

Since Bamboo uses an on-disk database to store the $(key, value)$ pairs managed by a node and in order to confirm that these disk accesses were the performance bottleneck, the same test was performed using a RAMDISK[2] as storage space for the database. This test exhibited way better results (790 REGISTER requests per second with one node and 690 with two nodes). The performance drop with four nodes is again explained by the overload of the CPUs.

[2]Virtual disk actually using a portion of the computer memory as storage; this greatly improves performance since RAM is a lot faster than a hard disk

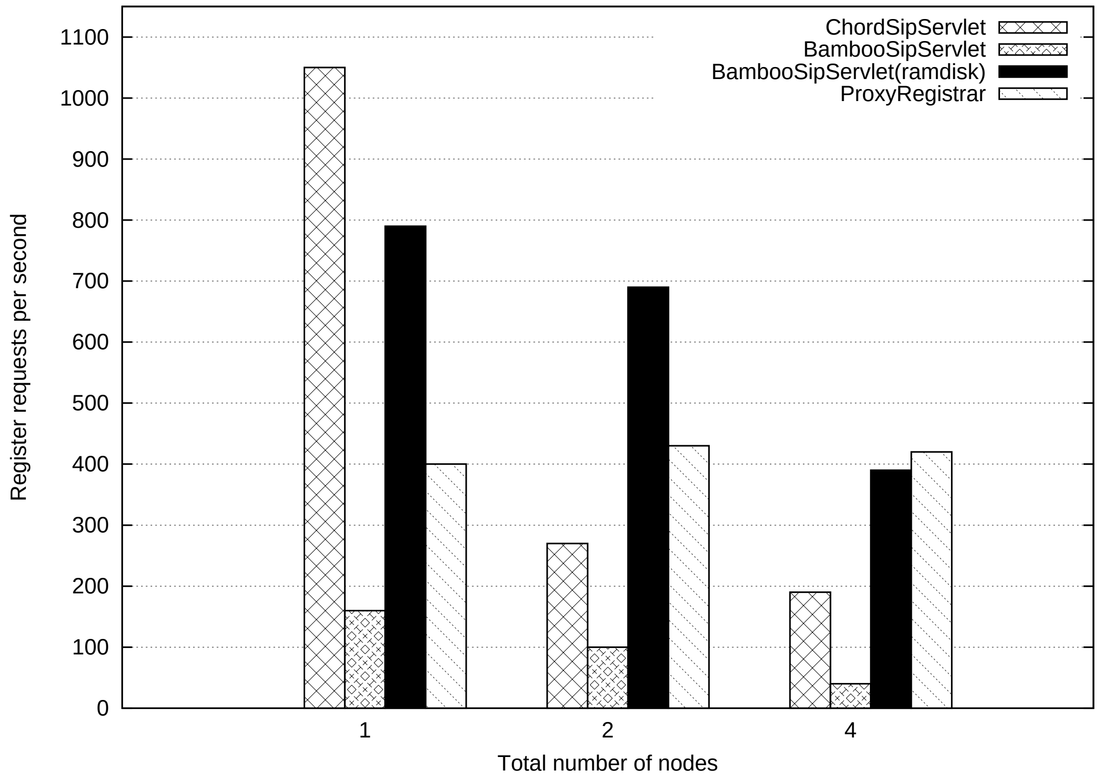


Figure 6.9: Single server deployment: registration scenario

Call scenario

In the call scenario (see Fig. 6.10), the ChordSipServlet started with very good results but dropped quickly as nodes were added and CPUs got loaded as in the registration scenario.

Bamboo showed better results, logically getting better with two nodes since the requests were load balanced between the two. Performance dropped a bit when four nodes were deployed because the CPUs got overloaded. This performance drop was a little quicker when the RAMDISK was not used, due to the concurrent accesses to the disk.

SipDHT

The P2PSIP Servlet implementing David A. Bryan's draft could not be tested as extensively as the two other implementations. This was mainly due to the lack of support for multi-agent deployment of the SipDHT P2PSIP Servlet. The multi-agent deployment is not possible because the SipDHT P2PSIP Servlet also uses the SIP stack to send its overlay messages. If these messages are sent in a multi-agent deployment the load-balancing policy of the SIP stack will forward the received message to one of the agents of the group, that one has a high probability of not being the recipient of the message. This limitation of the SipDHT implementation required it to be deployed alone in a group. The only test that could be performed was the registration scenario with only one deployed node. This test reached 1,100 REGISTER requests per second.

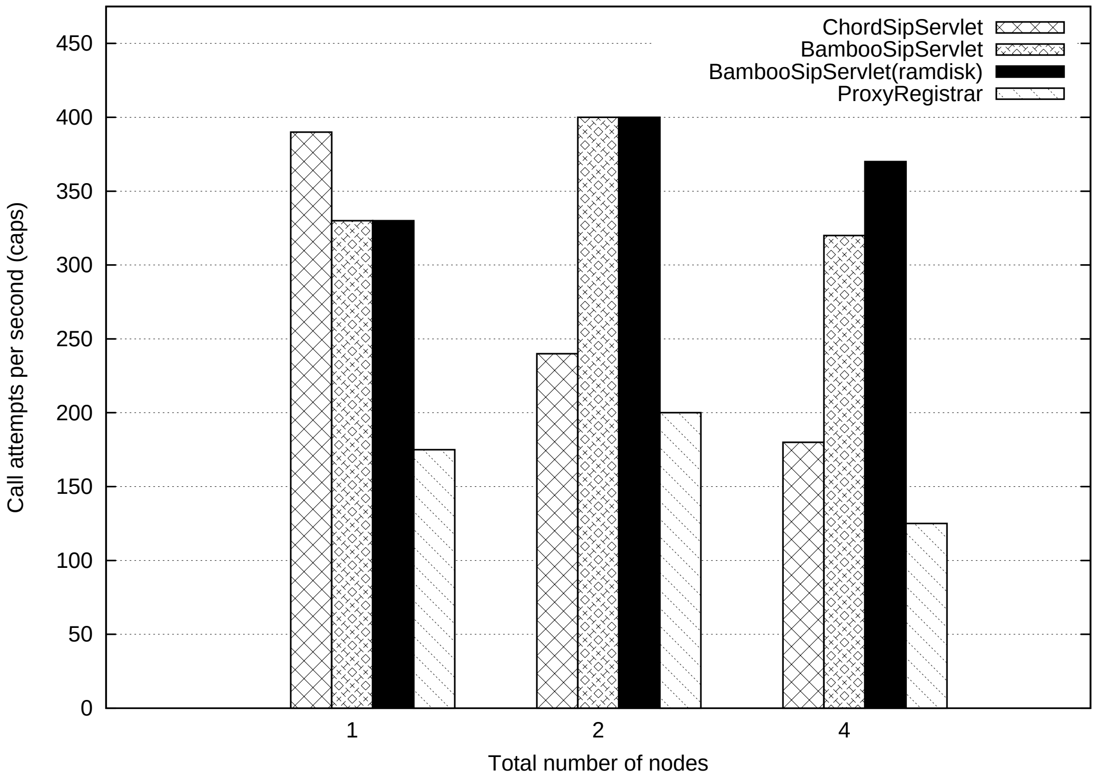


Figure 6.10: Single server deployment: call scenario

6.8.3 Distributed scenario

The next tests tried to show the influence of the number of nodes on the load a P2PSIP Servlet can deal with. The nodes were deployed on up to five different servers with one or two nodes per server. All the requests were sent to the same single node running during all tests.

Registration scenario

In the registration scenario, the ChordSipServlet once again started with very interesting results but dropped quickly as nodes were added. This test could not determine if the maximum capacity of a fully deployed Chord overlay was reached as the values kept dropping (see Fig. 6.11).

At the opposite, Bamboo quickly found its cruise rhythm oscillating between 100 and 200 REGISTER requests per second and almost 700 requests per second when using a RAMDISK.

Call scenario

As shown on Fig. 6.12, during the call scenario, the ChordSipServlet started again with a high number of call attempts per second (caps) but dramatically dropped when nodes were added. The reason for the drop to 70 caps could not be found in the SipServlet behavior and most likely lies within the OpenChord implementation.

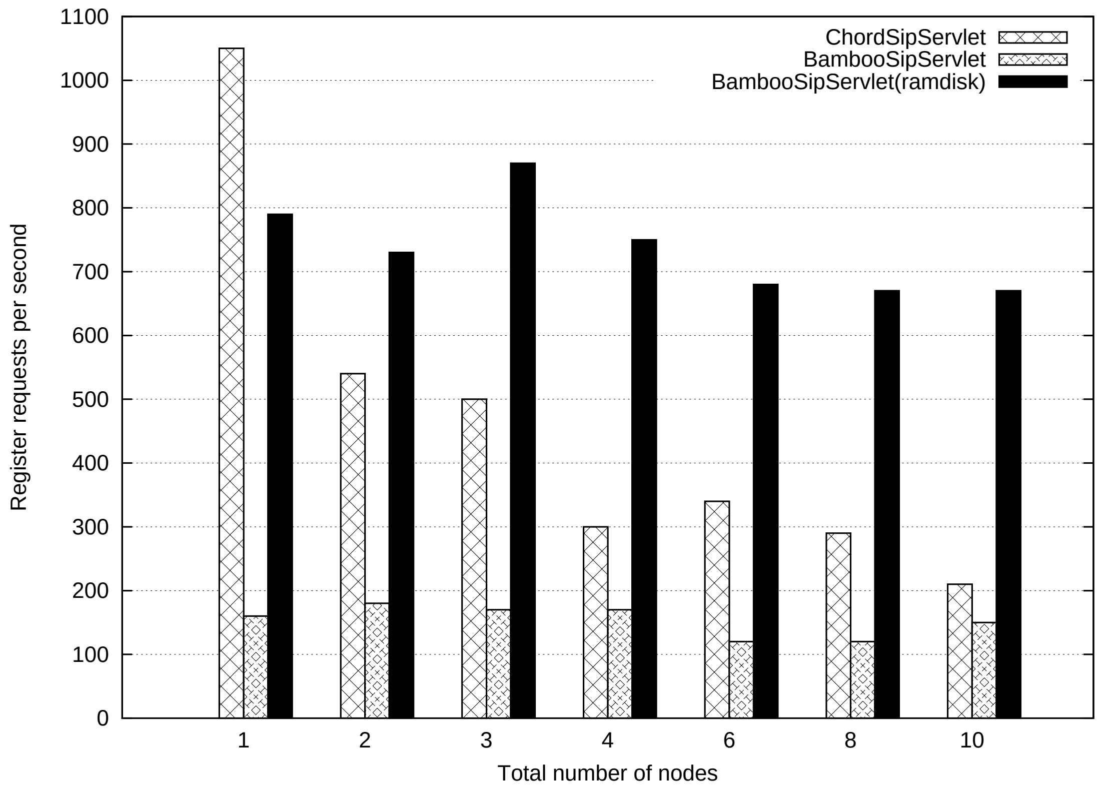


Figure 6.11: Influence of the amount of nodes: registration scenario

The BambooSipServlet, with or without using a RAMDISK, oscillated between 300 and 350 caps. The fact that it only reads one $(key, value)$ pair from the DHT in this scenario explains that the results are similar when using a RAMDISK or not.

6.8.4 Load balancing the requests

In these last tests, the nodes were deployed on the same servers as in the previous tests but now the requests were automatically load-balanced on all the running nodes by the SIP Stack component of the Proxy Platform.

Registration scenario

In the registration scenario shown on Fig. 6.13, the ChordSipServlet load capabilities dropped when a second node was added and were then slowly growing towards its initial rate. Unfortunately, no value for the deployment with more than four nodes could be measured because OpenChord crashed unexpectedly during each test and at various moments. The tests with two and four nodes were already unstable and the values observed should not be considered as very accurate. This erratic behavior of the ChordSipServlet seemed to be directly related to the OpenChord implementation. Apparently the stress put on the implementation was too high.

Once again due to the blocking I/O calls, the BambooSipServlet (without using a RAMDISK) produced poor performance oscillating between 150 and 210 REGISTER requests per second.

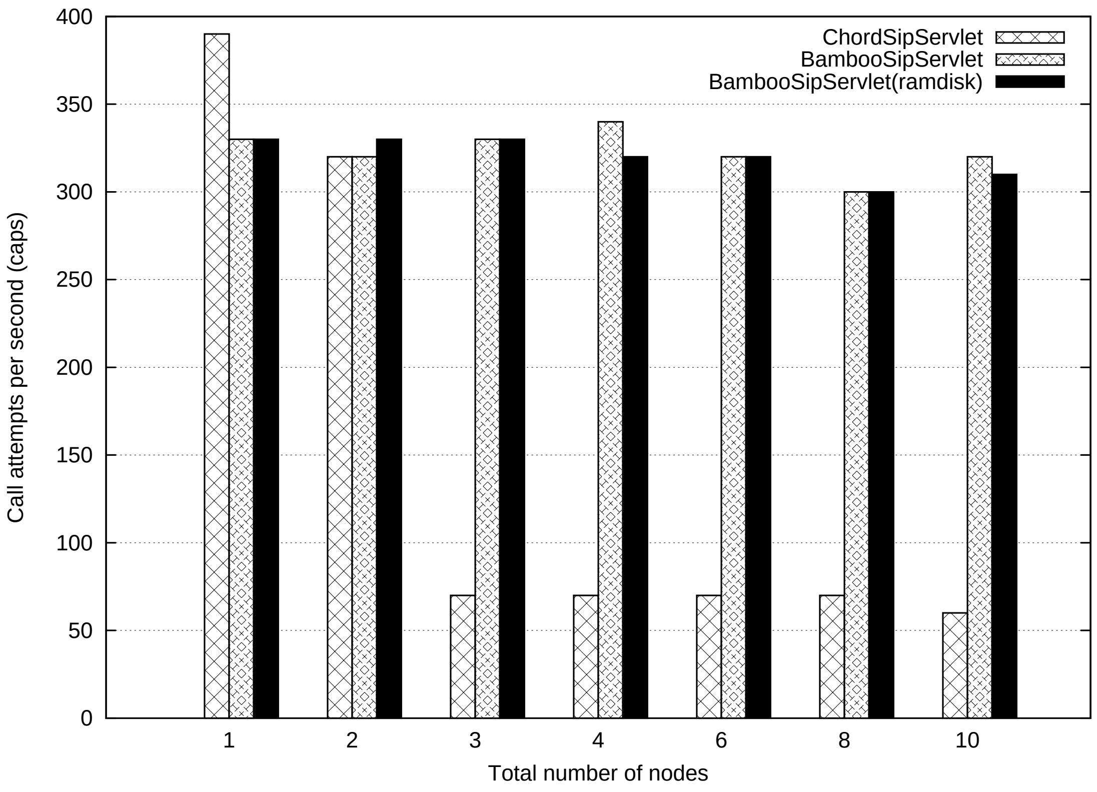


Figure 6.12: Influence of the amount of nodes: call scenario

This is really a bad behavior since the CPU usage is almost the same on every node. This means that the global CPU usage is rising while the performance remain at the same level, which leads to a waste of resources.

Finally, using the BambooSipServlet with a RAMDISK proves to be very efficient and to scale quite well.

Call scenario

During the call scenario (see Fig. 6.14), the three P2PSIP Servlets scaled correctly until more than four nodes were deployed. With more than four nodes per server, performance decreased differently with each implementation. The performance loss with more than four nodes deployed could be explained by different factors. The growing signaling traffic could result in increased latencies. The loss could also be due to the replication management layer of each DHT implementation. Moreover, the growing size of the DHT could result in a longer delay in retrieving the contact address of the callee from the DHT.

The ChordSipServlet dramatically dropped below 200 caps when six nodes were deployed and then turned out to cope with more caps than the other deployments afterwards. At first this could look like a testing error but the test with six nodes has been performed several times and each time with the same erratic behavior.

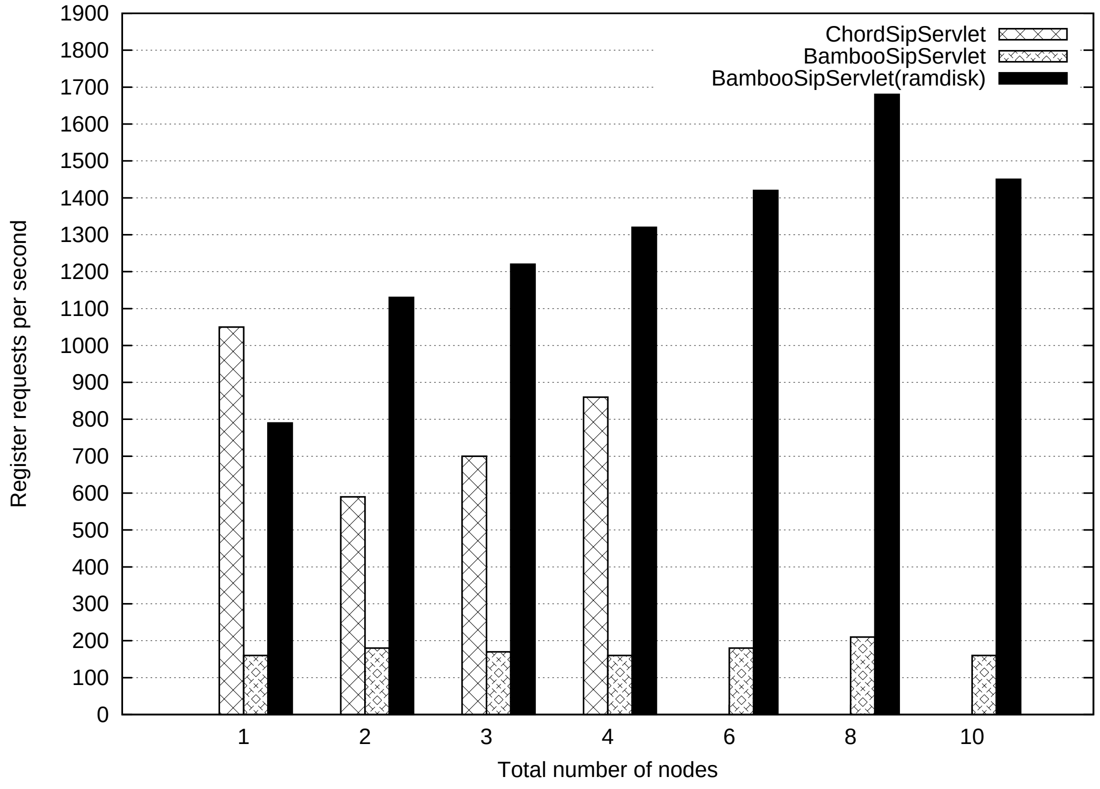


Figure 6.13: Load balancing: registration scenario

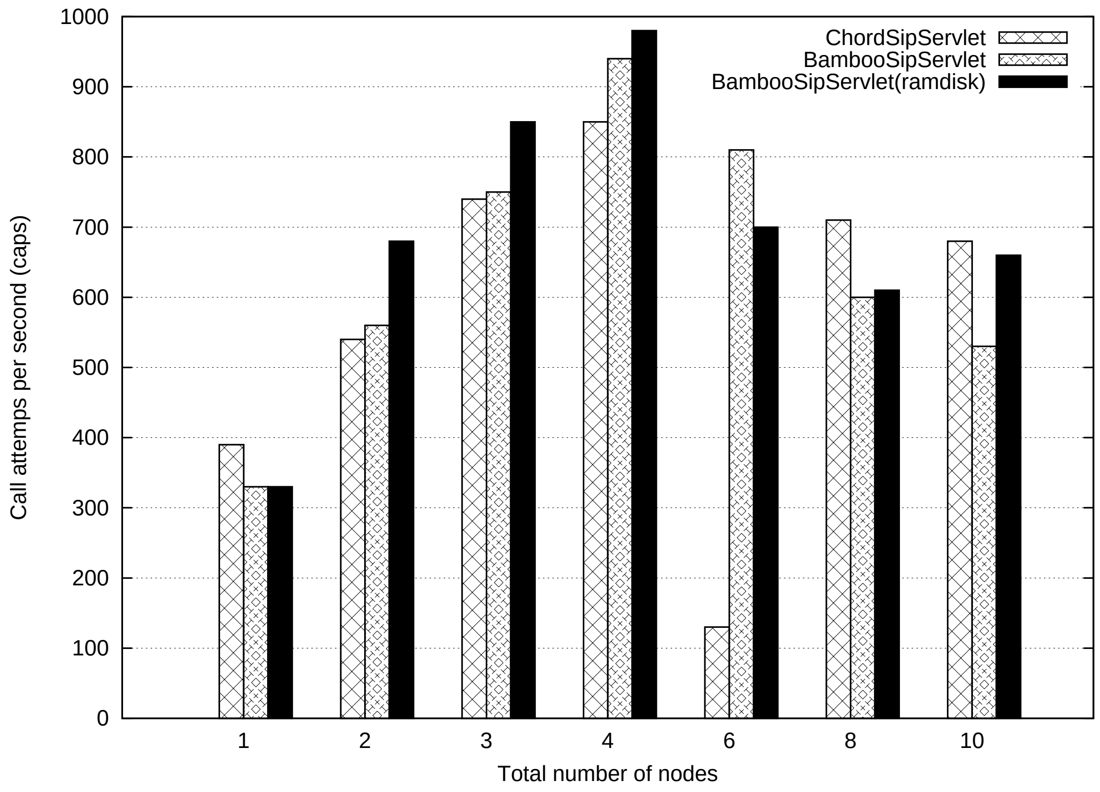


Figure 6.14: Load balancing: call scenario

6.8.5 Remark about limited deployment

A DHT is a distribution mechanism that is designed to be deployed on relatively many nodes (several hundreds or several thousands, sometimes even more). Here we deployed at most 10 nodes due to hardware limitations and the fact that each agent has to be manually configured. A deployment of many more nodes could show different results than those previously presented in this chapter. If different results happen, they could be explained by various factors.

In these small deployments, we can assume that in the majority of the scenarios each node is aware of the existence of the others (as successor, predecessor, in its routing table or in its finger table), and therefore message forwarding rarely happens.

On the other hand, replication mechanisms within the DHT implementation could put more load on some solicited nodes in a small deployment. If the implementation replicates the data on as many nodes as the overlay counts, the data could be replicated on the data root node itself. Also, the few number of nodes does not help with the key distribution, the same nodes are often solicited and then they replicate (if any replication system is used by the implementation) on the same nodes. The load distribution should be better with many more nodes.

6.8.6 Results interpretation

Table 6.1 summarizes the results of this section for the registration scenario with a specific set-up, namely two deployed nodes. As expected the distributed configuration with load-balancing achieves the best performance. Comparing DHTs Bamboo with RAMDISK doubles up the number of REGISTER requests processed with OpenChord.

	OpenChord	Bamboo	Bamboo (RAMDISK)
Single server	270	100	690
Distributed	540	180	730
Load-balanced	590	180	1130

Table 6.1: Summary of registration scenario results with two deployed nodes

In this section, the concept of a distributed Proxy/Registrar based on a DHT has been demonstrated, and promising performance results have been shown. With load-balancing enabled, results get even more interesting, such that P2PSIP turns out to be a real option for large, decentralized deployments.

However, the performance tests showed that some results could not be explained by the behavior of the P2PSIP Servlet but were related to the underlying implementation of the DHT. When OpenChord's performance dropped down or when it started to crash randomly, some limitations of the implementation could have been reached. This points out the fact that in order to get optimal performance, a P2PSIP implementation should have a tight control and a thorough knowledge of the DHT implementation it is relying on.

Chapter 7

Security in P2PSIP

This chapter covers the main security issues of P2PSIP but it does not pretend to be exhaustive; it also presents some solutions. All the security threats presented in this chapter have already been identified by the P2PSIP working group (see [33]). Some drafts of solutions are also taking shape (see [34]).

7.1 Security threats

This section describes the most common threats that a P2PSIP overlay should be addressing. Some of these threats have already been introduced in a DHT-only context in Section 3.10. They are now presented in a P2PSIP context and the protections against them can now make use of mechanisms introduced in a P2PSIP system.

7.1.1 Rogue node

A rogue node is a node that has nothing to do inside the overlay and, most of the time, whose intentions are hostile. If a single entity controls a lot of this type of node, this is known as a Sybil attack [18] (see Section 3.10.1). A P2PSIP overlay should therefore prevent such nodes to participate to the overlay.

7.1.2 Abusing node

Since participating to the overlay uses resources like bandwidth or memory space, some nodes could try to limit this usage and then not fairly participating to the overlay. Another abuse is to insert or lookup a lot of entries putting unnecessary stress on the overlay. The P2PSIP overlay should have some mechanism allowing it to stop such nodes to continue their activities.

7.1.3 Misinforming node

A node could get compromised and then start to abuse the system but it can be more vicious and hard to detect. A node can continue to act normally and just answer that the data does not exist to incoming requests. The P2PSIP overlay should implement a way to detect such a node and to block it.

7.1.4 Message modification

As a message can be relayed by one or more nodes before reaching its destination, any node on the way could alter its content. Also a transmission error or an wire interception could lead to changing the message content. A P2PSIP overlay should prevent such modifications or at least detect them.

7.1.5 Replay attacks

A replay attack consists of simply capturing a particular stream of data from a transmission and sending it again later. The captured data in such a scenario is often related to authentication, the attacker trying to get access to the system using the data previously exchanged by another user with the system.

7.1.6 Data access

In a P2PSIP overlay, the data will be distributed among the participating nodes. It means that any node will store information inserted by anyone that interacts with the system. This information cannot be removed or modified by anyone. In some use cases, the data should even only be readable by a specific user (or group of users).

7.2 Securing P2PSIP

This section proposes solutions to the previously pointed out security issues a P2PSIP overlay should be able to deal with.

7.2.1 Enrollment

The enrollment process is quickly mentioned in the working group charter [1]. Its role is to provide the node with the required information to join the overlay. This process will take place outside the overlay itself in a centralized manner. The enrollment operation will be needed only at regular interval or after loosing the credentials. This enrollment process can be configured to prevent arbitrary nodes to join the overlay.

The proposed solution is to provide a X.509 certificate [35] signed by the authority managing the overlay to each node. The enrollment process will also provide a pair of keys to each node, those keys will be used to secure its data in the overlay. In this system, the credentials are replaced by the X.509 certificate.

7.2.2 Credentials

As previously explained, the credentials are replaced by a X.509 certificate. The X.509 certificate system also comes with a revocation system allowing the authority to revoke a certificate, or the owner to revoke his/her own certificate.

This revocation support allows the managing authority to block any abusing node by revoking its credentials at any time. However, there will be a delay between the revocation and the actual blocking of the node because the other nodes check the revocation list at periodic intervals.

7.2.3 Data signature

The key pair provided by the enrollment mechanism can be used by the node to sign the data it inserts into the overlay. This enables any third-party retrieving the data to easily check its integrity.

7.2.4 Data encryption

If a given node needs to encrypt some data before storing it in the overlay, it can also use its key pair or the public key of any other node. Using some well known security protocols it can therefore store data in the overlay with the guarantee that only the recipient will be able to decrypt and read it.

This system could be extended to work with any public key and to the clients. In the case a client wants to store confidential information in the overlay, it only needs to be able to find the public key of the recipient, then it just has to delegate the storing operation to its associated peer.

7.2.5 Data replication

In order to counter potential compromised and misinforming nodes, a node executing a query should also query a replica of the requested information. The replica is created either by the node inserting the information or directly by the distribution system, the same goes for the removal. Then the node just needs to compare the results. If they are identical then the information is valid otherwise one of them is reporting false information.

This technique also enables a better fault tolerance of the distribution system. However it increases the latency for retrieving information from the overlay.

7.2.6 Timestamping

Adding a timestamp into each message (or at least significant messages) is a common technique to block replay attacks. Another well known technique is the usage of a nonce instead of a timestamp. One of these techniques should be used in a P2PSIP overlay to block replay attacks.

7.2.7 General remark

Implementation of cryptography and security is tricky. The best practice requires that the implementation of the previously suggested mechanisms uses well known, well tested and already existing implementations of the different algorithms.

7.3 Current state

As already mentioned in the introduction of the present chapter, the P2PSIP working group is aware of the security side of developing the P2PSIP protocols, several drafts have been written addressing different parts of the securing process (see [33, 34, 36]).

The working group is also aware of the previously developed issues but, in April 2007, no concrete solution has been proposed. Various discussions about the enrollment process and the issued credentials took place. These discussions reached some sort of consensus with the usage of a certificate signing the keys of the participating nodes but, even if X.509 was mentioned several times, the working group did not decide which technique should be used. Actually, since the working group Charter [1] states that "[i]*n order to simplify this problem* [security and privacy problems], *the WG will assume that all participants in the system are issued unique identities and credentials through some mechanism not in the scope of this working group, such as a centralized server, and that the data stored in the network will be authenticated by the storing entity in order to address the integrity issue and to some extent alleviate the DoS issue.*", the problem of how the enrollment process works and how to secure it will not be addressed by the P2PSIP working group.

Chapter 8

Conclusion

The IETF P2PSIP working group has chosen the Distributed Hash Table as basic structure for the distribution of data among the participating nodes. Therefore, the most common algorithms of Distributed Hash Tables have been presented and compared at the beginning of this thesis. Since the P2PSIP working group still has to choose one of these well-known DHT algorithms as base of the future P2PSIP Peer protocol, studying them was a prerequisite to the understanding of P2PSIP and its requirements.

The general techniques used in these DHTs were applied to the intra-cluster data sharing system of the Alcatel-Lucent A5350 platform allowing it to be become less dependent on a central component and to gain precious time when restarting that central component in a large deployment scenario. This step brought a better understanding of the internals of DHTs.

Then, the service discovery in a P2PSIP environment has been discussed and proved itself to be an important issue for the future of P2PSIP. It is required for its own operation in NAT environments as well as for the end-user willing to use a particular service. Even if this issue has yet to be addressed by the P2PSIP working group, a technique for indexing services has been introduced and is currently subject to a pending patent.

A prototype of distributed Proxy/Registrar supporting different DHT implementations has been developed and tested. This prototype pointed out some limitations of the A5350 platform when dealing with a P2PSIP scenario. The tests of the various implementations of this distributed Proxy/Registrar showed promising results when load-balancing the requests on the servers. Considering these results, P2PSIP turns out to be a real option for large, decentralized deployments. These tests also pointed out that in order to obtain the best performances, a P2PSIP implementations should have a tight control and a thorough knowledge of the DHT implementation it is relying on.

Finally, security of P2PSIP has been discussed. The main threats against a P2PSIP system have been presented along with some proposals for securing it against them.

This thesis explored the bases of P2PSIP: its distribution mechanism (a DHT) and, quickly, its security issues. It showed that the P2PSIP working group could keep its promise and bring an easy, cheap and scalable SIP-based network to the SIP community. However, there is still a lot of work to do before actually using it.

Future work

The working group would like to produce overview documents by July 2007 and the first peer protocol documents by March 2008: P2PSIP still has a long way to go before being ready for any public usage or even a first prototype providing a Skype-like service.

Lots of discussion happened on the P2PSIP mailing list but so far, May 2007, not much has been decided. The only consensus in the P2PSIP community is that the distribution will be based on a DHT and that a party outside the P2PSIP overlay will issue credentials (whose form is yet to be determined). The first decision the working group will have to take is whether the Peer protocol will be based on SIP or not. Then the actual work on securing the protocol itself will be able to take place. The NAT traversal mechanisms that will be deployed also depend on this decision.

Other issues such as defining the basic set of services provided by P2PSIP or allowing lawful interception have been discussed and will require some decision making in the near future. Finally, when P2PSIP is in a more mature state, connection with C/S SIP, interconnection of overlays and the deployment of common SIP-based services in a P2PSIP environment will require some work too.

Bibliography

[1] P2PSIP WG. *Draft Charter for the P2PSIP WG.* IETF, November 2006.

[2] David A. Bryan, P. Matthews, E. Shim, and D. Willis. Concepts and terminology for Peer-to-Peer SIP, October 2006. `http://www.p2psip.org/drafts/draft-willis-p2psip-concepts-03.html`.

[3] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. *RFC 3261 – SIP: Session Initiation Protocol.* IETF, June 2002.

[4] J. Rosenberg, C. Huitema, R. Mahy, and D. Wing. Simple Traversal Underneath Network Address Translators (NAT) (STUN), October 2006. `http://www.ietf.org/internet-drafts/draft-ietf-behave-rfc3489bis-05.txt`.

[5] J. Rosenberg, R. Mahy, and C. Huitema. Obtaining relay addresses from Simple Traversal Underneath NAT (STUN), October 2006. `http://www.ietf.org/internet-drafts/draft-ietf-behave-turn-02.txt`.

[6] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Methodology for Network Address Translator (NAT) Traversal for Offer/Answer Protocols, October 2006. `http://www.ietf.org/internet-drafts/draft-ietf-mmusic-ice-12.txt`.

[7] C. Jennings and R. Mahy. Managing Client Initiated Connections in the Session Initiation Protocol (SIP), November 2006. `http://www.ietf.org/internet-drafts/draft-ietf-sip-outbound-06.txt`.

[8] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-to-Peer lookup service for internet. In *ACM SIGCOMM 2001*, San Diego CA, August 2001.

[9] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale Peer-to-Peer systems, 2001.

[10] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable Content-Addressable Network. In *ACM SIGCOMM 2001*, San Diego, CA (USA), August 2001.

[11] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on selected areas in communications*, 22(1):41–53, January 2004.

[12] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer information system based on the XOR metric, March 2002.

[13] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, June 2004.

[14] Sean Christopher Rhea. *OpenDHT: A Public DHT Service*. PhD thesis, University of California, Berkeley, 2005.

[15] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proc. ACM SIGCOMM*, August 2003.

[16] Venugopalan Ramasubramanian and Emin Gün Sirer. Beehive: $O(1)$ lookup performance for power-law query distributions in Peer-to-Peer overlays, March 2004.

[17] George K. Zipf. *Human Behaviour and the Principle of Least-Effort*. Addison-Wesley, Cambridge MA, 1949.

[18] J. Douceur. The Sybil attack. In *Proc. of the IPTPS02 Workshop*, Cambridge, MA (USA), March 2002.

[19] Sriram Ramabhadan, Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. Prefix Hash Tree: an indexing data structure over Distributed Hash Tables. Technical report, Berkeley Intel Research, February 2004. `http://berkeley.intel-research.net/sylvia/pht.pdf`.

[20] James Aspnes and Gauri Shah. Skip graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, Baltimore, MD (USA), January 2003.

[21] William Pugh. Skip Lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[22] Baruch Awerbuch and Christian Scheideler. Peer-to-Peer systems for prefix search. In *ACM Symposium on Principles of Distributed Computing*, Boston, MA (USA), July 2003.

[23] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. Brief announcement: Prefix hash tree. In *Proceedings of ACM PODC*, St. Johns, Canada, July 2004.

[24] Donald Knuth. *The Art of Computer Programming*, volume Sorting and Searching, pages 492–512. Addison-Wesley, third edition, 1997. Section 6.3: Digital Searching.

[25] IANA. *Assigned Internet Protocol Numbers*. `http://www.iana.org/assignments/protocol-numbers`.

[26] Anders Kristensen. *SIP Servlet API Specification*. Dynamicsoft, February 2003. Version 1.0.

[27] Matthew David Welsh. *An Architecture for Highly Concurrent, Well-Conditioned Internet Services*. PhD thesis, University of California, Berkeley, 2002.

[28] David A. Bryan, Bruce B. Lowekamp, and Cullen Jennings. A P2P approach to SIP registration and resource location, October 2006. `http://www.p2psip.org/drafts/draft-bryan-sipping-p2p-03.html`.

[29] Jdht. `http://dks.sics.se/jdht/index.html`.

[30] SIPp. http://sipp.sourceforge.net.

[31] A. Johnston, S. Donovan, R. Sparks, C. Cunningham, and K. Summers. *RFC 3665 – Session Initiation Protocol (SIP) Basic Call Flow Examples*. IETF, December 2003.

[32] Henning Schulzrinne, Sankaran Narayanan, Jonathan Lennox, and Michael Doyle. Sipstone - benchmarking sip server performance. `http://www.sipstone.org`, April 2002.

[33] M. Matuszewski, J-E. Ekberg, and P. Laitinen. Security requirements in P2PSIP, February 2007. http://tools.ietf.org/wg/p2psip/draft-matuszewski-p2psip-security-requirements-00.txt.

[34] Cullen Jennings. Security mechanisms for Peer to Peer SIP, February 2007. http://tools.ietf.org/wg/p2psip/draft-jennings-p2psip-security-00.txt.

[35] R. Housley, W. Polk, W. Ford, and D. Solo. *RFC 3280 – Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. IETF, April 2002.

[36] B. Lowekamp and J. Deverick. Authenticated identity extensions to dSIP, February 2007. http://tools.ietf.org/wg/p2psip/draft-lowekamp-p2psip-dsip-security-00.txt.

Appendix A

Performance tests

A.1 Registrations

A.1.1 SIPP scenario

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<scenario name="Basic Sipstone UAC">
  <send retrans="10000">
    <![CDATA[

     REGISTER sip:ser.net SIP/2.0
     Via: SIP/2.0/[transport] [local_ip]:[local_port];
          branch=[branch]
     From: <sip:[field0]@[field1]>;tag=[call_number]
     To: <sip:[field0]@[field1]>
     Call-ID: [call_id]
     CSeq: 1 REGISTER
     Contact: sip:[field0]@[local_ip]:7060
     Content-Length: 0
     Expires: [field2]

    ]]>
  </send>

  <recv response="503" optional="true" next="1">
  </recv>

  <recv response="500" optional="true" next="1">
  </recv>

  <recv response="408" optional="true" next="1">
  </recv>
```

```
<!-- RECEIVE 200 OK -->
  <recv response="200" rtd="true">
  </recv>

<label id="1"/>

</scenario>
```

A.1.2 Example of values used

```
user1;ser.net;3600
user2;ser.net;3600
user3;ser.net;3600
   ⋮
user150000;ser.net;3600
```

A.2 Calls

A.2.1 SIPP scenarios

Registration

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<scenario name="register">
  <send>
    <![CDATA[
      REGISTER sip:alcatel.fr SIP/2.0
      Via: SIP/2.0/[transport] [local_ip]:[local_port];
           branch=[branch]
      From: bob <sip:bob@alcatel.fr>;tag=[call_number]
      To: bob <sip:bob@alcatel.fr>
      Call-ID: [call_id]
      CSeq: 1 REGISTER
      Contact: sip:bob@[local_ip]:7600;transport=[transport];
               expires=3600
      Max-Forwards: 70
      Content-length: [len]
    ]]>
  </send>

  <recv response="503" optional="true" next="1">
  </recv>
```

```
    <recv response="500" optional="true" next="1">
    </recv>

    <recv response="408" optional="true" next="1">
    </recv>

    <!-- RECEIVE 200 OK -->
    <recv response="200" rtd="true">
    </recv>

    <label id="1"/>
  </scenario>
```

Callin

```
  <?xml version="1.0" encoding="ISO-8859-1" ?>

  <scenario name="Basic UAS responder">
    <recv request="INVITE" crlf="true">
    </recv>

    <send>
      <![CDATA[

        SIP/2.0 180 Ringing
        [last_Via:]
        [last_From:]
        [last_To:];tag=[call_number]50
        [last_Call-ID:]
        [last_CSeq:]
        [last_Record-Route:]
        Contact: <sip:[local_ip]:[local_port];
                 transport=[transport]>
        Content-Length: 0

      ]]>
    </send>

    <send retrans="500">
      <![CDATA[

        SIP/2.0 200 OK
        [last_Via:]
        [last_From:]
        [last_To:];tag=[call_number]50
        [last_Call-ID:]
        [last_CSeq:]
        [last_Record-Route:]
        Contact: <sip:[local_ip]:[local_port];
                 transport=[transport]>
        Content-Type: application/sdp
        Content-Length: [len]

        v=0
        o=- 1995890563 1995890659 IN IP4 [local_ip]
        s=-
        c=IN IP4 [local_ip]
        t=0 0
        m=audio 8576 RTP/AVP 0
        a=rtpmap:0 PCMU/8000
        a=sendrecv

      ]]>
    </send>

    <recv request="ACK"
          optional="true"
          rtd="true"
          crlf="true">
    </recv>

    <recv request="BYE">
    </recv>

    <send>
      <![CDATA[

        SIP/2.0 200 OK
        [last_Via:]
        [last_From:]
        [last_To:];tag=[call_number]50
        [last_Call-ID:]
        [last_CSeq:]
        Contact: <sip:[local_ip]:[local_port];
                 transport=[transport]>
        Content-Length: 0

      ]]>
    </send>
  </scenario>
```

Callout

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<scenario name="Basic Sipstone UAC">
  <send retrans="5000">
    <![CDATA[

      INVITE sip:bob@alcatel.fr SIP/2.0
      Via: SIP/2.0/[transport] [local_ip]:[local_port];
           branch=[branch]
      From: alice <sip:alice@alcatel.fr>;tag=[call_number]
      To: bob <sip:bob@alcatel.fr>
      Call-ID: [call_id]
      Cseq: 1 INVITE
      Contact: sip:alice@[local_ip]:[local_port]
      Max-Forwards: 70
      Subject: P
      Content-Type: application/sdp
      Content-Length: [len]

      v=0
      o=- 1995890563 1995890659 IN IP4 [local_ip]
      s=-
      c=IN IP4 [local_ip]
      t=0 0
      m=audio 8576 RTP/AVP 0
      a=rtpmap:0 PCMU/8000
      a=sendrecv

    ]]>
  </send>

  <recv response="500" optional="true" next="19"> </recv>

  <recv response="408" optional="true" next="19"> </recv>

  <recv response="100" optional="true"> </recv>

  <recv response="180" optional="true"> </recv>

  <recv response="200" rtd="true" rrs="true">
        <action>
         <ereg regexp="&lt;(.*)&gt;" search_in="hdr"
          header="Contact: " assign_to="3,4"/>
 </action>
```

```
    </recv>

  <!-- SEND ACK -->
    <send>
      <![CDATA[
        ACK [$4] SIP/2.0
        Via: SIP/2.0/[transport] [local_ip]:[local_port];
             branch=[branch]
        Max-Forwards: 70
        [last_From:]
        [last_To:]
        Call-ID: [call_id]
        [routes]
        Contact: <sip:alice@[local_ip]:[local_port]>
        CSeq: 1 ACK
        Content-Length: 0

      ]]>
    </send>

    <send retrans="5000">
      <![CDATA[

        BYE [$4] SIP/2.0
        [last_Via]
        [routes]
        [last_From:]
        [last_To:]
        Call-ID: [call_id]
        Cseq: 1 BYE
        Max-Forwards: 70
        Content-Length: 0

      ]]>
    </send>

    <recv response="481" optional="true" next="19">
    </recv>

    <recv response="200" crlf="true">
    </recv>

    <label id="19"/>

    <!-- definition of the response time
```

```
        repartition table (unit is ms)   -->
    <ResponseTimeRepartition
       value="10, 20, 30, 40, 50, 100, 150, 200, 300,
              500, 1000, 2000"/>

    <!-- definition of the call length
         repartition table (unit is ms)     -->
    <CallLengthRepartition value="10, 50, 100, 500,
                                  1000, 5000, 10000"/>

  </scenario>
```

Appendix B

Article accepted by NGMAST’07

Implementation and Performance Evaluation of a P2PSIP Distributed Proxy/Registrar

Jean-François Wauthy
FUNDP - The University of Namur
jfwauthy@student.fundp.ac.be

Laurent Schumacher
FUNDP - The University of Namur
lsc@info.fundp.ac.be

Abstract

In this paper, the design of a distributed Proxy/Registrar based on Distributed Hash Tables is discussed. Three implementations, relying on OpenChord and Bamboo, are presented, and two of them are thoroughly tested in registration and call scenarios. In a distributed configuration with load-balancing, this distributed Proxy/Registrar manages to handle 1,500+ REGISTER requests per second and 700+ call attempts per second. These performance show that P2PSIP is a valid option for large, decentralized set-ups.

1. Introduction

The fast and still growing success of Skype showed that an (almost) fully decentralized VoIP system [1] is technically implementable and economically viable. That brought the SIP/IETF community to wonder whether it can come up with a similar successful, but open, protocol for Peer-to-Peer Internet Telephony based on the Session Initiation Protocol (SIP) [11] and simultaneously decentralize standard client/server SIP.

A new IETF Working Group (WG) has therefore been initiated for that project under the name "P2PSIP". P2PSIP will use a supernode-based architecture. Therefore a set of two protocols is currently under specification: the P2PSIP Peer protocol and the P2PSIP Client protocol, the second being a subset of the first. The P2PSIP Peer protocol will be used between the P2PSIP overlay peers, these peers are the ones that take part to the distribution of data. The P2PSIP Client protocol will be used by peers prevented from participating to the overlay due to computing and/or connectivity limitations (portable devices, disconnected environments, etc.).

This paper starts by presenting how data distribution works in a P2PSIP environment. Section 3 introduces the generic design of a distributed Proxy/Registrar as a SIP Servlet [4] and quickly lists the different implementations that have been realized so far. The performance tests of these Servlets and their results are presented and discussed in Section 4. Finally, the last section is devoted to conclusions.

2. Distribution in P2PSIP

Some argued that P2PSIP's main goals, namely distributing information among the nodes and decentralizing SIP, could be implemented using existing DNS extensions (such as mDNS, dynamic DNS, etc.). This has been (longly) discussed and due to some use cases, especially the ones involving disconnected environments, the forming WG clearly stated that P2PSIP would not be built upon DNS but would rather most likely be using a Distributed Hash Table (DHT) to distribute the information among the participating peers. This statement has been confirmed in the WG Charter [8] approved by the Internet Engineering Steering Group (IESG) at the end of February 2007 when the WG has been officially established. The WG will choose a specific DHT algorithm that will be required in any P2PSIP implementation but will allow additional algorithms, to be regarded as optional.

The main DHT algorithms are Chord [15], Pastry [12], CAN [9], Tapestry [16], Kademlia [6], Bamboo [10]. In their own way, they all map data identified by a key to a node participating to the DHT and maintain the structure of the DHT itself based on the nodes joining and leaving the network overlay. Some algorithms like Bamboo implement the data storage itself while others like Chord just handle the mapping. A short comparison of the major DHT algorithms is presented on Table 1 where N stands for the number of nodes in the DHT and d represents the size of the CAN hyperspace.

A DHT is similar to a hash table, the main difference being that the data is stored on the various nodes forming

Algorithm	Lookup performance	Pros	Cons
Chord	$O(\log N)$	simplicity	rely on application for replication
Pastry	$O(\log N)$	network locality	parameters to tune
CAN	$O(dN^{\frac{1}{d}})$	[redacted]	complex underlying geometry
Tapestry	$O(\log N)$	network locality	complicated design
Kademlia	$O(\log N)$	XOR symmetric metric	no explicit key removal
Bamboo	$O(\log N)$	enhancing Pastry	features to choose

Table 1. Comparison of major DHT algorithms

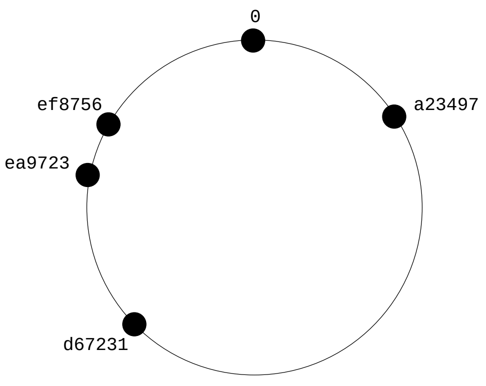


Figure 1. Distributed Hash Table (DHT)

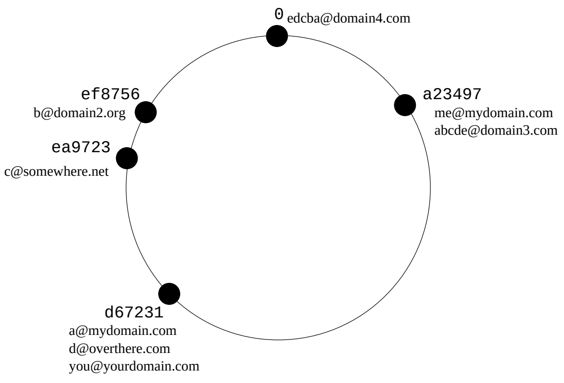


Figure 2. DHT usage in P2PSIP

the DHT overlay. An unique identifier is attributed to each node. These identifiers should be assigned as most uniformly as possible in the identifier space in order to efficiently distribute the information. Basically, a DHT stores $(key, value)$ pairs. The key is used to determine the pair identifier (usually computing a hash of the key); this pair identifier resides in the same space as the node identifier. The pair identifier is then used to determine on which node the pair should be stored. Also, a DHT has to cope with the arrival, departure or failure of a node. In each DHT algorithm, a node maintains information about some of its "neighbor" nodes in order to help inserting a new node into the overlay, to allow a node to leave or to detect failure and react accordingly. The fact that any node can leave the overlay without previous notification can lead to data loss. To avoid such loss, most DHT implementations provide some fault tolerance mechanisms. Fig. 1 shows a generic representation of a DHT identifier space.

The basic usage of a DHT in P2PSIP is to store the address bindings of an address-of-record (AOR), hence achieving the distribution of the Registrar and Location servers of standard client/server SIP set as one of the main goals of P2PSIP. Fig. 2 shows such usage on top of a generic DHT ring. Other usages of the DHT in P2PSIP are discussed too, such as storing voice mail, service information, etc.

3. P2PSIP Servlet

The P2PSIP Servlet is a SIP Servlet [4] providing the functionality of a distributed Proxy/Registrar. In such a distributed deployment, the client shall not be aware that the Proxy/Registrar is distributed. It has just to send its requests to one of the participating servers.

The proposed implementation is an abstract class that needs to be derived to support a specific DHT implementation. It has been required that it would support Java and be asynchronous, so as to fit in the testing environment of a third party. Given these requirements, a thorough evaluation of the existing implementations of the major DHT algorithms in view of their insertion into the P2PSIP Servlet class lead to the implementation and test of three instances. Those instances are mono-threaded and rely exclusively on asynchronous calls and callbacks.

The first one is based on OpenChord [7]. OpenChord is a Chord implementation that provides both a synchronous and an asynchronous Application Programming Interface (API), it supports generic keys and data and provides an application layer handling storage of the data. The second implementation uses Bamboo as DHT layer with some minor modifications regarding time-to-live (TTL) handling. Bamboo has been designed to support low-latency under very high churn rates and reliable, high-performance storage with low *get* latencies. The last P2PSIP Servlet implementation is based on David A. Bryan's draft [2] (which, itself, uses techniques from Chord) with some minor modifications when handling peer failure. However, only the OpenChord and Bamboo implementations are fully tested in the next section, as the third P2PSIP Servlet implementing David A. Bryan"s draft could not be fully deployed in the tested scenarios.

In this P2PSIP Servlet environment, the protocol used by the DHT implementation can be considered as the P2PSIP Peer protocol and SIP as the P2PSIP Client protocol. However, this comparison is not fully compliant with the WG Charter [8] since SIP is not a subset of the protocol used by the DHT implementation.

4. Performance evaluation

4.1. Testing environment

Several tests were performed, in order to determine the maximum load that the P2PSIP Servlet could support. Each test was based on two scenarios:

- a simple registration scenario: a *SIPp* [14] process simulates an UAC that successfully registers (without authentication) to the Registrar. This Registrar is either a specific node or the P2PSIP overlay. Fig. 3 details the callflow which is based on the one presented in subsection 2.1 of [3]. This scenario was run using a list of 150,000 distinct AORs.

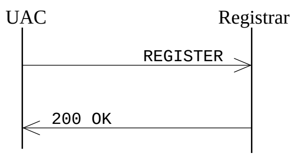


Figure 3. Registration scenario flow diagram

- a basic call scenario: a *SIPp* process simulates a call from an user to another user simulated by another *SIPp* process. This call is proxied through the distributed Proxy/Registrar. The callflow detailed on Fig. 4 is taken from subsection 4.2 of [13]. In this scenario, the caller was calling the same callee every time and the call was immediately terminated by the caller.

Each test was performed on three configurations:

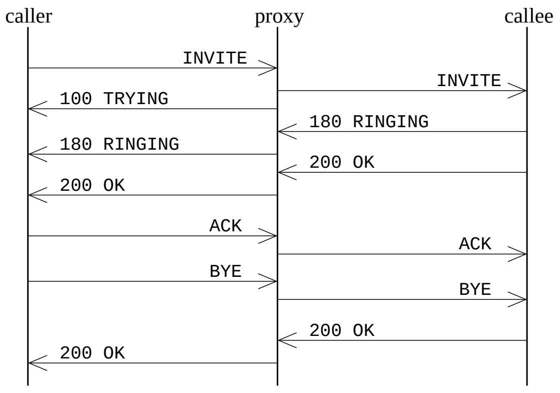


Figure 4. Call scenario flow diagram

- Single server, up to several nodes
- Up to five servers, up to two nodes per server, no load balancing
- Up to five servers, up to two nodes per server, load balancing between active nodes

In the tests, unless otherwise mentioned, the P2PSIP Servlets were deployed on dual 3.06Ghz Intel Xeon server equipped with 3.6Gb of RAM; the SIP traffic was generated using *SIPp* from an other server equipped with two 3.4Ghz Intel Xeon CPUs and 3.6Gb of RAM. Also, all test servers were connected together on a private gigabit Ethernet LAN. The overlay was initialized previous to the tests and remained stable during their execution (no churn).

4.2. Deployment on a single server

In this test, a total of 2Gb of RAM has been dedicated to the deployed P2PSIP Servlets in each scenario and the requests were automatically load balanced on the nodes.

The results of the registration scenario test are presented on Fig. 5. The ChordSipServlet showed promising results, being able to deal with 1,050 REGISTER requests per second when a single node was deployed. Unfortunately, this value dropped to 270 REGISTER requests per second when a second node was added. This performance loss was likely due to the exchange of DHT maintenance messages between the nodes. It even dropped lower when more nodes are added but the decrease of performance was then mainly due to the overload of CPUs.

The BambooSipServlet showed poor performance (160 REGISTER requests per second) even decreasing (100 and 40 REGISTER requests per second) when adding nodes. However, this was not due to the CPUs being loaded. Actually the CPU load level of the nodes remained low but the processes were wasting time waiting for I/O accesses.

Since Bamboo uses an on-disk database to store the $(key, value)$ pairs managed by a node and in order to confirm that these disk accesses were the performance bottleneck, the same test was performed using a RAMDISK[1] as storage space for the database. This test exhibited way better results (790 REGISTER requests per second with one node and 690 with two nodes). The performance drop with four nodes is again explained by the overload of the CPUs.

As already mentioned, the P2PSIP Servlet implementing David A. Bryan's draft could not be tested as extensively as the two other implementations. The only test that could be performed was the registration scenario with only one deployed node. This test reached 1,100 REGISTER requests per second.

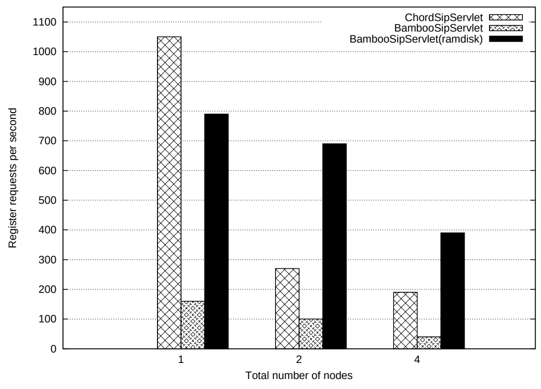


Figure 5. Single server deployment: registration scenario

In the call scenario (see Fig. 6), the ChordSipServlet started with very good results but dropped quickly as nodes were added and CPUs got loaded as in the registration scenario.

Bamboo showed better results, logically getting better with two nodes since the requests were load balanced between the two. Performance dropped a bit when four nodes were deployed because the CPUs got overloaded. This performance drop was a little quicker when the RAMDISK was not used, due to the concurrent accesses to the disk.

4.3. Distributed scenario

The next tests tried to show the influence of the number of nodes on the load a P2PSIP Servlet can deal with. The nodes were deployed on up to five different servers with one or two nodes per server. All the requests were sent to the same single node running during all tests.

[1]Virtual disk actually using a portion of the computer memory as storage; this greatly improves performance since RAM is a lot faster than a hard disk

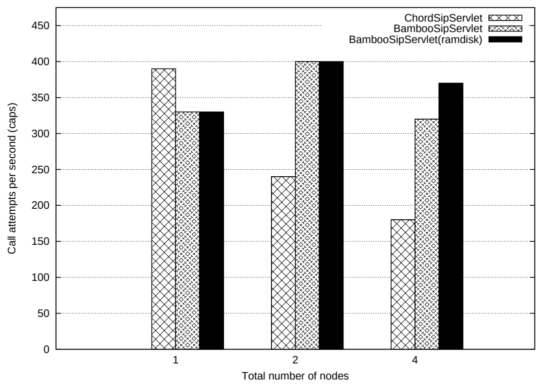


Figure 6. Single server deployment: call scenario

In the registration scenario, the ChordSipServlet once again started with very interesting results but dropped quickly as nodes were added. This test could not determine if the maximum capacity of a fully deployed Chord overlay was reached as the values kept dropping (see Fig. 7).

At the opposite, Bamboo quickly found its cruise rhythm oscillating between 100 and 200 REGISTER requests per second and almost 700 requests per second when using a RAMDISK.

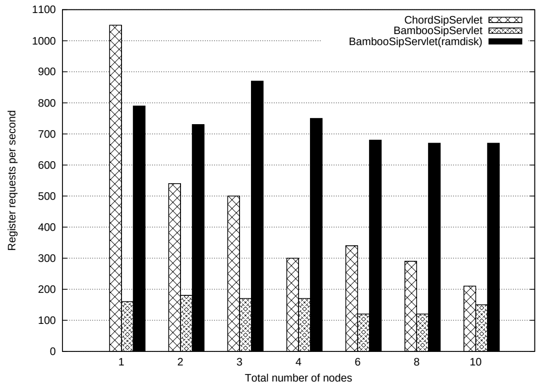


Figure 7. Influence of the amount of nodes: registration scenario

As shown on Fig. 8, during the call scenario, the ChordSipServlet started again with a high number of call attempts per second (caps) but dramatically dropped when nodes were added. The reason for the drop to 70 caps could not be found in the SipServlet behavior and most likely lies within the OpenChord implementation.

The BambooSipServlet, with or without using a RAMDISK, oscillated between 300 and 350 caps. The fact that it only reads one $(key, value)$ pair from the DHT in this scenario explains that the results are similar when using a RAMDISK or not.

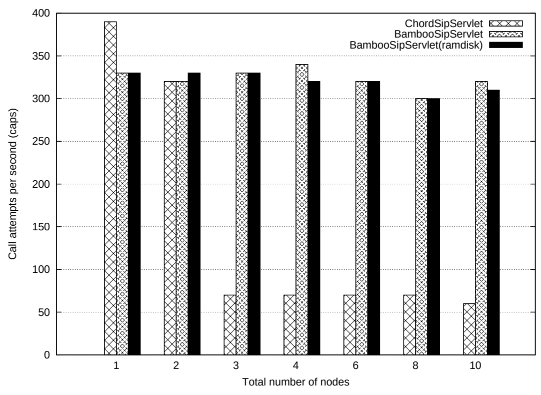


Figure 8. Influence of the amount of nodes: call scenario

4.4. Load balancing the requests

In these last tests, the nodes were deployed on the same servers as in the previous tests but now the requests were automatically load-balanced on all the running nodes.

In the registration scenario shown on Fig. 9, the ChordSipServlet load capabilities dropped when a second node was added and were then slowly growing towards its initial rate. Unfortunately, no value for the deployment with more than four nodes could be measured because OpenChord crashed unexpectedly during each test and at various moments. The tests with two and four nodes were already unstable and the values observed should not be considered as very accurate. This erratic behavior of the ChordSipServlet seemed to be directly related to the OpenChord implementation. Apparently the stress put on the implementation was too high. Forensic analysis of log files could not be realised due to a conflict with the log reporting environment. Moreover, switching to the official Chord implementation in C was not an option as the SipServlet should run asynchronously in Java.

Once again due to the blocking I/O calls, the BambooSipServlet (without using a RAMDISK) produced poor performance oscillating between 150 and 210 REGISTER requests per second. This is really a bad behavior since the CPU usage is almost the same on every node. This means that the global CPU usage is rising while the performance remain at the same level, which leads to a waste of resources.

Finally, using the BambooSipServlet with a RAMDISK proves to be very efficient and to scale quite well.

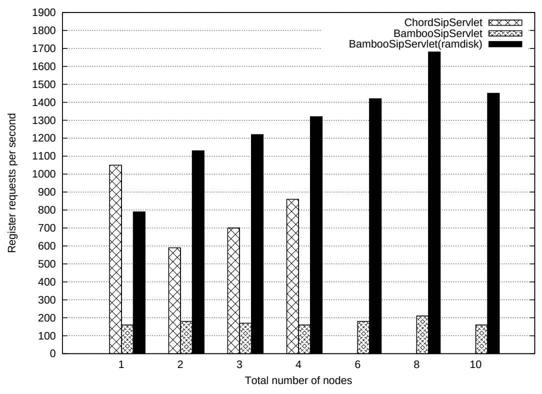


Figure 9. Load balancing: registration scenario

During the call scenario (see Fig. 10), the three P2PSIP Servlets scaled correctly until more than four nodes were deployed. With more than four nodes per server, performance decreased differently with each implementation. The performance loss with more than four nodes deployed could be explained by different factors. The growing signaling traffic could result in increased latencies. The loss could also be due to the replication management layer of each DHT implementation. Moreover, the growing size of the DHT could result in a longer delay in retrieving the contact address of the callee from the DHT.

The ChordSipServlet dramatically dropped below 200 caps when six nodes were deployed and then turned out to cope with more caps than the other deployments afterwards. At first this could look like a testing error but the test with six nodes has been performed several times and each time with the same erratic behavior.

Comparing results from the two distributed configurations, one can notice that the load-balancing enables to achieve performance results up to three times higher than without load-balancing for both the registration (compare Fig. 7 to Fig. 9) and the call (compare Fig. 8 to Fig. 10) scenarios. As the last configuration is the most realistic with respect to a real life deployment of a basic P2PSIP scenario (with only a few deployed nodes), P2PSIP hence appears as a real option for large, decentralized deployments.

Table 2 summarizes the results of this section for the registration scenario with a specific set-up, namely two deployed nodes. As expected the distributed configura-

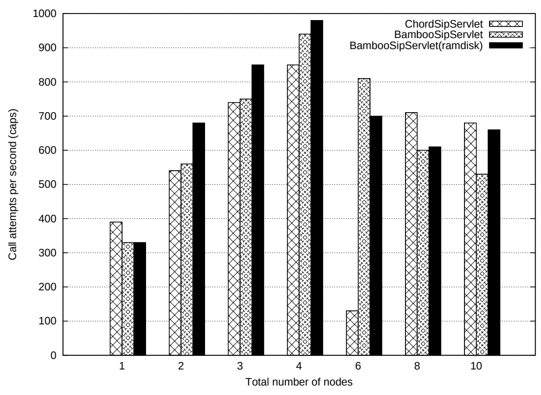


Figure 10. Load balancing: call scenario

tion with load-balancing achieves the best performance. Comparing DHTs Bamboo with RAMDISK doubles up the number of REGISTER requests processed with OpenChord.

	OpenChord	Bamboo	Bamboo (RAMDISK)
Single server	270	100	690
Distributed	540	180	730
Load-balanced	590	180	1130

Table 2. Summary of registration scenario results with two deployed nodes

5. Conclusion

In this paper, the concept of a distributed Proxy/Registrar based on a DHT has been demonstrated, and promising performance results have been shown. With load-balancing enabled, results get even more interesting, such that P2PSIP turns out to be a real option for large, decentralized deployments.

However, the performance tests showed that some results could not be explained by the behavior of the P2PSIP Servlet but were related to the underlying implementation of the DHT. When OpenChord's performance dropped down or when it started to crash randomly, some limitations of the implementation could have been reached. This points out the fact that in order to get optimal performance, a P2PSIP implementation should have a tight control and a thorough knowledge of the DHT implementation it is relying on.

Finally, since the distribution in P2PSIP makes use of storage and network bandwidth of the participating peers and since the messages can be relayed by one or more peers before reaching its final destination, there are security issues that need to be addressed. The main security issues are nicely summarized in [5]. But these issues are beyond the scope of this paper, which was only addressing the distribution system itself.

References

[1] S. A. Baset and H. Schulzrinne. An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol. Technical report, Department of Computer Science, Columbia University, September 2004.

[2] D. A. Bryan, B. B. Lowekamp, and C. Jennings. A P2P approach to SIP registration and resource location, October 2006. http://www.p2psip.org/drafts/draft-bryan-sipping-p2p-03.html.

[3] A. Johnston, S. Donovan, R. Sparks, C. Cunningham, and K. Summers. *RFC 3665 – Session Initiation Protocol (SIP) Basic Call Flow Examples*. IETF, December 2003.

[4] A. Kristensen. *SIP Servlet API Specification*. Dynamicsoft, February 2003. Version 1.0.

[5] M. Matuszewski, J.-E. Ekberg, and P. Laitinen. Security requirements in P2PSIP, February 2007. http://tools.ietf.org/wg/p2psip/draft-matuszewski-p2psip-security-requirements-00.txt.

[6] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-Peer information system based on the XOR metric, March 2002.

[7] OpenChord. http://www.uni-bamberg.de/en/-fakultaeten/wiai/faecher/informatik/lspi/bereich/research/-software_projects/openchord/.

[8] P2PSIP WG. *Draft Charter for the P2PSIP WG*. IETF, February 2007. http://www.ietf.org/html.charters/p2psip-charter.html.

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable Content-Addressable Network. In *ACM SIGCOMM 2001*, San Diego, CA (USA), August 2001.

[10] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, June 2004.

[11] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. *RFC 3261 – SIP: Session Initiation Protocol*. IETF, June 2002.

[12] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale Peer-to-Peer systems, 2001.

[13] H. Schulzrinne, S. Narayanan, J. Lennox, and M. Doyle. SIPstone – Benchmarking SIP Server Performance. http://www.sipstone.org, April 2002.

[14] SIPp. http://sipp.sourceforge.net.

[15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-to-Peer lookup service for internet. In *ACM SIGCOMM 2001*, San Diego CA, August 2001.

[16] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on selected areas in communications*, 22(1):41–53, January 2004.