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	$O(dN^{\frac{1}{d}})$		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

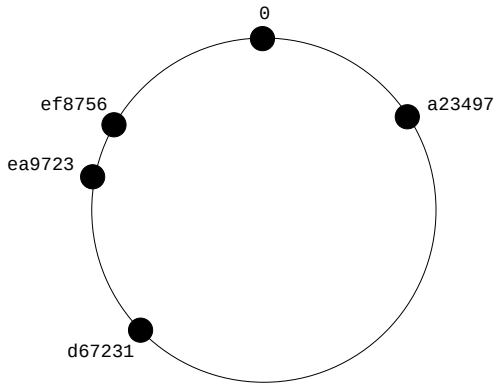

Figure 1. Distributed Hash Table (DHT)

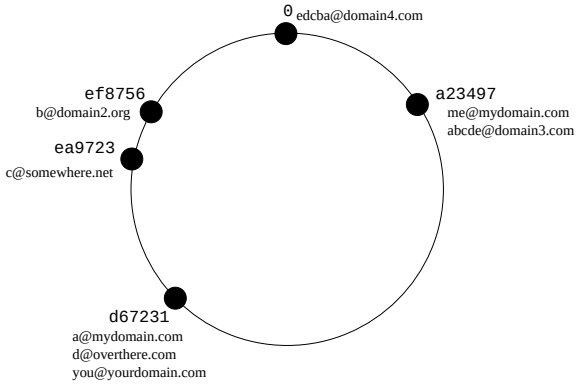

Figure 2. DHT usage in P2PSIP

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.

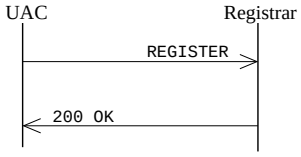

Figure 3. Registration scenario flow diagram

- 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:

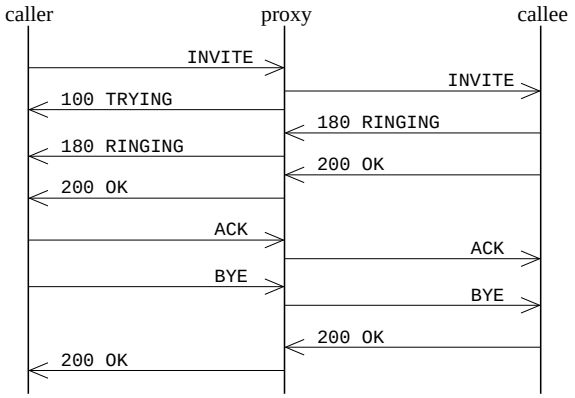

Figure 4. Call scenario flow diagram

- 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.

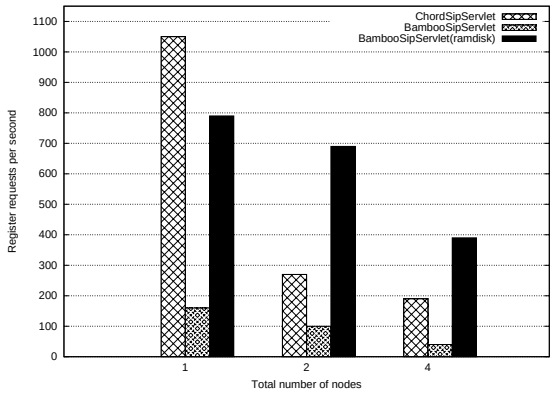

Figure 5. Single server deployment: registration scenario

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

[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

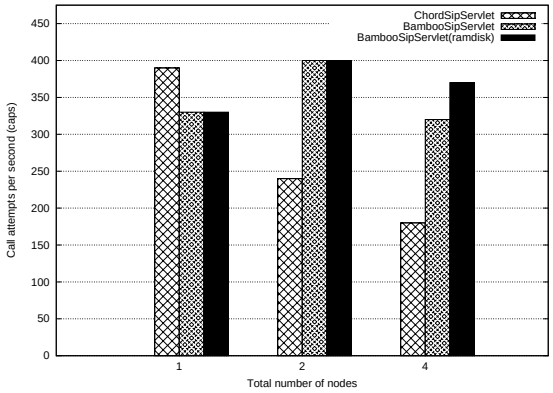

Figure 6. Single server deployment: call scenario

one or two nodes per server. All the requests were sent to the same single node running during all tests.

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.

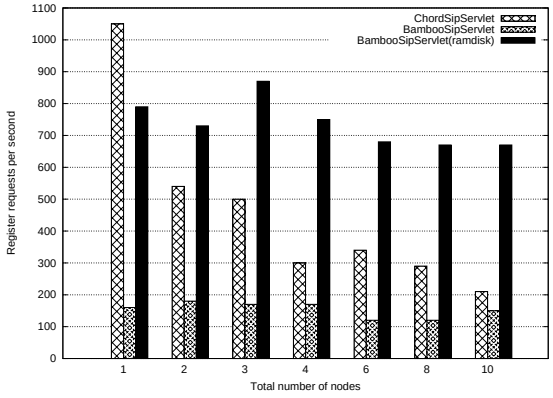

Figure 7. Influence of the amount of nodes: registration scenario

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.

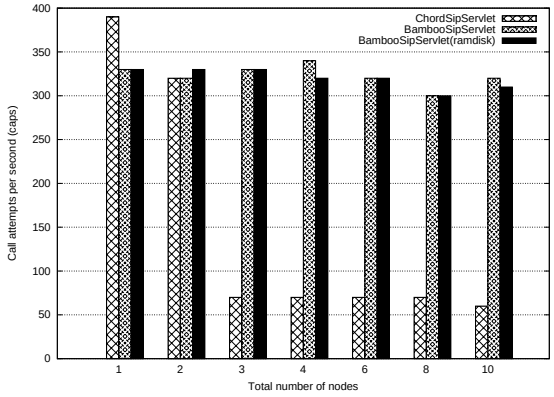

Figure 8. Influence of the amount of nodes: call scenario

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.

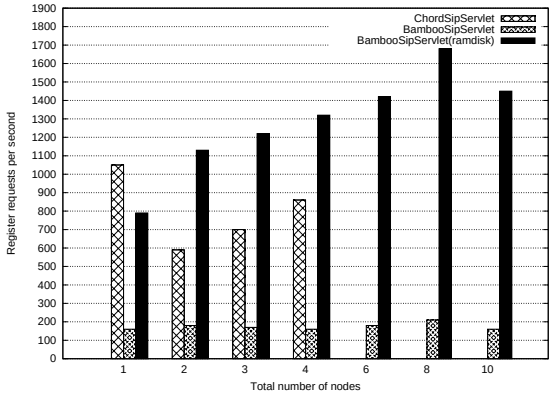

Figure 9. Load balancing: registration scenario

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-

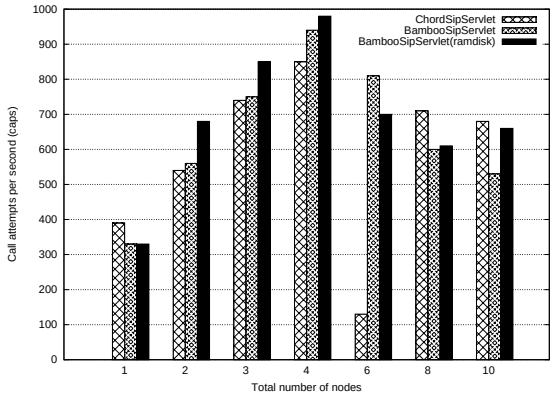

Figure 10. Load balancing: call scenario

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.