MSPastry protocol implementation in Peersim

Elisa Bisoffi, Manuel Cortella [⊥]

Master Degree in Computer Science
Project for the exam of Distributed System: Theory

Keywords. Pastry, MSPastry, Peersim, Large Scale Routing

Abstract. In this document we describe our implementation of MSPastry protocol in a Peersim environment. MSPastry is a large scale adaptable protocol for the routing of messages within a network, which has very good scalability properties. We created the MSPastry overlay and provided some basic primitives in order to allow at an application level to communicates with other applications based on MSPastry. Additionally we observed that the performance of the implementation is comparable to the theoretical results.

[⊥] elisa.bisoffi@studenti.unitn.it manuel.cortella@studenti.unitn.it

Table of Contents

1 Introduction

1.1 Overview

In this paper we are going to describe MSPastry implemented in Peersim, a java peer-to-peer simulator.
After a brief introduction to Pastry, MSPastry and Peersim, we will present a report about the implementation of our MSPastry library, avoiding to go too deep in implementation and coding details. After that a guide to the usage of the system will be presented, a short description of the set of primitives of the library and to how set the configuration. Finally some conclusions and statistical analysis of our work will conclude this document.

1.2 Pastry and MSPastry

According with [1], Pastry is a *scalable, distributed object location and routing substrate conceived for peer-to-peer networks.*
The purpose of this protocol consists in allowing the routing of messages (or objects) in very large networks.
Pastry was designed as a general substrate for many peer-to-peer environments, and is a self-organizing overlay, adaptable to failures.

MSPastry [2] comes from Pastry by re-designing the routing and maintenance techniques to improve Pastry original behaviour, by adding a consistent and reliable routing. MSPastry takes care of ensuring that messages are delivered to the node responsible for the destination key with high probability, according to the protocol, even in a churning or unstable environment; this outcomes in high dependability and good performance.

1.3 Peersim

Peersim [3, 4] is a java environment for the simulation of peer to peer protocols. It is extremely scalable and support dynamicity, configurability and is very flexible. Peersim supports both cycle-based and event-driven simulation models.

- In the cycle-based model nodes can communicates directly, and the control is assigned to each node in a cyclic manner, in order to process their operations.
- In the *event driven* model when it's generated an event (we can substitute the term event equivalently with the terms message, or Object), addressed to a particular protocol of a certain node, there is a method that takes care to handle incoming events/messages and processing them.

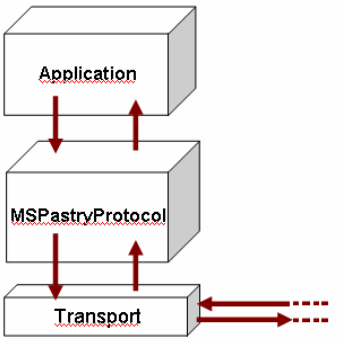
In order to develop our implementation of MSPastry protocol we adopted event-driven simulation model, because in a pastry network messages can be sent (and then received) in any unpredictable time instant, starting and coming from unpredictable nodes.

2 Implementation

From an implementational point of view, in the MSPastry version realized for the Peersim some functionalities were simplified with respect with the protocol specifications (like node failure detection). The following sections will describe how the system is implemented

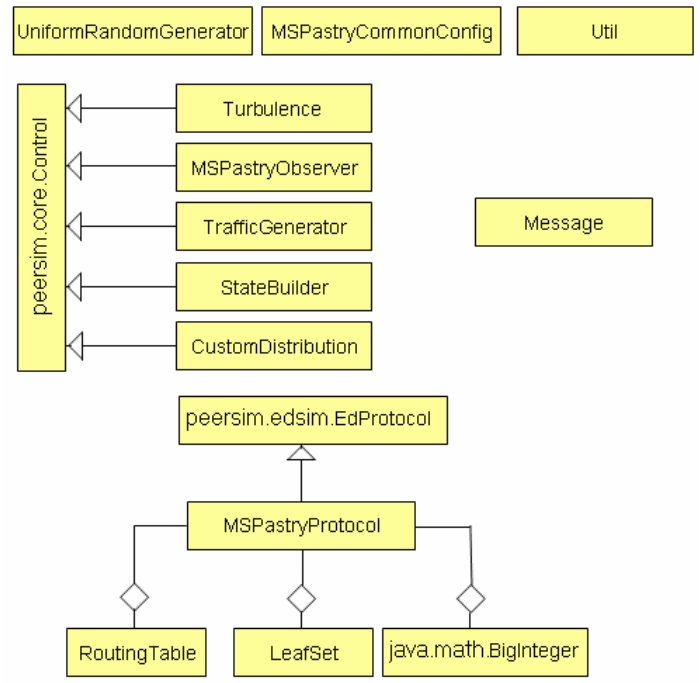
2.1 MSPastry protocol stack

MSPastry is a substrate between the Application level and the transport level, thus MSPastry protocol stack relies on a Transport protocol to work properly, as shown in the next picture:



2.2 Package Description

The next pictures illustrate the set of classes of the package `example.mspastry`.

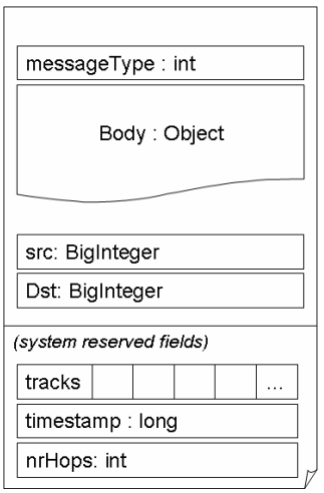


Now will be presented a short description of each class. We suggest to consult the *javadoc* references for more details.

Classes `UniforRandomGenerator`, `Util` and `MSPastryCommonConfig` have no special meaning in the package because are used only for support.

Class `Message` is a wrapper for a message. A message is intended for both Application message (also called Lookup message in the Pastry specification) and service message (routing, join requests, join reply, …) that are invisible to the application level. The body o a message is very generic, allowing to wrap every type of data. Message class besides provide some service information for statistical and debug analysis, not interesting for the application recipient.

Classes `Leafset`, `RoutingTable` are data wrappers containing and structures routines to maintain a pastry Leaf Set and Routing Table.



Class `MSPastryProtocol` inherits from `peersim.edsim.EDProtocol` and implements the MSPastry protocol specifications. We will discuss later about how the class works in order to implement message handling.

Classes `CustomDistribution` and `StateBuilder` e are the two inizializers for the pastry simulated network. CustomDistribution assigns unique random NodeIDs to the existing nodes of the simulation, by random peeking the values from the space $[0..2^{128}]$. Once the NodeIDs are stated, the initializator can start the work StateBuilder, by filling the initial routing tables, asserting that the consistency properties specified by the protocol are respected.

Class `Turbulence` is a Control that periodically perform the joining of anew node in the network, or the removing of and existing node from the network (thus simulating failure/departure of nodes).

Class `TrafficGenerator` is a Control that periodically generates some message traffic between pastry nodes. In practice it simulates a very simple Application level based on pastry, by randomly selecting two nodes and requesting the delivery of a message from the two endpoints, by relying on the routing pastry mechanism.

2.3 Setting up / Building of the system

Once Peersim has created the network (the set of the nodes), by reading from the configuration file the various settings (number of nodes, …), the nodes have to be initialized. Firstly every node need to have its own NodeID, with the properties specified by the pastry protocol. Class CustomDistribution is the initializator devoted for this job. It uses a random generator in the space $[0..2^{128})$ to generate unique NodeIDs to the nodes.

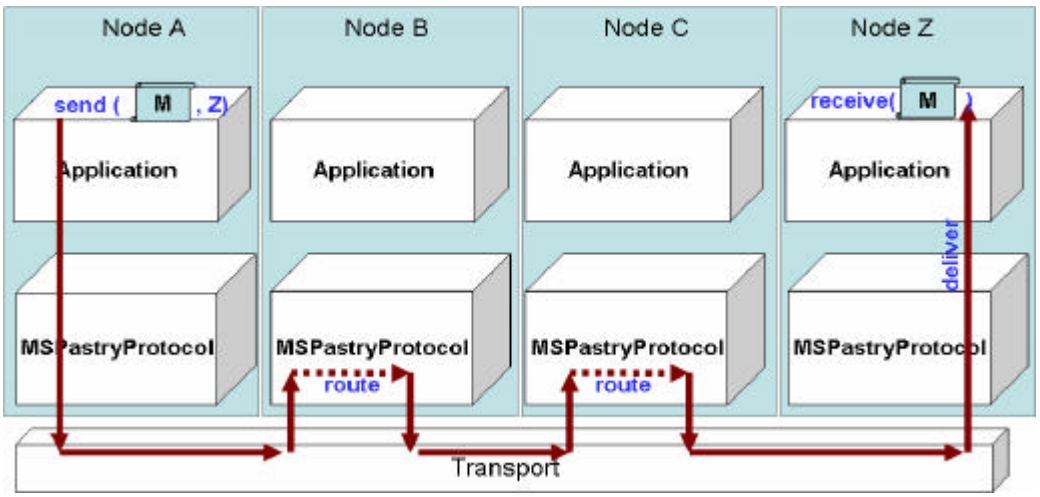
After every node has its NodeID, the tables (state) of the nodes must be initialized. This work is done by the StateBuilder. StateBuilder build the leaf set and routing table of every node, according to the properties of these data structure specified in the protocol.

2.4 Routing of Message

When a message arrives to a node, the node can decide whether route it again in the network or deliver it to the upper level. The route() method in the MSPastryProtocol class implement this functionality; the algorithm is the same described by the specifications (see [2]).

The main aim of the routine of a message consists *not* to deliver the message to the right recipient, but to deliver it to the node which the Node ID is numerically nearest than the specified destination.
In the MSPastry network messages with no really existing destination is supported: those messages, according to the protocol, are delivered in this way to avoid loops in the circulation of the messages.

A message is generated from the upper level protocol with respect to MSPastryProtocol layer from the source node. MSPastry get the sending command and route the message to the MSPastry node identified to be the next hop route path. The message travel among the path until it reaches the destination. At destination the MSPastryProtocol will take care of forward the message to the upper level. The next picture illustrate this situation:



2.5 Joining of a node

For the joining of a node, our implementation follows a slightly variation of the MSPastry specification. When a new node wants to join to the pastry network, a setup phase is preliminarily needed. The node get a Node Id, allowing that other nodes can refers to it, then the joiner must chose a node in the network and make to it the Join request.
The selection of the *joineer* from the joining node is made by peeking randomly ten possible candidates and electing the one which is the geographically nearest (i.e. with the lowest latency). Apart from this variation, the rest of the implementation follows exactly the pastry protocol specifications.

2.6 Failure /Departure

When a node fails, simply it status is set "`DOWN`". A node in state "`DOWN`" is not removed from the list of nodes of the `peersim.core.Network` class, but the Peersim system will not allow it to do any other action, including the receiving of messages/events.
In order to allow the active nodes are able to recognise other node failures, at regular time intervals is needed an update of their tables: in this way nodes avoid to send messages to inactive nodes.

3 Working with MSPastry

At application level MSPastry can be used in order to transport messages from one point (sender) to another (recipient). Every application based on MSPastry should handle both sending of messages and the receiving of messages from other nodes.
Our implementation provides a basic set of primitive actions in order to send a message or receive messages into Peersim.

3.1 Configuration file

MSPastry works by relying over a transport protocol. To set up a pastry network, first it is necessary to call the initializers `CustomDistribution` and `StateBuilder`.
The following picture illustrate a portion of the configuration file, showing the Protocol and the Initializers related to MSPastry:

```
#PROTOCOLS
protocol.p0_link peersim.core.IdleProtocol

[...]

protocol.p2_uniftr              peersim.transport.UniformRandomTransport
protocol.p2_uniftr.mindelay     10
protocol.p2_uniftr.maxdelay     100

protocol.p3_mspastry            example.mspastry.MSPastryProtocol
protocol.p3_mspastry.transport  p2_uniftr

[...]

# ::::: INITIALIZERS :::::

init.in1_uniqueNodeID           example.mspastry.CustomDistribution
init.in1_uniqueNodeID.protocol  p3_mspastry

init.in2_statebuilder           example.mspastry.StateBuilder
init.in2_statebuilder.protocol  p3_mspastry
init.in2_statebuilder.transport p2_uniftr
[...]
```

Specifying the `MSPastryProtocol` (also called `p3_mspastry` in the configuration file) we also indicates the `Transport` protocol that will be used.
The two initalizer, `CustomDistribution` and `StateBuilder` (also called `in1_uniqueNodeID` and `in2_statebuilder` in the configuration file) specify the protocol to initialize (`p3_mspastry`) and the transport layer that MSPastry uses.

3.2 Joining at application level

An existing node that is not yet connected to the network have to make a *join* request. These few instructions make a join request. The joining operations works according with the protocol specifications.

```
Node myNode = …;
MSPastryProtocol myPastry = myNode.getPid(pastryid);

myPastry.join();
```

3.3 Sending messages at application level

Application based on pastry can send a message simply by using the send() primitive of the protocol, or equivalently by putting the message in the event queue of peersim with the parameters described below.

```
Node myNode = …;
```

```
MSPastryProtocol myPastry = myNode.getPid(pastryid);
…
Message m = new Message(“hello world”);
BigInteger dest = …;
myPastry.send(dest, message);
```

3.4 Handling with message receiving at application level

An event handling system is available to manage the receiving of the messages from other node. The Application can set a specific event handler to pastry that is activated each time a Lookup message is received from the node at the pastry level.

```
Node myNode = …;
MSPastryProtocol myPastry = myNode.getPid(pastryid);
…
myPastry.setListener(
    new MSPastryProtocol.Listener() {

            public void receive(Message m) {
              Object data = m.body;
              System.out.println(“received message < “ +
                                 data.toString() +
                                 ” > from address: ”+ m.src);
            }

     }
);
```

3.5 Departure of a Node

Failure / departure of nodes is automatically handled by our implementation. When a Peersim Node is set down (or dead), the Peersim system automatically will ignore that node. No external service operation are needed (our system supports brutal departure of nodes, and adapt itself by consequence).

3.6 Example

The following steps show how to run an experiment with MSPastry. The experiment consists of a set of nodes that are initialized. Then an entity (`TrafficGenerator` control) in the system provide the generation of random traffic between nodes, and another component provides some type of turbulence in the network, by adding and removing nodes periodically (`Turbulence` control).

The configuration file is here shown:

```
# ::::::::::::::::::::::::::::::::::::::::::::::::::::::
# :: MS Pastry Default Configuration
# ::                13.37 27/05/2007
# ::::::::::::::::::::::::::::::::::::::::::::::::::::::

# ::::: GLOBAL ::::::

SIZE 5000
K 5

MINDELAY  500
MAXDELAY  900

TRAFFIC_STEP 1000
OBSERVER_STEP 2000
TURBULENCE_STEP 4000


# ::::: network :::::
random.seed 24680

simulation.experiments 1

simulation.endtime 1000*60*5
```

```
network.size SIZE


# ::::: LAYERS :::::
protocol.0link peersim.core.IdleProtocol

protocol.1uniftr peersim.transport.UniformRandomTransport
protocol.1uniftr.mindelay MINDELAY
protocol.1uniftr.maxdelay MAXDELAY


protocol.2unreltr peersim.transport.UnreliableTransport
protocol.2unreltr.drop 0
protocol.2unreltr.transport 1uniftr

protocol.3mspastry example.mspastry.MSPastryProtocol
protocol.3mspastry.transport 2unreltr


# ::::: INITIALIZERS :::::
init.0randlink peersim.dynamics.WireKOut
init.0randlink.k K
init.0randlink.protocol 0link


init.1uniqueNodeID example.mspastry.CustomDistribution
init.1uniqueNodeID.protocol 3mspastry

init.2statebuilder example.mspastry.StateBuilder
init.2statebuilder.protocol 3mspastry
init.2statebuilder.transport 2unreltr


# ::::: CONTROLS :::::
control.0traffic example.mspastry.TrafficGenerator
control.0traffic.protocol 3mspastry
control.0traffic.step TRAFFIC_STEP

control.2turbolenceAdd example.mspastry.Turbulence
control.2turbolenceAdd.protocol 3mspastry
control.2turbolenceAdd.transport 2unreltr
control.2turbolenceAdd.step TURBULENCE_STEP


# ::::: OBSERVER :::::
control.3 example.mspastry.MSPastryObserver
control.3.protocol 3mspastry
control.3.step OBSERVER_STEP
```

We simulate a Network of N=5000 nodes[1], running for 300'000 time instants.
In debug mode (when `Util.debug` is set to `true`), a verbose output is put on the standard output and on the standard error, allowing to "trace" what's happening internally, in particular to see the path a message makes from the source to its destination.
After starting Peersim, random traffic and churning of the network results in various statistical outcomes of the times a message requires to reach its destination, and the number of hops it makes. In the following chapter is presented an analysis of these statistics. The two controls and the observer run over the MSPastry layer.

3.7 Reference Platforms

This work was realized by using a JBuilder 2006 Environment.
The experiments have been run by using two notebooks:

- Intel P4 1600 MHz, with 256 MB RAM, jdk 1.6.0.1
- Intel P4 1700 MHz, with 256 MB RAM, jdk 1.6.0.1

[1] running a simulation with a really great number of nodes can be very memory and time expensive. Java Environments provides a standard size for the heap memory, this value can be overridden by using the command line specifiers –xms and –xmm. To run the experiments with more than 5 000 nodes in our machines, we had to reserve up to 200 MB of our memory to the Java virtual machine

3.8 Compilation and Execution

Package `example.mspastry` does not have particular dependencies, so that it can be compiled just with a single instruction.

For the next commands we will suppose that the current directory is the root of the project.

This command allow to compile the classes by using the `javac` command:

```
javac -cp <CLASSPATH> -d .\classes .\src\example\mspastry\*.java
```

supposing that peersim jar library and related jars were put in ./lib folder, we have that `<CLASSPATH>` takes value

```
.\lib\djep-1.0.0.jar;.\lib\jep-2.3.0.jar;.\lib\peersim-1.0.1.jar
```

3.8.1 Distribution

To distribute mspastry library in a single `JAR` file, just execute:

```
jar cvf mspastry.jar -C classes/example/ .
```

Remember that MSPastry also requires the following libraries: Peersim[2] and Java Expression Parser[3].

3.8.2 Running

Once the Peersim configuration file is edited and all classes are compiled, the following command will suffice to start the simulator:

```
java –cp <CLASSPATH> peersim.Simulator ./MSPastry.config
```

where `<CLASSPATH>`, similarly to the one described in the section *"Compilation and Execution"*, should also include the location of Peersim classes (including related libraries), MSPastry classes and application level classes specified in the configuration file.
In the previous command line, "`./MSPastry.config`" indicates the location of the configuration file.

Note that for very time/space costly simulations, the `java` command may also require the overriding of the memory reserved to tha *jvm*, by using the options `-Xms` and `–Xmx`.

[2] http://sourceforge.net/projects/PeerSim
[3] http://www.singularsys.com/jep/

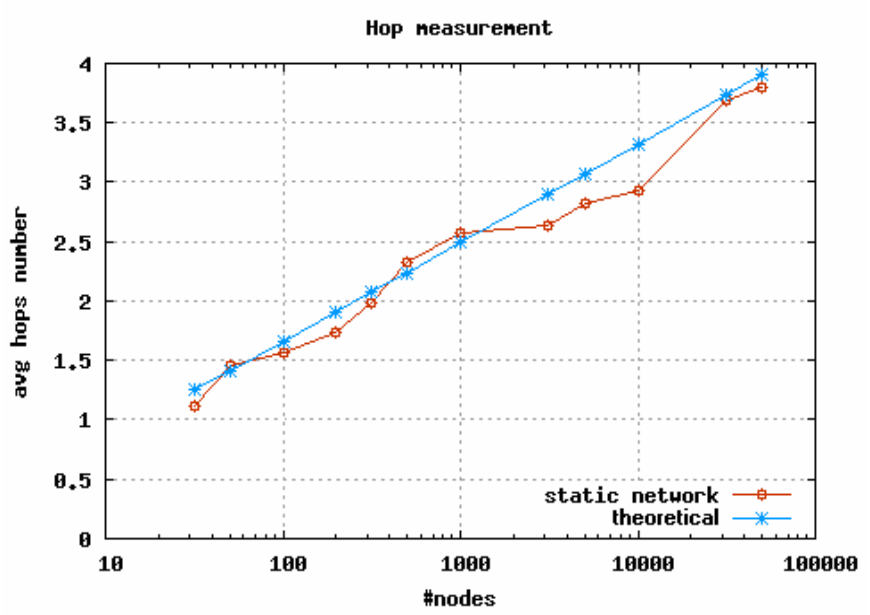
4 Conclusions

4.1 Comparison between theoretical expectations and obtained results

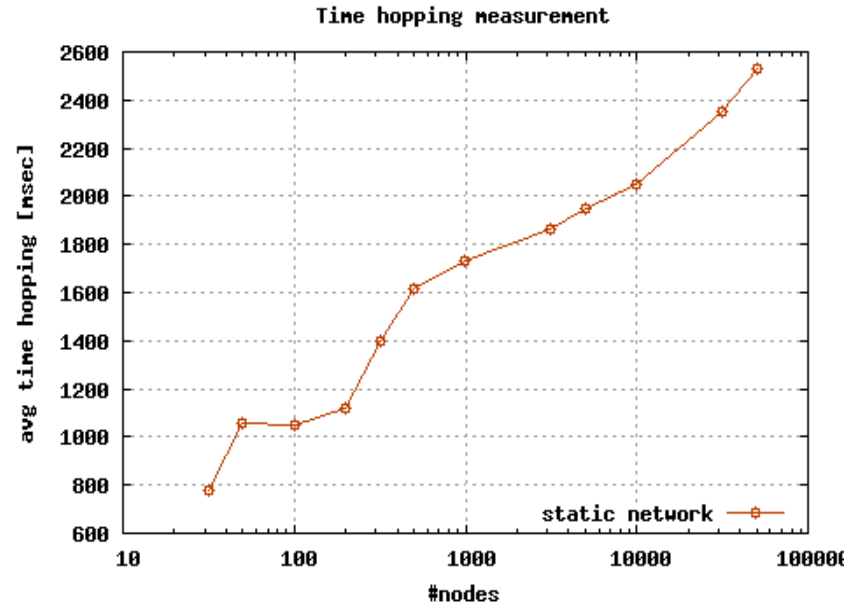
Let us now consider a comparison between the theoretical outcomes and what we obtained from the running of our simulation system.

The large scale adaptability of MSPastry was proven to have the properties that every message, starting from source to its destination, makes in average $\log_{2^b}(N)$ hops, with b=4; this means that the average number of hops in the network is logarithmic with the size (N) of the network itself.
We want to check this logarithmic function in our implementation.
We generated a control that randomly sends a great number of messages over the MSPastry overlay, and observing the number of hops every message takes to reach the destination. Additionally we observed the number of time instants every message spent to arrive.

The checking of the results was made by varying the size of the network from N = 32 to N = 50000. The following graph shows our results:



#nodes	Practical	theoretical
32	1,12	1,25
50	1,45	1,41
100	1,56	1,66
200	1,73	1,91
316	1,98	2,07
500	2,32	2,24
1000	2,56	2,49
3122	2,64	2,90
5000	2,81	3,07
10000	2,93	3,32
31622	3,69	3,73
50000	3,79	3,90



#nodes	Time
32	776
50	1059
100	1048
200	1119
316	1402
500	1615
1000	1732
3122	1862
5000	1948
10000	2053
31622	2354
50000	2534

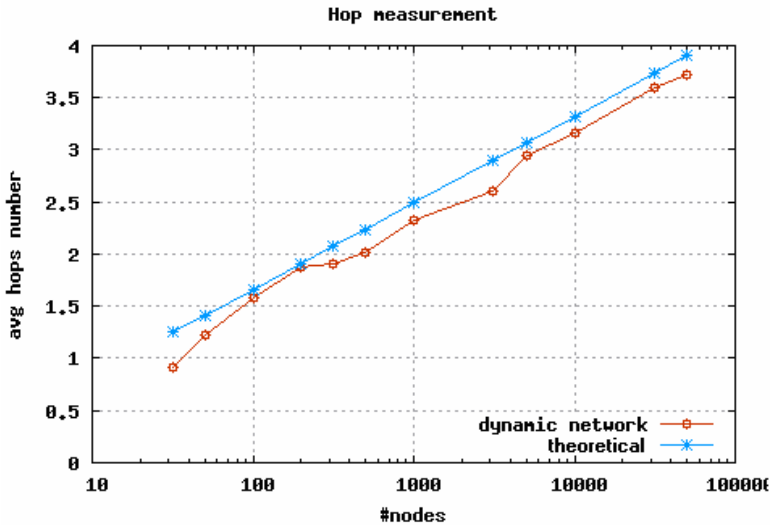
From the graphs we can observe that the performance obtained from our implementation of MSPastry are comparable the theoretical expectations.

In order to obtain some traffic in the network, we used a test Peersim Control, called `TrafficGenerator`, which works randomly by sending application level (lookup) messages.

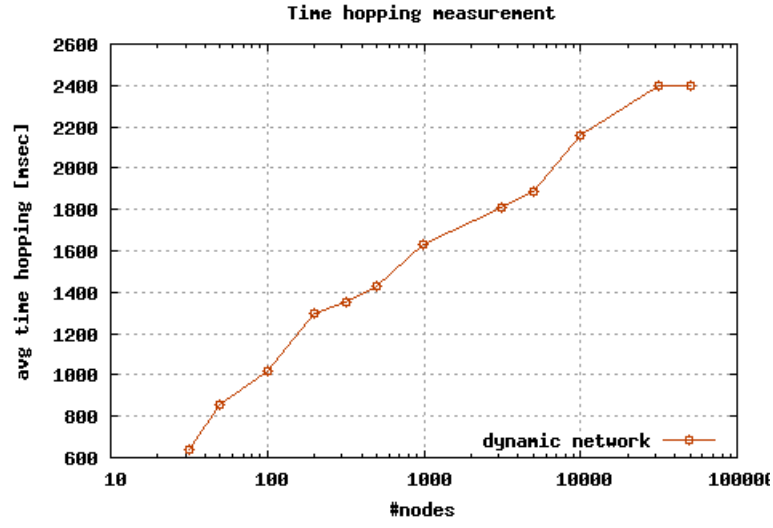
Data gathered during the execution of the experiments (represented in the graphs and in the tables) were collected by an Observer (`MSPastryObserver` class). Aim of `MSPastryObserver` was to accumulate statistical information of the average time spent by the messages to reach destination, and average number of hops made during the routing from source to the recipient.

The previous analysis was made is a *static* network, i.e. a network that does not suffer of joining of new nodes and neither failures of nodes. Now we want to observe how our system behave when the networks suffer of random oscillations. For this purpose a Peersim Control provide the (random) churning of the nodes in the network: `example.mspastry.Turbulence`.

The tests were run with the same set of network sizes as above. As we can notice from the output values reported in the table, for each test the size of the network was modified by randomly adding or removing nodes, respect with the initial network.



#nodes (at start)	#nodes (after)	Practical	theoretical
32	35	0,91	1,28
50	49	1,23	1,40
100	103	1,58	1,67
200	195	1,88	1,90
316	315	1,91	2,07
500	505	2,01	2,24
1000	989	2,33	2,48
3122	3109	2,61	2,90
5000	5131	2,95	3,08
10000	10087	3,17	3,32
31622	31603	3,59	3,73
50000	49913	3,72	3,90



#nodes	Time
32	641
50	859
100	1020
200	1296
316	1352
500	1432
1000	1629
3122	1811
5000	1890
10000	2162
31622	2401
50000	2398

By observing the results, even operating some churning of the number of nodes during the execution of the experiments, MSPastry behaves very well and is stable. In a “oscillating” network environment, pastry is able to show good performance, similarly in the case of a “static” network.

In order to obtain Dynamicity (adding and removing of nodes), we used a test Peersim Control, called `Turbulence`, which works randomly by adding new nodes, or removing existing ones.

4.2 Future Works

Our implementation was focused on the functionalities of the protocol, with no stress factor about performance, especially in memory space. As this protocol was conceived for very large networks, it could be useful to pay more attention to the design of the data structures.
Although the protocol requires only few kilobytes to store information, putting all nodes in the same machine results in a very time and space consuming, especially in the start-up phase, when the nodes must be initialized together in order to setting up the network.

5 References

[1] A. Rowstron and P. Druschel (Nov 2001). "*Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems*".
IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany: 329-350 *(http://research.microsoft.com/~antr/PAST/pastry.pdf)*

[2] Miguel Castro, Manuel Costa and Antony Rowstron (Dec 2003), "*Performance and dependability of structured peer-to-peer overlays*" (*http://research.microsoft.com/~antr/MS/performance-tr.pdf*)

[3] Márk Jelasity, "*A Basic Event Driven Example for PeerSim 1.0*" (*http://peersim.sourceforge.net/tutorialed/*)

[4] Nikos Drakos, Ross Moore (1997) "*PeerSim HOWTO: Build a new protocol for the PeerSim 1.0 simulator*" (*http://peersim.sourceforge.net/tutorial1/tutorial1.html*)