Non-centralized environment to monitor and dynamically configure highly distributed systems

Roger Hernandez

directed by

Yolanda Becerra Fontal
David Carrera Pérez

Department of Computer Architecture (DAC)

Informatics Engineering - Final Project
Barcelona, September 2012

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) BarcelonaTech

Contents

Chapter 1

Introduction

1.1 Document structure

This document is structured by chapters. The first one is a description of the project, with its motivations, goals, and expectations.

Since distributed computing is a rather specialized field, the second chapter is an introduction to these kind of systems and their major concerns, which should help cover the essential doubts or unknown terms in case of unfamiliarity.

The third chapter is the core of the application, and focuses on explaining the chosen architecture, the alternatives, and why it was decided to go on with it.

Chapter four focuses on third party software used as support, any specific implementation or in-depth explanation of those frameworks can be found here. Although most of this chapter aims to give information on just the platforms, some overall references may be given towards the specific project, such as general configuration choices, caching policies, ... all in all, nothing that affects the architecture model.

Having read chapter three and four, the fifth is the one that puts it all together, explaining the architecture implementation by using specific concepts that were covered in the previous chapter.

At the end of the implementation comes chapter six, with a bit of testing and evaluation, demonstrations of the application running, and some analysis from the tested configurations.

Finally come chapter seven and eight, with the project management information, conclusions of the project and future work.

1.2 Motivation

Distributed computing, cloud computing... any computing architecture that moves away from the single node configurations or typical client-server paradigms, is a trend that grows and progresses inevitably. There is an obvious need to start forgetting the techniques that have been used until now for the future environments that are not as *future* anymore.

The need of this growth requirement was so sudden, that systems have been appearing in a way where they may be enough to provide the services but most likely not under the best performance environments.

Most attention so far has been directed to the constraints of the applications regarding their clients' requirements, in a way where those were carefully monitored to achieve the target times of execution, uptime, quality of service... but recently other worries are starting to emerge.

Power consumption has mostly been a secondary concern until now, when it is starting to shift to a priority. Turning on hardware to meet quality of service requirements can be achieved with close to zero complexity, but this new concern for reducing the number of resources used to the minimum makes of this a problem that resembles the traveling salesman one.

This presents in a number of other problems, one of them being the need for a monitoring system to see the status of every node and their applications. Curiously enough, monitoring such a large amount of nodes and information is not as easy as logging each one and merging it in a common place, and this is where this project takes part.

There is no known "best way" to exchange and process the monitored

data of such an enormous systems, and having an application that allowed to test and compare different configurations would certainly help on deciding which schemas could be worth using.

Having an application that allowed to test for different configurations that could be changed relatively easy, without needing to reimplement the whole system, and then evaluating their performance, would help all those trying to organize and coordinate their distributed systems.

Another thing that motivates this project, is the peer-to-peer approach proposed for this work. Peer-to-peer networks have been somewhat popular within distributed systems, and there was a certain interest on knowing how they performed and if positive results could be achieved.

1.3 Statement

Given the motivation of this document, a thorough study into the workings and possibilities of peer-to-peer networks will be required. Once a peer-to-peer approach has been chosen and consolidated, development of the project per se will be ready to have the focus of development.

Once a peer-to-peer basis is established, by having the tools the paradigm offers and taking the requirements into consideration, it will be possible to start building the system that fits the motivation's most relevant aspects: having a platform that allows to evaluate multiple configurations of a monitoring application in a distributed environment.

Those configurations should be as manageable as possible, so that no major further developing is required in a future. The most interesting metrics to be monitored for this setup come from the network end, however multiple options should be given when deploying the platform, so that other metrics can be easily studied, again without the need of deep modifications.

Evaluating, testing and abstracting conclusions from this project will provide a platform that fulfills all the main wishes of the motivation that could be read previously.

1.4 Goals

The main goal of the project is to provide a peer-to-peer based platform that can be configurable in a way where a structure and a metric to monitor can be configured.

The platform must allow to change the configuration of the structure with easiness, and a valid flux of information. With this, but not in the scope of the project, analysis on different configurations could be done to see which of them performs best in a given environment.

The core project goals are mostly the ones listed in the architecture section below, the rest of the list presents either personal or collateral goals, or requirements with special interests, which inevitably form part of the project roadmap to reach the final objective.

- Overall consideration of the project
  - Understand the problem
  - Consider if there is related work or work in progress
  - Evaluate possible approaches
  - Evaluate the use of peer-to-peer
- Peer-to-Peer
  - Deep understanding of the paradigm
  - Choose a platform/framework
  - Feel comfortable and with control over the chosen solution

  - Obtain an implementation of the sought architecture with the peer-to-peer tools

- Architecture
  - Define an architecture given the approach of the project
  - Implement the architecture
  - Satisfy the restrictions of configuration (structure and metrics)
- Documentation of the project
  - Brief description and basics of third party libraries/frameworks
  - Full explanation of the implementation without going into minor details
  - Make note of every relevant consideration taken whether it stayed in the final result or not
  - Present a few examples with their corresponding analysis

Chapter 2

A glympse on distributed systems

2.1 Description

Distributed systems have always been challenging to deal with not only because of the complexity of the application that has to be developed, but because of the underlying protocols and supporting implementations that get everything running together. Most of those form a common set of problems that are regularly found in any application with distributed characteristics.

This may not seem critical at first, since if the problems are common it's usual to think there are common solutions, unfortunately this is not always the case. Most cases in distributed systems are different given the implementation, and will require a previous evaluation and specific implementation.

There are however, some guidelines, tips and generic algorithms to get past those issues, and as it will be seen later on, there is even some interfaces such as ZooKeeper that provide a few basic tools to ease the implementation process (it's not called Zoo by sheer coincidence).

In this chapter those properties are explained to give the reader an idea of what is being dealt with, and why some decisions bring to such characteristic implementations.

The listed properties are the ones that will be most relevant for this project and here they will be just presented, later on this document detailed information is available to see how each of them was dealt with.

2.2 Common properties and pitfalls in distributed systems

2.2.1 Messaging

The final goal of the application is to build a flux of information, so a messaging solution will be needed for this. Data will also need to be transferred among the nodes of this application for their own internal functioning.

The chosen method will require point to point data transferring, and closed group propagation.

The system must be eventually reliable (messages must be delivered), and considerably efficient (can't just broadcast a message to every node in the system and let the receivers decide if it is for them). For this reason, a full fledged reliable multicast protocol is not required, because missed messages are acceptable as long as it is a rare case that can be covered in future messages.

2.2.2 Scalability

This is the main motivation of the project and one of the biggest sought characteristics in distributed environments of considerable size and growth.

The architecture should allow a considerable growth in nodes and/or information transfers without punishing the performance of the system, neither globally nor partially. A configuration that escalates ideally would perform exactly the same be it with one single node or any higher number.

2.2.3 Single point of failure

A single point of failure, as the name suggests, is a certain part of a distributed environment that will disable the whole system in case of failure.

This is obviously a situation to avoid, no one wants to have hundreds of nodes hanging from a single machine, especially in environments where faults are most common. The system that is being developed seeks to have no single point of failure, in fact the ideal setting would be one where every node is independent from the others.

2.2.4 Fault tolerance and recovery

Distributed systems are usually formed by many physical nodes, this is something that makes it hard for administrators to manage the clusters in case of failure, so avoiding those situations or adding automatic methods of recovery towards a more autonomic configuration is a trend that is worth exploiting.

It is very difficult to have every possible situation of failure into consideration when implementing a distributed system, mostly because of synchronization issues and all the different threads trying to work together. But many faults can be anticipated, and those should be dealt with, giving the application secondary instructions in case something doesn't go as expected.

It is not enough to provide a node with fixes just for himself, those should also be able to notice if other relevant nodes in the system failed and they have to take action to make up for it, as one whole system.

2.2.5 Discovery

Discovering resources or other nodes has been a requirement at least since the beginning of the networks. A distributed system, which inevitably runs on a network environment, will require of discovery methods.

On top of that, it is not enough that they are just on the physical layer, but also on the application one. This is because the platform that is being developed will be using logical nodes that organize in different structures, and knowing the structure and being able to contact other nodes is a must.

Some services will be required for the nodes to offer or use, and those must be available to the other nodes, or a subset of those. Dynamically discovering services or finding them, has to be taken into account.

Related to discovery issues, a point of entry is necessary for the nodes to decide where to start looking for information, or how to connect to the other nodes. In a configuration where there is no assigned roles from the beginning, choosing a point of entry that is not prone to fail, or that can be dynamic is something that also has to be thought deeply.

2.2.6 Group Membership

One of the most common methods of organizing a bunch of nodes is dividing them in groups. This simple thinking presents many problems, because the node has to be conscious of the group it is in, and every member of the group must be conscious of the other members in it. A group by itself is an abstract thing, it is common in such environments that a group doesn't really exist unless there is at least one node in it with conscience of the group.

Relevant facts

- A node that attempts to join a group that doesn't exist, will have to create it
- Creating a group is not straightforward, a node must be absolutely sure the group does not exist
- A node in a group must see every other node in the group, and only every node in the group
- When a node joins a group, every member of this group has to be conscious of the new member. In similarity, when a node leaves the group, or crashes, consciousness from the others is also required
- One of the most common reasons for having groups is sharing group specific information or services within this set, it is fundamental that spreading information is possible and reliable
- In case the group becomes empty, future nodes interested on joining it might be required to retrieve old information (in case the group doesn't delete all its traces)

2.2.7 Leader Election

This is the process that takes place in a system where from a set of equal nodes one or a subgroup of nodes, will be acting as leader of the group, offering a service or executing a task with guarantee that no others are doing so.

Relevant facts

- There has to be a chosen node eventually
- Only a defined number of nodes (usually one) can be leader at any time. This means, two nodes can't decide to become leader at the same time
- Decisions are not taken individually
- Once a node dies, it can't report his state to the others
- When selecting a new leader, it has to be guaranteed that the old leader is no longer active
- Once a leader is chosen, all the nodes must be aware of the decision

2.2.8 Flexibility

Even though this is not a specific characteristic of distributed systems it is important to have in mind. Building an application this complex would be meaningless if it was aimed to one unique possible configuration, or if it was only able to monitor a single resource.

The application should be built in a way where one can alter the configuration of the architecture without need of understanding the whole structure, ideally changes should be made only to the code sections where architecture is defined.

A similar policy should be used to change the data that is being monitored, making it easy to switch, for example, between network utility, disk, memory...

There is always a limit to how much those configurable parts can be abstracted from the core code, however it should be as much independent as possible, or at least as controlled as possible, so that if changes are required, another developer can figure out what part of the code needs to be modified.

Chapter 3

System architecture

3.1 Introduction

This chapter explains the process of deciding how the final architecture of the system was reached. Details on how it was implemented or how specific functionalities were achieved can be found in further chapters.

It should be mentioned that many architectural decisions that appear in this section were not a straight consequence of a previous analysis, but many situations had to be designed in a way that could adapt to the tools that were being used, as can be seen in the latter chapter of technical background 4.

This field was open for any solution, however the special interest in Peer-to-Peer networks inevitably gave shape in the way of tackling the problem. Peer-to-Peer architectures are centered in systems based in groups and services, therefore the architecture design will be focused around those two concepts.

The use of a P2P system provides the following:

- Group membership
- Direct and reliable communication (peer-to-peer)
- General unreliable message propagation
- Services
- Persistent network consistency in case of failure
- Inter-network communication

The physical configuration of the hardware systems is also a very influential factor in the shaping of the system. Datacenters usually consist of multiple clusters or containers, which host the physical nodes. This setup gives an air of hierarchical group architectures already, being the groups the containers and datacenters. However that doesn't resolve the problem, those groups are not controlled at all, and there is no standard whatsoever in the distribution of the number of nodes per container, or the connections among them.

3.2 First approach: Chain model

The first approach to the problem was a chain of connected nodes, much like a discontinued ring. Even though this served as an extra training with the third party tools such as JXTA, it was obvious that the setup was not a good choice. The whole point of this decision was to have a very simple configuration that resembled a known protocol and that would suppose a working base with evolution capabilities as obstacles appeared.

Every node has to be connected to another one, the *next* node, as opposed to the *previous* which connects to this one. The first node of the chain is an especially chosen one that offers a service to the network, allowing single nodes to seek a point of connection to join the chain.

The flux of information is pretty straightforward, every node just reports his information (and if required, the gathered one from the nodes it has behind in the chain), and passes it to his *next* partner.

The most obvious disadvantage of this build is the poor scalability it

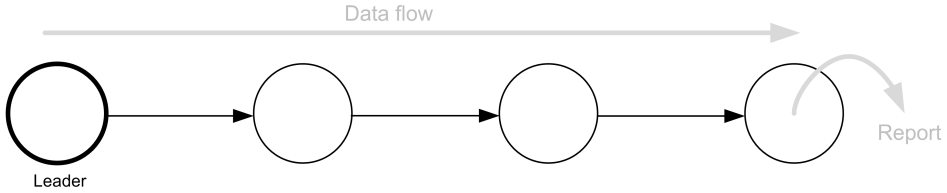

Figure 3.1: Chain mode example with four nodes

presents, with a single chain the time it would take for messages to propagate through the whole queue would be too long. There is also the detail of deciding who becomes the head of the chain, which is a strong requirement since one and only one node must offer this service.

Node faults can easily be detected by noticing a disconnection in the chain links (JXTA pipe disconnection). In case of failure though, one of the drawbacks of this configuration presents: the broken side of the chain can't possibly know which node is the last in the healthy chain.

At this point it was decided to allow multiple chains, that is multiple nodes offering the service that allowed others to discover them and join their queue. With this two major inconveniences were overcome: the system was more scalable since it allowed to limit the chain lengths, and nodes didn't need to find their previous chain anymore in case of failure, since the new chain's beginning could just start offering the service as a new independent piece of the system. The head node then, would have a new task to manage, ensuring the chain length wasn't surpassed, which could be easily achieved by issuing a message that walked the chain with an increasing counter on every step until a limit was reached, ordering an immediate cut.

However, those solutions proved to spawn more issues. The fact that chains could cut as they pleased (or failed) was a quite simple fix that killed

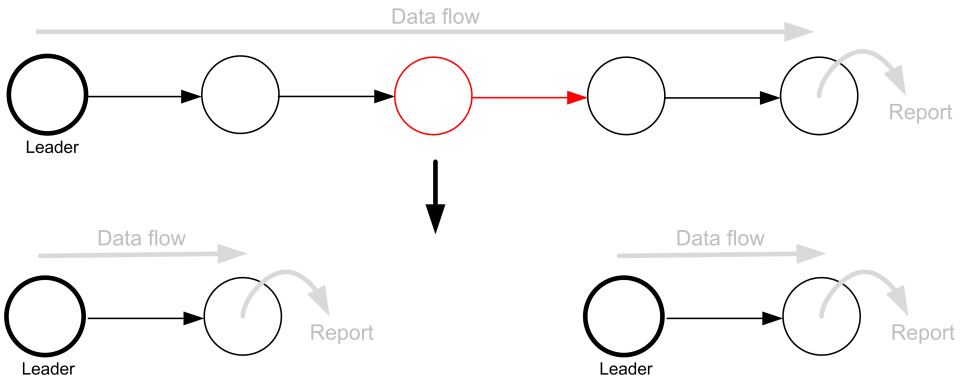

Figure 3.2: Chain node failure and recovery accepting more than one chain

two birds with one stone, but having too many cuts could lead to an excessive granularity, being the worst case scenario the situation where every piece in the system was a head node with no other followers in the chain, a chain merging system was required.

On top of that, with the existence of multiple chains the monitoring report was split into as many pieces as chains, so again a merging method was required for this purpose.

The original problem of selecting a starting head node could be easily dodged by manually setting a stable system (doing the choice on bootstrapping), however with having multiple chains this translates to the need of having some sort of consensus system to decide when new chains are starting. This isn't a strong requirement, since nodes could join an existing chain whatever its size it had, then get split immediately and start their own chain. However this could result in the need of merging the newly created heads, resulting in a repeating break and build situation. This situation would eventually even itself out in a relatively stable system, but this much reconfiguration is very prone to failure in distributed systems, so having some kind

of coordination mechanism is a great recommendation.

With those new difficulties and foreseeing that, besides introducing more parallel issues, new patches would require centralized solutions and hierarchical structures, it was decided to take a turn on the basic idea.

3.3 Selected architecture: Group model

Besides the latest setbacks of the chain model, having multiple queues started to lose the purpose of having every node in control of a single structure. At this point, even though it was attempted to keep the idea of having chains, it started to became evident this was starting to shape into an architecture with multiple groups where nodes weren't hanging from so few points of support.

Instead of having a customized structure, this time the peer-to-peer capabilities will be exploited, having a group manage sets of nodes that report to a single one. The leader of the group, which isn't in charge of managing the structure anymore, will act as information link among the other groups by collecting the data the members of its group offer.

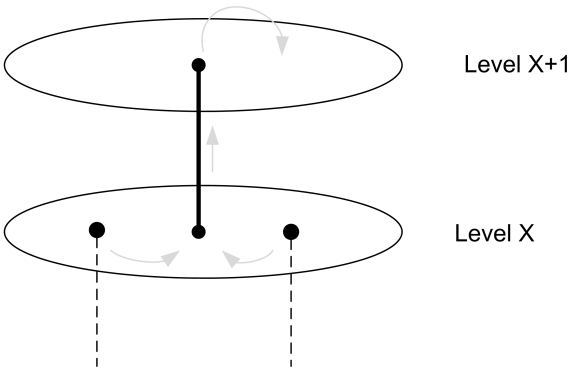

Figure 3.3: Basic configuration with two groups and data flow between them

The issue of having multiple points of data collecting still stands, however this will be fixed by creating a hierarchical tree structure for the data flow, with information going from bottom to top, ensuring a convergence towards a single point, the node that will issue the final report of the whole system.

With this, the leader election required for choosing chain heads disappears, but a similar situation emerges where, since all the nodes in a group are equal an in no order at all, there has to be a choice of who becomes the group manager or leader.

Clearing it up, when nodes are started they will attempt to join the group they have been given (most like a reference to their physical group status such as a container). If the group is full the node will instance a new group at the same level of the full one to extend its capacity under the size configuration restraints. A leader for the group is decided, and every other node reports information to the group, which will be collected by this chosen one.

The connection link between the gathering group and the receiver group (from a lower level to its higher one) doesn't involve communication between two nodes, instead the link is in a group-leader fashion. To avoid redundancy and massive changes in the tree in case of failure, each node will have presence in a maximum (ignoring the bottom, or leaf, nodes) of two groups, those being the lower group in which the node acts as an agent that gathers information, and the upper group where the node acts as a regular slave that reports information, which would be the one it collected in the lower group.

The collected information has to be converged to a single point of collection, to achieve this the leaders of the group will keep pushing the reports up in the structure. Each group belongs to a level of the tree, and each level

can have as many groups as required by the chosen group size constraint.

Therefore the tree will be built bottom-up, adding new levels to the tree from the top as they are needed. The growth will happen with a promotion system, where leaders (with the unique exception of the tree root node) and new nodes (starting at the bottom) always attempt to join their higher level. This will make the structure behave just as if nodes were bubbles in a pot of water, where as soon as they have space in a higher position, they will be promoted.

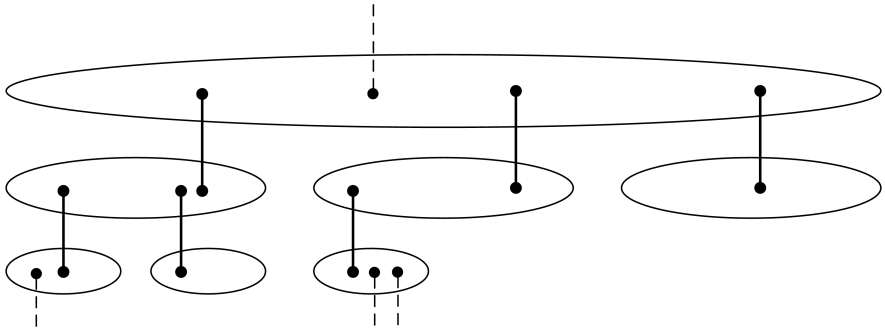

Figure 3.4: A configuration with multiple groups and levels seeing the links between them

In case of critical failure (that would be one of the leader nodes), the structure doesn't break at all and a mere replacement for the reporting node has to be found. This replacement will be chosen from the group of nodes lacking the leader with a consensus algorithm.

The newly promoted node will join a group of the higher level of the tree, which doesn't necessarily have to be the same as the pervious one. This doesn't affect the architecture at all, since what is of most importance is to satisfy the rules such as size and levels, which following this system will always be in check.

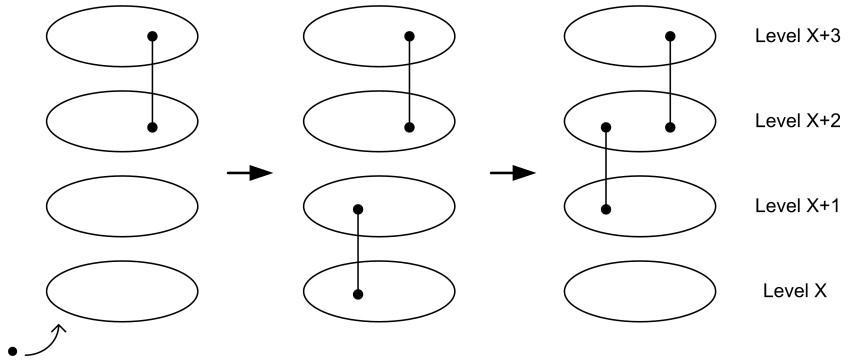

Figure 3.5: A new node joining the system and promoting upwards until groups with leaders are found to establish a completely linked structure (the promotion method is the same as when nodes fail)

Depending on the configurable number of nodes per group, the tree can host a ridiculous amount of nodes with very few levels, or grow high with a very small number of nodes per group. Any configuration will be valid and this flexibility was exactly what the project was aiming for, to provide a changeable architecture with options to monitor which is the best scenario depending on the datacenter it is being deployed on; in some cases a higher setup might be better, while in others a more wide configuration with less leaders could work just fine.

A late addition to the architecture was a tracking system to analyze how *broken* the tree was (in terms of balance and splitness). The nodes propagate the structure information altogether with the monitoring data up to the tree, and the root node is in charge of deciding if some changes should be done in any part of the tree (this has to be done from the top level because a global vision is required). If something has to be modified the root node issues a message with orders that is propagated from top to bottom for the nodes to read and take action. This restructuring protocol hasn't been thoroughly

tested and it is not a requirement of the project, in fact even if the tree is unbalanced, or even if it could be more compact, these facts don't affect the performance or architecture model at all.

Chapter 4

Technical background

4.1 Introduction

This chapter focuses on the main third party tools that have not been built in this project's scenario but that have been used in it. The decision of using external solutions comes from the motivation to go straight for the main problem without needing secondary issues.

The explanation of the platforms goes deep enough so that it can serve as a side documentation for future work on this project, or on similar ones. Everything explained in this chapter is exclusive to these technologies, and not heavily linked to what was developed in the project core, however all of the information given here is related in some way to the parts that were used on the main piece of work.

Those have been considered and added into the project as they were required and analyzed, and by the end two of such services have been used in the development: JXTA and Apache ZooKeeper.

JXTA was chosen due to the lack of alternatives to work in peer-to-peer environments, conditioned with the fact that this framework had already been used in a previous and similar smaller project without much success, however it was thought that with a bit more research it would prove a great tool to build a custom P2P network, the system offered all that was required for the project goals and much more.

In the other hand, ZooKeeper was more of an spontaneous decision to tackle a very specific problem. In this case alternatives were not considered, putting trust on Apache's reputation, and knowing this service was fully capable of helping on this issue, previous recommendations and references

were known for this system as well.

4.2 JXTA

4.2.1 Overview

JXTA is an open source peer-to-peer (P2P) protocol specification begun by Sun Microsystems based on Java technology.

Efficient communication between the nodes is a sure requirement in the middleware of this project, and JXTA provides a specification that allows to discover and establish connections, make service implementations and distribute information in a reasonable easy way.

On top of that, there will be a gain on benefits from the epidemic characteristics of the P2P protocols regarding data management and decentralization, which is a fact that directly reduces the impact of possible node failures in the system.

In this chapter a general idea of how JXTA works will be given. JXTA is vaster than the information that can be found in this document, and only the concepts and elements that are most essential to this project will be detailed, and if relevant enough, with further examples and comprehension.

4.2.2 Cache files and configuration

Since peer to peer networks don't tend to change much, in order not to rediscover everything a peer has found, each JXTA instance has its own cache in form of a file directory in the system, allowing to recover every discovered

advertisement in event of shutdown or reallocation. It is important not to mistake the *cache* nomenclature JXTA is using here, for this is not a cache to improve the system's performance or responsiveness, but the whole data JXTA uses to work (every structure, advertisement, etc.).

Two things have to be taken into consideration when implementing applications in JXTA:

- It is a good idea to manually set the location of the cache directory, not only this will offer greater control over the application, but also avoid conflict in a situation where multiple JXTA instances are started from the same machine (locations should obviously differ).
- Disk space has to be considered depending on the network structure, size, and node's role in it (rendezvous nodes tend to use a much larger cache size).

```
1 String localPath = "." + System.getProperty("file.separator
     ") + "cacheFolder");
2 File configFile = new File(localPath);
3 NetworkManager netManager = new NetworkManager(
     NetworkManager.ConfigMode.EDGE,
4 "PeerName", configFile.toURI());
```

Due to network discovery building up slowly (the peer sees the extension of the mesh progressively), it can take a long time for a peer to see enough of the network to have a complete enough vision, that is another advantage of the JXTA cache. Restarting the node after it was taken down, will allow the peer to be in the same status as when it crashed in terms of network

visibility. This of course comes with obsolescence issues, but that is usually not troublesome considering the nature of peer to peer networks, where obsolete information is a regular that has to be checked and purged when further testing or direct connections are attempted.

For convenience and to make sure no traces of other networks are interfering with the system, in the scope of this project the JXTA cache will always be wiped when starting the node, this way a clean start will be guaranteed.

JXTA multicasts by default, and this feature will be turned off. Even though in production conditions it is better to enable it, a multicast friendly environment can't always be guaranteed, so for this project the worse case scenario of a network will be assumed. With multicast off every message can still reach its destination as long as there is a stable and correct network configuration (seeds, rendezvous, relays...).

4.2.3 Basic concepts

JXTA is a complicated set of pieces that can be difficult to understand, but all of those move around the basic concept of the Peer. In this section Peers and PeerGroups are explained in detail so that when reading the following ones an easier understandment should be apparent.

Peer

A peer in JXTA is usually the representation of any networked device (phone, server, PDA, etc.) in a PeerGroup. Multiple peers can be started from the same JXTA instance, however the most common is to have a single peer for

each client.

It is important to understand that a Peer is a representation of an individual in a PeerGroup, which means, one individual can have as many representations as PeerGroups it wishes to join and each of those doesn't necessarily has the same characteristics, which means one individual can have multiple roles in the different groups it is part of.

The following code shows how the base Peer of a JXTA instance is started, this Peer is joining the general NetPeerGroup by default when starting it after setting the desired configuration parameters.

```
1 netManager = new NetworkManager(NetworkManager.ConfigMode.
    EDGE,name);
2 netConfigurator = netManager.getConfigurator(); // With the
    Configurator
3 instance the peer can be customized before starting it
4 netPeerGroup = netManager.startNetwork();
```

PeerGroup

A PeerGroup is a network resource which peers can discover and join in order to organize themselves, or share common interests such as services. The main characteristic of PeerGroups is that they are a representation of a set of services (in the next section the distinction between Peer and PeerGroup services is explained), therefore if a Peer outside of the group is interested in a service offered by certain group, this peer can access the service from any peer in this group.

There are two fundamental PeerGroups in any network that have unique

IDs known by all applications implementing JXTA

- WorldPeerGroup: It's the first group that initiates during the bootstrapping process and it isn't meant to perform P2P networking. Instead it serves as a configuration template of basic services, that allows instances of NetPeerGroups to be initiated; this group serves as the root of the system.

- NetPeerGroup: Child, and usually a copy, of the WorldPeerGroup. It is the starting point of the network peers and the lowest layer where custom networking should start. This is also the peergroup where any specific network configuration can be set for the first time.

In this system, PeerGroups will be used just for organization purposes since there's no service that can benefit from a whole PeerGroup perspective.

PeerGroup management has two approaches; the first is a very formal method with strict joining policies and credential requirements, implemented by using the Membership Service. Peers interested in joining the group must discover it, apply to the group, and finally join with the credentials provided in the application process. Finally, if a member wishes to, it can resign the group membership.

```
1 StructuredDocument creds = null;
2 // Generate the credentials for the Peer Group
3 AuthenticationCredential authCred =new
     AuthenticationCredential( grp, null, creds );
4
5 // Get the MembershipService from the peer group
6 MembershipService membership = grp.getMembershipService();
```

```
7
8  // Get the Authenticator from the Authentication creds
9  Authenticator auth = membership.apply( authCred );
10
11 // Check if everything is okay to join the group
12 if (auth.isReadyForJoin()){
13     Credential myCred = membership.join(auth);
14     System.out.println("Successfully␣joined␣group␣" + grp.
           getPeerGroupName());
15 }
```

However, the membership protocol is bound to careful configuration and adds an extra layer of complexity to the application, which doesn't require such member control functionalities. For those reasons, the basic peergroup membership option has been chosen.

To achieve this simple membership protocol the PeerGroup's newGroup() method will be extensively used, for this method creates a new instance of the desired peergroup given the peergroup advertisement, or either the details of the group.

```
1  PeerGroup newGroup = netPeerGroup.newGroup(null, implAdv,
       groupName, groupDesc);
```

To join an already existing group the same method can be used, ignoring the fact of the method name being misleading. To ensure the same group is joined (giving details wouldn't work, since multiple groups can have the same name but different PeerGroupID's), the peergroup advertisement will be rebuilt by using the existing PeerGroupID, and using it altogether with the newGroup() method to get the instance of the group.

```
1 PeerGroup joinedGroup =
2 netPeerGroup.newGroup(PeerGroupID.create(URI.create(groupId
     )));
```

4.2.4 Protocols and services

Services are the basics of JXTA to get the protocols working. Altogether with Advertisements which could be seen as the bricks of the JXTA network, services act like the blueprints that use those bricks to make the network stick together. In practice, services are the knowledge each Peer owns, uses or offers to other peers to constitute the whole JXTA network.

The JXTA core is based on a set of services in charge of making the network run. Every Peer owns or implements those services innately, and those are: the Access Service, Discovery Service, Endpoint Service, Membership Service, Peer Info Service, Pipe Service, Rendezvous Service and Resolver Service.

Most of these service functions are self-explanatory by looking at their name, and have available interfaces for the programmer to check network parameters. For example, the Discovery Service is the one in charge of finding and scanning advertisements through the network, which will be used to publish and discover different advertisements as the application finds fit. Multiple options are in hands of the developer, such as choosing expiry time or propagation methods, filtering, etc.

Clients in a peer to peer network exploit their discovery capabilities to find the information they seek, and once they locate an available endpoint,

they establish a direct connection to it. In JXTA this direct connection can be made thanks to the Pipe Service, peers can create pipes and publish their advertisements to allow other peers to connect to this pipe. In fact, most services use pipes implicitly by having their pipe advertisements embedded into the service ones.

Services can either be group-wise or at peer level; Group services are available to any Peer that joins this group, and they will be able to retrieve the service and use it at their will. Due to the peer to peer nature, group services are very common given that P2P networks can grow very large in size and it would be impossible for every peer to know everything in the network, so those will only be acquired when needed

In a controlled network such as the one described in this project where, as will be seen, every peer is equal, group services will not be required, so any service implemented will be a Peer Service, and every peer will bootstrap with all the services of the mesh. Even though Peer services will be being used, the system will resemble one where every peer is sharing the same services, so if all the peers were to be seen as if they were in the same peergroup, it could be said that the services are peergroup-based.

To implement a service the class will have to inherit the JXTA Service implementation. The developer will be in charge to make sure the following structures are built for the service:

- **Module Class Advertisement:** Used for advertising the existence of a JXTA module, it links an unique module ID to the service.
- **Module Specification Advertisement:** Serves as a reference to the

module specification. It also allows to create remote communications, by transporting pipe advertisements, proxy modules or authentication modules.

- **Module Implementation Advertisement:** This advertisement contains the implementation of the service, its provider, location, etc. It identifies and gives information about different implementations of the same service. This is the document that allows peers to learn services from peergroups that were unknown to them beforehand.

- **Pipe Advertisement:** In case the service wants to use direct communication, the most common way to do so is with pipes. This advertisement can be embeeded into the Module Specification Advertisement of the service for a more standard and accessible discovery. If the service is expecting communications with a pipe, it will require to implement the Java *Runnable* interface, forcing the implementation of methods such as *init*, *startApp*, *stopApp* and *run*. This is to allow the service to listen to incoming pipe connections in a separate thread, so every pipe management code such as initialization and opening of the pipe should be done in those methods. Doing so will unavoidably require the implementation of a pipe listener class to process the incoming messages.

4.2.5 Connection management

Each peer can be started with (or mutate to) a specific network role among three possible ones

- Edge: Regular peers with no specific network infrastructure capabilities.

- Rendezvous: Edge peers that implement and offer the rendezvous service to their PeerGroup. Those peers are responsible of bridging JXTA resources that aren't directly connected (as would be the case, usually, in a LAN network). They keep a map of all connected peers in a group in order to help forwarding messages among it.

- Relay: Edge peers with the purpose of helping other peers, hidden by some network features (such as firewalls or TCP/IP limitations), to overcome those boundaries.

Both the rendezvours and relay roles are achievable by a single peer at the same time. There are two more unmentioned roles here which won't offer any advantage do this project, those being AdHoc and Proxy peers.

Rendezvous discovery, be it either among rendezvous peers or edge peers trying to locate rendezvous ones, becomes really slow if no hints are given, and in some cases even impossible (LAN networks with restrictions, no broadcasting, or Internet). Seeds, which are actually common IP addresses, will have to be given to the peers so that they can find other remote peers easily. Once a rendezvous has been reached, the peer will automatically (according to the configured time of information propagation) see everything the rendezvous is already gathering (as long as his group memberships allow him to do so), acting as a common link between peers.

```
1 // Give the peer a set of rendezvous seeds
```

```
2 netConfigurator.addSeedRendezvous(new URI(
      rendezvousSeedsList));
3
4 // This call blocks and returns true if a connection to a
      rendezvous was
5 successfully established
6 netManager.waitForRendezvousConnection(timeout);
```

There is no specific ratio of edge-rendezvous peers given for a network, so this configuration has to be carefully studied [1]. However, it is recommended to have two layers of rendezvous in the JXTA structure, one with the rendezvous that will communicate different edge peers, and another one to communicate rendezvous with rendezvous. This way, potential overload on the rendezvous will be less likely.

JXTA offers a method from the RendezVous Service that allows a Peer to consider its role in a PeerGroup given the performance and discovery behaviours of the network. Since a perfect peer to peer network topology is not within the purposes of this project, it has been decided to leave the RendezVous management up to JXTA by turning this feature for every Peer in the system, this way a guarantee of communication will be achieved even though it might not be the best choice in terms of efficiency. The line of code below sets this automatic behaviour on.

```
1 peerGroup.getRendezVousService().setAutoStart(true);
```

4.2.6 Communication and data structure

There are different means to communicate within the network, the most basic one being the advertisement propagation system, and though this is very convenient for discovery, scalability, mass publication, etc., it's not the best way to establish peer to peer communications. There are specific methods for doing so, which involve using the JXTA implemented pipes (unidirectional or bidirectional), or framework sockets. Establishing those communications is achieved by network discovery, but once a peer has discovered any of those, it can start a direct and private communication protocol.

Advertisements

JXTA protocols make use of XML formatted documents (messages) to provide information to the network. Those elements are called Advertisements, and besides identifying different resources of the network structure, they compose the core of the JXTA messaging system.

Most of those advertisements (especially the network control related ones) are automatically propagated by the default services offered by the network. Otherwise publish methods can be used to explicitly propagate them. Additionally, control over the advertisements expiry time is fully customizable by the developer.

There are different kinds of advertisements, some of them being the following.

- Peer Advertisement: Identifies a peer in the network.
- PeerGroup Advertisement: Identifies a peer group in the network.

- PipeAdvertisement: Contains information regarding a communication pipe.
- Service Advertisements: There are different kinds of service advertisements (Module, Implementation, Specification...) and they are used to implement the different services in the network, provide connection information for those, etc.
- Custom Advertisements: JXTA offers the developer a means to create his own advertisements with custom fields of information, data types, etc.

Implementation of a custom advertisement Own advertisements with custom fields can be created by using the Interfaces JXTA provides, and those can be used as a custom method for exchanging information within the network. Since the creation of a new custom advertisement is generic for any case, it is detailed as follows:

1. Define the desired fields by declaring the field containing attribute and the tag that identifies it.
2. Decide which fields are indexable, this will allow for the Discovery Services to search by these specific tags.
3. Implement constructors, getters and setters, clone.
4. Fill the processing function (custom attributes are managed with tags, this is the function that acts as a link between tag and content).

5. Complete the document generating method in charge of parsing the object into an XML format.

The new advertisement class has to be known to the network, therefore it has to be registered at some point before the JXTA bootstrapping, this is achieved with the following line of code:

```
1 AdvertisementFactory.registerAdvertisementInstance(
      MyCustomAdvertisement.getAdvertisementType(),
2 new MyCustomAdvertisement.Instantiator());
```

Advertisement publishing and discovery As stated before most advertisements required for the network basics to work are automatically propagated, but application specific advertisements (such as custom ones) are ignored by the network services, those must be published manually.

Each peer's cache of advertisements can be seen as the *local* area of information, while data from other peers compounds the *remote* information source. At least one remote operation is required for custom created advertisements to become widely accessible in the network.

When publishing an advertisement this is done in the bounds of a certain PeerGroup, as can be seen below, since to publish an advertisement a Discovery Service is required, and each PeerGroup has its own Discovery Service. Having this in mind, one has to be mindful when publishing and fetching advertisements, because looking at the wrong group will return no results, or the wrong ones.

```
1 // Local publish, requires remote discovery by other peers
2 peerGroup.getDiscoveryService().publish(advertisement);
```

```
3
4 // Remote publishing, attempts to publish on all configured
      transports
5 peerGroup.getDiscoveryService().remotePublish(advertisement
    );
```

Locally publishing an advertisement will result in the advertisement being stored in the local cache, and it will require peers interested in it to forcefully discover remote advertisements by using the getRemoteAdvertisements() method.

```
1 // Local publish, requires remote discovery by other peers
2 peerGroup.getDiscoveryService().publish(advertisement);
```

Publishing an advertisement remotely propagates it to the peers that somehow are in contact with the publishing one for the given expiration time. Remotely published advertisements are not stored in the local cache, and may create many undesired connections depending on how it is used.

```
1 // Remote publish
2 peerGroup.getDiscoveryService().remotePublish(modSpecAdv);
```

To see or retrieve advertisements published by other peers, those must be read from the local cache. If looking for specific or recently updated advertisements, a call to getRemoteAdvertisements() can be made to attempt to fetch advertisements in other peers' caches to the cache of the requesting peer, by doing so the network will start being remotely scanned. Considering the topology of the network and the characteristics of a peer to peer paradigm, this is a best efforts search that attempts to reach for the matching advertisements without any guarantee or prediction over the result, that's why,

if consistent information is expected, the topology must be completely controlled by the implementation, for example by assuring that all the wanted destinations are reachable by common RendezVous nodes.

Different filters can be applied to ease the search and performance by using regular expressions, advertisement types or the number of advertisements to scan for.

Making a request of 0 advertisements of *PEER* type advertisements is a special combination which will make every peer that receives the request to only send back their own Peer Advertisement.

```
1 // The parameters define filters as can be read from the
     reference.
2 peerGroup.getDiscoveryService().getRemoteAdvertisements(
     null, advType, null,
3 null, 100);
```

Obviously this call can't deliver an instant result, so certain time must be allowed from the time it is issued until the local retrieval of advertisements. To look for advertisements up in the local cache the getLocalAdvertisements() method can be used, with parameters similar to the ones used in the remote fetch, it will return a list with the matching query results.

```
1 // Getting advertisements locally
2 Enumeration<Advertisement> enu =
3 peerGroup.getDiscoveryService().getLocalAdvertisements(
     advType, null, null);
```

When publishing an advertisement there are two configurable parameters that can both determine how up to date one can expect a snapshot of the

network is, and have a heavy impact on the system's performance.

Those are **lifetime** which determines how long an advertisement will exist, and **expiration** which indicates the time an advertisement is allowed to be cached locally by other peers. Up until now the methods without those parameters were shown, where the default JXTA lifetime and expiration values are used, however those can be set manually when publishing an advertisement as seen below.

```
1 netPeerGroup.getDiscoveryService().publish(advertisement,
    lifetime, expiration);
```

Pipes

Pipes in JXTA are used to exchange messages between peers. Those can be exchanged between two peers in a point-to-point fashion, or from one to a set of peers, which would be a propagate pipe. Options to use secure unicast pipes are also available.

JXTA's PipeService will be the tool to use pipes. A pipe is represented with a PipeAdvertisement, which contains the required information to establish a connection, any peer interested in communicating with another peer by a pipe, will require the discovery of this advertisement beforehand.

To create a pipe, the first step is to create an advertisement as shown below.

```
1 PipeAdvertisement pipeAdv = (PipeAdvertisement)
    AdvertisementFactory.newAdvertisement(PipeAdvertisement.
    getAdvertisementType());
```

```
2 PipeID pipeID = IDFactory.newPipeID(PeerGroupID.
     defaultNetPeerGroupID, name);
3 pipeAdv.setPipeID(pipeID);
4 pipeAdv.setType(PipeService.UnicastType);
5 pipeAdv.setName(name);
6 pipeAdv.setDescription("Created␣by␣" + name);
```

Once the advertisement is done, the pipe may be created or instantiated (when using unidirectional pipes, when creating a pipe one must choose between an input or an output pipe.

```
1 PipeService pipeSvc = defaultGroup.getPipeService();
2 MyPipeListener aPipeListener = new MyPipeListener();
3 InputPipe inPipe = pipeSvc.createInputPipe(pipeAdv,
     aPipeListener);
4
5 //Publish the newly created pipe
6 defaultGroup.getDiscoveryService().publish(inPipe.
     getAdvertisement());
```

When calling the creator method a listener object has to be set besides giving a PipeAdvertisement, this will allow the gathering of messages. In the previous example the custom class MyPipeListener was used to receive messages in the listening method *PipeMsgEvent*, which should look something like the following code.

```
1 public class MyPipeListener implements PipeMsgListener {
2     public void pipeMsgEvent(PipeMsgEvent pme) {
3         // Received a message, process it
4     }
5 }
```

Pipe messages are XML documents as expected, though JXTA provides a friendlier interface to manage them, based on messages, message elements, and workspaces.

To send a message a similar procedure is followed:

```
1  Message msg = new Message();
2  OutputPipe outPipe =
3  peerGroup.getPipeService().createOutputPipe(pipeAdv,timeout
       );
4  outPipe.send(msg);
5
6  // Pipe must be closed AFTER the message is retrieved at
       the other end
7  outPipe.close();
```

If bidirectional pipes are to be used, the connection implementation varies a little, however message retrieval and processing stay the same.

```
1  PipeMsgListener myListener = new BPOutput();
2  JxtaBiDiPipe MyBiDiPipe = new JxtaBiDiPipe(NetPeerGroup,
3  BPListener.GetPipeAdvertisement(), timeout, myListener);
4
5  // Check if the connection is successful
6  if (MyBiDiPipe.isBound()) {
7      // Sending a message
8      Message MyMessage = new Message();
9      MyBiDiPipe.sendMessage(MyMessage);
10
11     // Closing the bidipipe
12     MyBiDiPipe.close();
```

```
13 }
```

4.3 Apache ZooKeeper

4.3.1 Overview

Apache ZooKeeper is an simple and open source solution to the top of the most common requirements in distributed systems, such as configuration management, synchronization, group management, naming, presence, consensus...

Isolating those issues from the programmer, ZooKeeper provides a very simple and reduced interface which allows the developer to focus on his own implementations ignoring the development of these support protocols which are usually required on most applications running on a distributed system.

A general overview of how Apache ZooKeeper works is covered in this chapter, followed by a justification of why was ZooKeeper chosen and the implementation that was used in this project.

4.3.2 Infrastructure and configuration

Even though Apache ZooKeeper is aimed towards the development of distributed systems, ZooKeeper itself is intended to run on a distributed environment where every server machine is connected to each other. This is to allow the system to be replicated, not only giving the service more resilience, thus as long as the majority of the servers are available it will work consistently, but also boosting its read capabilities.

Every time a write is issued the operation doesn't finish until at least half plus one of the nodes have successfully updated it, this ensures the recovery constraint regarding reliability. This makes the write operations about ten times more time consuming than read operations, which not only can't produce inconsistencies in the data, but also benefit from the data being available in more than one node allowing for load balancing improvements.

ZooKeeper works in a complex synchronization environment, therefore optimal network performance is required. It is highly recommended to have all the nodes in the same fast network bounds or cluster, but at the same time with the sufficient independence in case of hardware faults, to avoid scenarios such as switch failure taking the whole ensemble down. For similar reasons it is obvious that the service should run in dedicated machines to maximize performance.

For the project executions a standalone ZooKeeper server has been used since there is no difference between using one or more servers other than performance. Even so an ensemble with more than one node (up to 4) has been tested successfully, the configuration for a system with multiple nodes is done as follows:

- The file **/conf/zoo.cfg** in the ZooKeeper root directory has to be completed as shown below, and shared (copied) for every node in the ensemble, therefore any change in the quorum will require an update of this file for every node in the system (remember every node has to connect to each other).

```
1  tickTime=2000
2  dataDir=/var/lib/zookeeper
```

```
3  clientPort=2181
4  initLimit=5
5  syncLimit=2
6  server.1=zoo1:2888:3888
7  server.2=zoo2:2888:3888
8  server.3=zoo3:2888:3888
```

  - **initLimit:** Timeout value for the nodes to connect to the ZooKeeper leader at bootstrapping
  - **syncLimit:** The allowed time gap between ZooKeeper nodes, failing to satisfy this restriction will reset the node
  - **ticktime:** This is the unit of time that will be used by the other settings of this file (in this case one tick equals 2000 milliseconds)
  - **List of hosts:** The end of the document is a list of addresses and ports of all the nodes that compose the ZooKeeper ensemble (including the address of the executing node, since this list is common for all of them). The first port is used for connection management between the leader and the slaves, the second port is used during a phase of Leader Election as a free and parallel connection.

- In past versions of ZooKeeper, each server in the ensemble had a unique identifier that fell between 1 and 255 and had to be manually set in a special file of the chosen *dataDir*. This is being mentioned because with the latest versions of ZooKeeper since this project was started, this is

automatically managed, leaving the first step as the only configuration requirement for the ensemble.

4.3.3 Data model

ZooKeeper manages the data in a hierarchical tree composed by znodes. Since this service is supposed to be a coordination solution rather than storage means, the data nodes have is of reduced characteristics, and limited to 1 MB.

This is a common list of the information a znode stores: the first row is data (which is untagged), control information, version of the data...

```
urn:jxta:uuid-333FAB17B6CF48288A34ABC3F75D7BEC02
cZxid = 0x7da
ctime = Wed Jul 18 16:59:17 CEST 2012
mZxid = 0x7da
mtime = Wed Jul 18 16:59:17 CEST 2012
pZxid = 0x7e1
cversion = 2
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 48
numChildren = 0
```

Every access to the data, be either read or write, will be processed atomically. Reads will return either everything or nothing, and writes will either succeed or fail. On top of that, this applies to the whole ensemble as an individual system.

Znodes are referenced by paths which have an Unix-like structure (separated by slashes), they are unique, and also canonical (two different routes can't bring to the same znode), so resolution algorithms are not required. Every path begins at /, which is the root znode.

ZooKeeper provides a very basic, strict and sturdy structure, so gimmicks such as automatic reconfigurations are not provided (for example, rearranging a node's children after the parent's deletion). Any operation that requires secondary actions must be either checked or arranged in advance.

Znode types

Znodes can be either persistent or ephemeral, and once set this characteristic is absolute. Ephemeral znodes are continuously linked to the physical client with a heartbeat that will destroy the node in case the keep-alive is cut, whilst persistent znodes will remain on the system unless an explicit delete is issued. There is one restriction for ephemeral nodes, they can't have any children.

Ephemeral nodes are particularly useful when dealing with applications that require a certain level of availability awareness, we will see later on how this helped to deal with the Leader Election requirements of the project.

Znodes can be created with a sequence flag, which will create the nodes with an appended numerical identifier, for which ZooKeeper guarantees uniqueness and sequentiality. Altogether with the ephemeral znodes functionality, This is another fundamental piece that will allow the implementation of Leader Election solutions.

4.3.4 Interface and primitives

Operations

ZooKeeper provides nine basic, well-aimed and extremely simple operations with which the developer can create a wide number of applications towards distributed environments. Those come in different methods allowing for synchronous or asynchronous calls feedback, watch setting, optional data setting, version checking, etc.

- **create:** Creates a znode (if parent exists)
- **delete:** Deletes a znode (if no children)
- **exists:** Tests the existence of a znode, metadata is returned
- **getACL and setACL:** Gets/Sets the ACL of a znode
- **getChildren:** Returns the list of children the znode owns
- **getData and setData:** Gets/Sets the data associated with a znode (max. 1 MB)
- **sync:** Synchronizes a client's view of a znode with ZooKeeper

Watches

Watches are a notification method by means of subscription that allows znodes to be alerted when certain operations take place on the watched node.

Those can be set at any time and will trigger just once, so reregistering the watch is required if continuous feedback is required. Full configuration

is available, allowing the client to register for specific nodes and operations, they can even be set when performing another unrelated operation by piggybacking the watch order on the primitive.

Versioning

Every time a node is updated its version is increased by one. ZooKeeper's set of operations always comes with a *version* parameter, which is checked against the znode the primitive is executed on, if the numbers differ the operation will fail (requiring to get a new version and retrying). This behaviour can be overridden by putting a version number of -1.

ACLs

Every znode has an Access Control List (ACL) which determines who can perform operations on it. Those can be extensively configured allowing major access control settings.

Access can be denied or allowed by user/password authentication, by host or by IP address.

4.3.5 Command line client

ZooKeeper comes with a handy console based client which provide the nine operations in a manner that resembles a Unix environment file system.

To run the client the following command must be executed from the ZooKeeper root directory:

```
1 bin/zkCli.sh -server server_ip_address:server_port
```

This tool has been extensively used during the development on the project for testing, live modifications and validation of the system; just with the *ls* (list) command one can have a wide vision of the status of the system.

4.3.6 Implementation

Even though ZooKeeper was added into the project as a medium to solve just Leader Election cases, as the development progressed it was seen that it could be used for a few more issues that would have been too troublesome, if possible at all, with JXTA alone.

Not only that, but as the project advanced and it became more clear that even though ZooKeeper is simple and small, it offers a very powerful basis for growing as much as the developer wishes. So after evaluating the system and reviewing the implementation and concept, a conclusion was reached where a system without JXTA and a more extensive of ZooKeeper could possibly be a way more elegant and robust solution (as seen in the final thoughts 8.2.3).

To completely abstract ZooKeeper from the project functions, a class was created which acted as bridge between the ZooKeeper interface and the logic of the developed system, this being *ZkConnection.java*.

Having in mind that the implementation would use ZooKeeper mostly for group management, the ZooKeeper primitives were wrapped in methods that in the end would do the same action but in a more understandable and customizable way. A very simple example of this is the function that makes a node leave a group, which would translate to ZooKeeper as deleting the path that represents this node.

```
1 public void leave(String nodePath) throws
    InterruptedException, KeeperException {
2     zk.delete(nodePath, -1);
3 }
```

Below is a list of the methods this class implements divided in groups according to their biggest role.

Connection

Connecting to ZooKeeper is usually straightforward, but an improvement has been made for better consistency. When creating the instance in *connect()* via the *ZooKeeper* constructor, the object is returned immediately, but connection and bootstrapping processes have already been started in another thread. Starting to launch operations with this newly created instance will fail until the connection has been completely finished, which is not guaranteed after the object creation.

To do so, a watch is set on the object instancing instant, and it won't be until the watch feedback is received by the *process()* callback function that the connection won't be considered as established. Once the event *SyncConnected* is received other operations will be allowed on the new object.

- void connect(String hosts)
- void process(WatchedEvent event)
- void close()

Group operations

Most the operations here are self-explanatory by their given name, however some further details are worth mentioning.

Since this project doesn't require access control this feature won't be used, so all znodes will be created with the default setting, that allows full and public access (chosen by the ZooKeeper flag *OPEN_ACL_UNSAFE*).

Every znode that represents a client node will be ephemeral and sequential for the Leader Election algorithm to work, that makes the ZooKeeper *create()* operation in the *join()* method to have the flag *EPHEMERAL_SEQUENTIAL*.

On the other hand, since groups of nodes are perpetual and not directly related to any node liveness status, those will be created with the *PERSISTENT* flag in the *createGroup()* function.

Znode versioning is not crucial in this project because there is few updates, and they're always guaranteed to be made by an unique client. Ephemeral nodes don't even take part into a versioning environment. Given those two facts, every update will be forced with a version of -1.

Most exceptions are propagated with *throws* rather than dealt with, since functions can be used in different situations and different actions might be required to take in case of failure.

Watches via piggyback won't be used at all because it is not convenient for this implementation. There is a specific function, *watchExistence()* to put a watch on a specific node.

- String join(String groupName, String memberName)
- void leave(String nodePath)

- boolean groupExists(String groupName)
- void createGroup(String groupName, byte[] data)
- void deleteGroup(String groupName)
- byte[] getData(String groupName)
- void setData(String groupName, byte[] data)
- List¡String¿ getChildren(String groupName)
- void watchExistence(String nodePath, Watcher watcher)
- int getNumberOfNodesInGroup(String groupName)

Group management

When it was decided to split groups into multiple instances of the same group, a counter became necessary. There is other ways for this to be done, such as iterating the whole group instances to see which one has the largest index, however for convenience it was decided the index would be stored in a counter for each group, in a separate ZooKeeper directory which is */indexGenerators*. Those can be read or requested (and internally increased) by the methods listed below.

The only occurrence of a version sensible operation is found in *getNewIndexForGroupTag()*. It could be that two nodes read the same data version at the same time thus producing inconsistences, this is why this function has two steps. The first phase is to get the requested index value, keeping the version of this one, the second is to increase the value by one using the version that

was just retrieved. If the update fails, the retrieved index is discarded and both steps are repeated.

- int readBiggestIndexForGroupTag(String groupTag)
- int getNewIndexForGroupTag(String groupTag)

Chapter 5

Putting it together

5.1 Introduction

This chapter explains how the technical tools and available capabilities were used to achieve the defined and final architecture, it basically links the system architecture 3 and the technical background 4 chapters.

Two big support systems were used to complement this project, JXTA and Apache ZooKeeper. JXTA was planned in the project from the very start, however ZooKeeper, the Apache's solution to coordinating distributed systems, was started to be considered as an asset of the work by the time leader election situations started to appear, in the stages of the chain structures.

As will be seen in this chapter, ZooKeeper was meant to be used for the sole purpose of reaching consensus for leader election issues, but ended up taking place in other aspects of the implementation that couldn't be covered with alternatives, unwittingly ZooKeeper proved to be the most useful addition to the project as it advanced. In fact, as can be seen in the conclusions 8 of the project, a whole similar system could probably be developed with a much stronger focus on ZooKeeper, without a P2P environment covered with simpler alternatives.

As a reminder, the structure to build is one consisting of a tree hierarchy where groups of nodes are placed in the different levels of the tree. Each group will have a dedicated node that will also have presence in the superior level, allowing the flux of information upwards. Nodes always enter the system from the bottom levels, and get promoted up as required (be it due to crashes or space limitations).

Considering that the main project goal is to have a platform that allows to have different configurations for the structure, it has been made so that it is very easy to change the nodes each group allows to host. Besides that, changing any other parameters that are common for any configuration but that can affect performance in other ways can also be easily changed with isolated code definitions.

Over this chapter there will be some graphics showing the sequence the system follows in specific situations. Those will follow the legend shown below, where each color shows an interaction with the different two major systems, or both at once. Grey indicates generic code that doesn't interact with those, and an soft item marked by a slashed line indicates the code is there but is not currently being used.

Figure 5.1: Legend

5.2 Overview

5.2.1 Main loop

Every node joins the system in a homogeneous manner, it is later that the structure itself decides if the node should be granted special tasks or stay as a simple client.

Other than joining the base group as can be seen in the next section, every node will be in a never-ending loop of four tasks. Those are detailed

in their own sections, but complete the following execution sequence:

- **Publishing:** The node's contribution to the system, be it either self data or an aggregate report
- **Gathering:** The network request to update the remote information from other nodes
- **Structure check:** A chance to review if the structure of the system is good enough in terms of balancing
- **Promotion check:** A safe check to see if the node should change its role

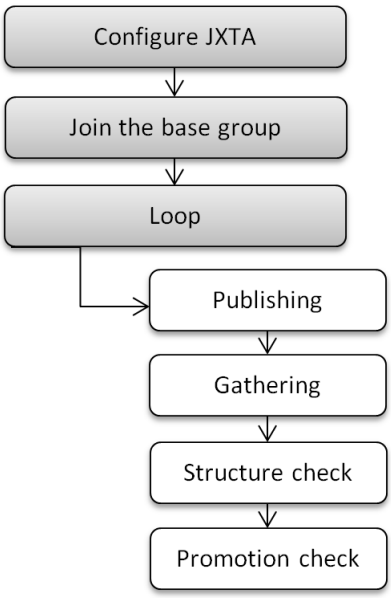

Figure 5.2: Main loop

With this general execution environment explained, the following sections will detail the more specific and internal implementation of the pieces to make the main loop possible.

5.2.2 Group management

To implement the groups and everything related to them, JXTA simple PeerGrouping will be used backed up by ZooKeeper.

When nodes join the system, they are manually given a group to join (for ease of management it could be a container or cluster identifier, but anything is fine). Those groups will be at the bottom of the tree, and every node entering the structure will always pass through them. If the group to join is full, an empty copy of the group will be created to host more nodes, so if the group to be joined was *CLUSTER1*, the first nodes would join *CLUSTER1_0*, and once it was full, an overflowing node would create and join *CLUSTER1_1*, at the same level. The two leaders of those groups, would then be promoted to *LEVEL1*, and following the same ruleset the exact groups they would join would be *LEVEL1_0*, *LEVEL1_1*, or whichever was the first group to have slots.

Peers know which level or leaf-group they need to join, however they don't know the exact instance of it (the last number in their tag). To allow those to know their destination, there is two possible ways of doing this: with JXTA peer services, or with the ZooKeeper interface. Both methods have been implemented and tested, however ZooKeeper proved to be more resilient, faster and trustworthy, so even if the code shows starting services (such as *AgentService*), they are not being used.

To check group availability, the node will connect to ZooKeeper and check the groups database (more details on this database will be given in the Agent node selection section, since it is what it was originally built for). ZooKeeper

actively tracks the nodes' status with heartbeats, so any count returned by this service is reliable enough, while with JXTA it would be more of a vague approximation.

When a group with space is found, the interested peer retrieves the JXTA PeerGroupID from the group representation in ZooKeeper. This piece of information doesn't necessarily need to be stored there, but it speeds the process much more than if the peer had to look for a valid advertisement among the whole network. With the PeerGroupID, the node can reinstantiate the group advertisement, and with it join the JXTA group, altogether with signing up at ZooKeeper's database.

Figure 5.3: Joining group

As a late experiment it was decided to use *index generators* for the level instances. To reduce load on the ZooKeeper service servers, a special znode directory was created to store specific data for each level or base group, which

contained a counter to track how many instances were created so far. This can be used to check the largest instance of a level, in order to iterate them without overflowing; and to check which should be the next subgroup index in case of group creation. The get-and-increment function has been coded specifically to perform in atomically having in mind the versioning in the ZooKeeper operations.

5.2.3 Information flow and messaging

Every messaging component in the system will be implemented with JXTA advertisements. Other methods such as services and pipes have been implemented and tested, and not only they proved to be more difficult and troublesome, but they were bound to performance issues.

Custom advertisement types such as *ReportAdvertisement* and *RestructuringAdvertisement* had to be created in order to achieve some of the message requirements. This also made the lookup and filtering of advertisements much easier, since applying filters by type could focus directly on those.

When a group agent, has to aggregate information from its group members, there are two basic options to achieve so: either the members of the group push the information to it, or either it pulls the data from each node. Any of those actions can result in bad performance, specially if using pipes or services, but with JXTA's exploiting on advertisement propagation they can be greatly mitigated.

In regular intervals, nodes will collect information and put it in a JXTA advertisement publishing it locally (remote publishing is also being used but

its reliability is inexistent). At some point, when the agent of the group is interested in reading it, it will issue a remote gathering order so that it can fetch the advertisements. This process exploits the peer-to-peer characteristics much more than using direct connections, making use of every node's resources.

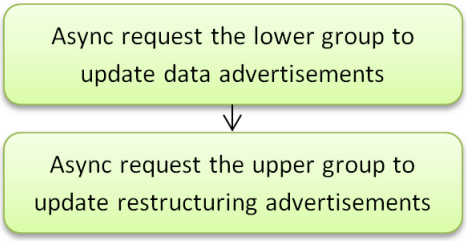

Figure 5.4: Gathering data

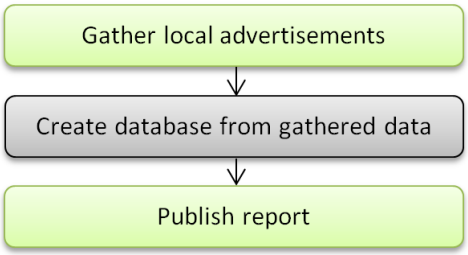

Figure 5.5: Publishing reports

One drawback of this procedure is the amount of advertisements being published. The document's expiration settings have to be set carefully depending on the frequency of publishing. And not only that, but to overcome possible network fluctuations and delays they will be versioned, so the receiver agent will keep a list of the peers issuing advertisements and the latest version they have discovered.

A side effect of the versioning system is that versioning is completely in charge of the issuing node. This means that any crash, that would leave the

node at version number 0 would generate an inconsistence that would render this node's reports useless as long as it didn't reach the latest global version, or old ones with higher versions didn't expire (which wouldn't work either way since agents have a versioned database of the found advertisements). To fix this issue, nodes will check the network status by discovering advertisements and keeping the largest version found when any major change such as a promotion is produced.

The information following this method won't be as up to date as if direct communication was used, however this doesn't prove an inconvenience for the system since critical reports were not a requirement, it is a few seconds after all.

5.2.4 Agent node selection

Each group in the system will have only and only one agent at a time, reporting information from this group up in the tree. To achieve this, a leader election protocol was required.

A very simple but solid algorithm was used to tackle this consensus issue. This had to be executed independently for every group in the structure, so for each group in the system a znode (ZooKeeper node) was created, it would act as a directory host for the agent-candidate nodes, and the agent itself. This directory would be persistent, and it would also keep the JXTA PeerGroupID in its *data* slot.

By using the *SEQUENCE—EPHEMERAL* flags when creating the znodes they would automatically be ordered by ZooKeeper, and those znodes

would be deleted as soon as ZooKeeper's configurable heartbeat detected a connection cut. The smallest ranked znode is to be leader of the group, so every other node should watch it and in case of receiving a crash trigger, check if it's the smallest node in the group and become the new leader.

To avoid a huge spike on ZooKeeper in case of failure on the smallest znode, instead of having every node watch the same one, every node will just watch the node immediately lower in identifier than itself, making it only one watch trigger in case of crash. If a watch is triggered, the node gets flagged as leader, and as soon as it reads the change in the main loop it is bound to promote itself. This also implies that if a node that isn't the leader crashes, the node that was watching it will have to update its watch to a new znode.

Unlike the need of checking which instance of a group or level a node has to join, they don't require to check up which level they are being promoted to, because promotion is always upwards and by unitary steps, so if a node is at *LEVEL4*, the level to join would be *LEVEL5*. The only node that doesn't promote itself is the unique root node in the tree, which instead, publishes a final report or issues commanding information downwards.

This ZooKeeper configuration allowed to ease some other tasks such as checking the number of nodes in a group, or to list level instances as could be seen in the first section of this chapter.

It also served as a great way of debugging to have a visual and clear view of the system's structure since ZooKeeper is reliable in terms of node's presence. Having a list of groups and alive nodes in each group, it is very easy to abstract the system status and structure without recurring to JXTA (which would be troublesome and unreliable).

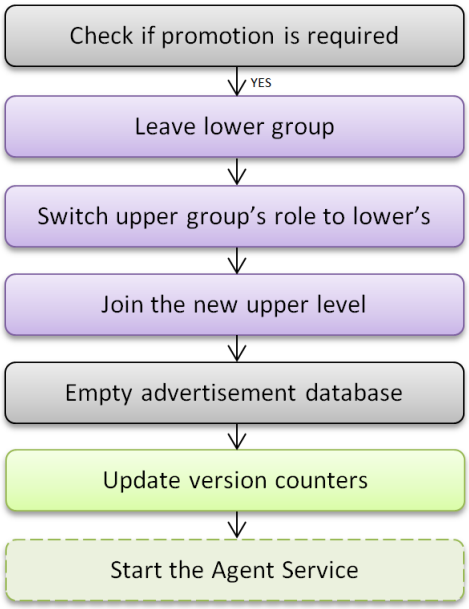

Figure 5.6: Getting promoted

5.2.5 Metrics collection and processing

In this project and for the executions that have been made, the information that has to be propagated or analyzed by the system will be left as XML files in a specific disk location, to be read by the application and processed into the JXTA advertisements.

The system is ready to gather information from the system, and to process it. It is up to the developer using it though, to decide how to deal with this information. Depending on his interest on performance, or the desired level of detail, it might either be convenient to merge the data into aggregated reports as they move up, or to stack the reports so that in the end the final report is a compilation of every single report in the system.

Chapter 6

Evaluation

6.1 Introduction

To test the system works as expected, two major configurations have been designed. The main idea is to see how those build up, and once they are stable make some changes in the system to see its behaviour.

In the beginning a more realistic environment where true application placement took place was considered, and in fact some configurations with Apache HTTP Server connected to multiple Apache Tomcat instances were configured, with a far goal to have a multi-node Cassandra system. However this support work was taking too much time that in the end didn't provide any difference when compared to exchanging dummy information, so major complementary configurations of other systems were dropped, to use instead void data.

The build will work in any Java enabled environment and it has been tested in different ones, however the experiments listed here have been in machines running Linux. The worker nodes, where the applications would be running in a real scenario, were all part of a small and private LAN cluster with one gate to the Internet.

Additionally, a single remote server was used to test if network boundaries were being successfully overcome. This server hosted a very basic JXTA implementation that would act as a generic rendez-vous node, this would be the configured seed for the other nodes that would serve as the point of entry to the system.

This remote server is also the place where the whole ZooKeeper system was running as stand-alone service. As explained before in this document

ZooKeeper was tested in a multi-node configuration, specifically in the small cluster, however to have the cluster devoted to the JXTA nodes it was considered best to leave the JXTA rendezvous and ZooKeeper running together in a single, remote node.

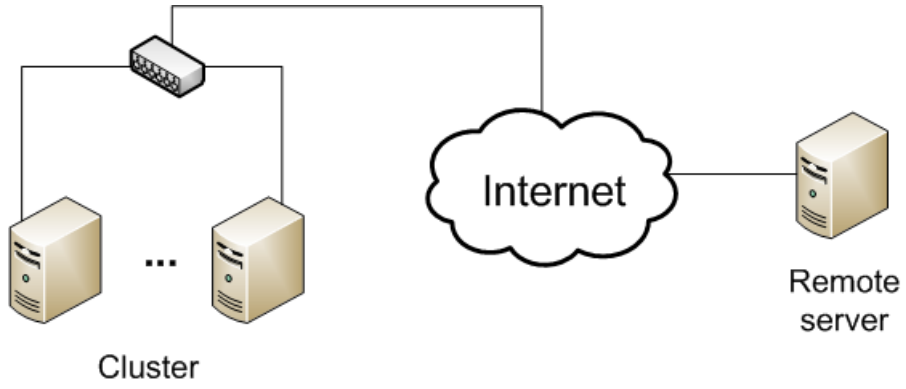

Figure 6.1: Network configuration

The aim of the experiments is to prove and confirm the system is working as expected, the most relevant behaviours to be seen are the following:

- When a configuration is defined for a configuration, it has to be under those constraints at any given moment of the execution.
- The flow of information must be correct, nodes must publish and gather information in the groups they are in, and have a valid propagation upwards.
- In case of node failure, the system has to recover, to have a correct structure that still collects information from every node converging to the root.

In the following sections two representative experiments from the many

that were executed are shown in detail, proving the objectives were met and that all the given situations are working as expected.

The testing scenarios come with support figures, and the legend of those is as usual, groups are represented with circles, nodes in groups as dots (two dots joined by a line if the node has presence in two groups, one where it gathers information and the other where it reports it), and an R representing the root node who is in charge of giving the whole aggregated report. Nodes painted in red are under a failure scenario and will be disappearing in the next step. The nodes have no tags next to them to have cleaner pictures, so for easy identification the order in which they appear doesn't vary among the steps.

6.2 Configuration 1: 3 nodes per group

The goal of this test is to see the gradual growth of the system as nodes connect to it, under the constraint of 3 nodes per group, and altogether with a correct flux of information. Once a stable configuration with four nodes has been achieved, they will start getting killed one by one to simulate failures so that the recovery of the structure can be tracked in detail.

6.2.1 Structure construction

The figure 6.2 below shows the process that is described step by step right after it.

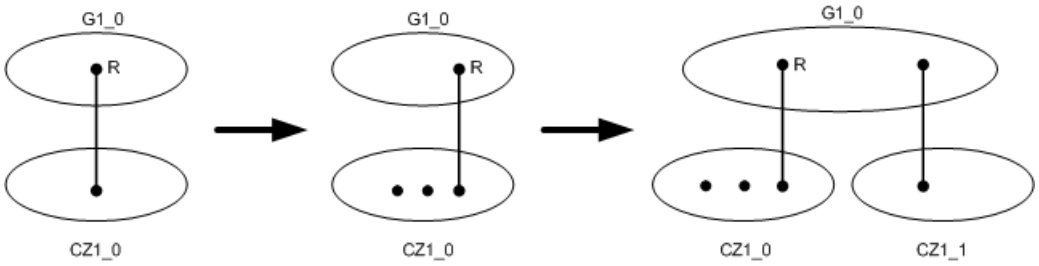

Figure 6.2: Build-up progress of the first configuration

First node entry

The first node that enters the system starts by looking for available rendezvous from its seeds, and successfully connects to the one in the remote server (every node will follow this procedure).

Once connected to the basics of the network, it will join the group it has been assigned which is *CZ1*, since the structure is empty, the system assigns it the specific instance *CZ1_0* and seeing it doesn't exist, the new peer creates and joins it, immediately detecting it has to be the leader of this one-member group.

As soon as it joins it starts publishing information in this group as slave, and as a leader it has to promote itself to the upper layer, which is the group *G1_0*. Just like before, the group doesn't exist so the same procedure is followed and again the node is leader of this group, though this time its role on the upper level will be as root reporter of the system.

At this point the node will be checking the lower group for reports (and it doesn't find anything since there is no nodes other than itself), publishing its own information in the upper group and, as root, gathering information on the upper group and reporting it (there is a single node in the root report,

its own).

Second and third node entries

The second to join will follow the same steps, joining the already existing group, and will immediately start publishing his own data on the upper group (remember that for leaf nodes the upper group is their only group, in this case *CZ1_0*).

In the previous section the node was not gathering any reports from the bottom group since it was empty, besides his own presence that was being reported to *G1_0*. With this new addition to the system, the first node detects reports in the bottom group of *CZ1_0* and starts propagating information up to *G1_0*.

Looking at ZooKeeper two directories can be seen for the two groups that have been created so far. Listing their contents it can be seen that *G1_0* has one znode (the current root leader), and that *CZ1_0* has two znodes (the first node that is leader on the lower group, and the new node which is watching it).

The third node will behave exactly like the second, effects can be seen in ZooKeeper's new appearances, the report counters increased from the first node, and this new node watching the second one at ZooKeeper.

Fourth node entry

This is the last node to enter the system in this configuration. With this a new *horizontal* group is required, since it is told to join *CZ1* but *CZ1_0* is full on 3 nodes, so when it attempts to join *CZ1*, it is directed to *CZ1_1*,

which has to be created and joined.

By doing so, the node becomes leader of the new group and just like the first node, is promoted straight to *G1_0* which now is existent, so the new peer will sit *watching* the root.

Meanwhile, it will be gathering information from *CZ1_1* without success due its loneliness in this lower group, and reporting information to *G1_0*, which will be successfully collected by the first node as can be seen from the root reports' counter.

6.2.2 Fault tolerance, crashes and recovery

With the system of four nodes that has just been built, nodes will be killed one by one in the other they joined the system, that being second, first, fourth and third. The process is detailed in the following figure 6.3 and described afterwards.

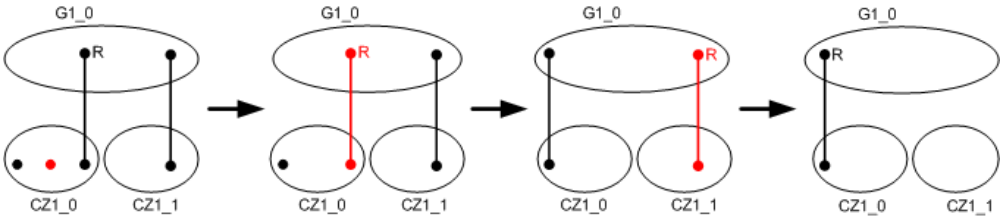

Figure 6.3: Failure and recovery of the first configuration

Second node failure

When cutting the execution of the second node that joined the system, its presence instantly disappears from the ZooKeeper services triggering the watch on the node that was watching it, which was the third that will change

its watch to the leader, that was being watched by the node that just went down. Since it wasn't a leader of any group, no major structure changes are required.

At the JXTA rendez-vous console a TCP connection close can be seen, however this is only informative in a very low level, since the JXTA network still has information relating to the node that just went down (in fact if in the same console peer advertisements are checked, it stays there indefinitely).

However it can be seen that even though JXTA isn't being very consistent, as any other conventional peer-to-peer system would neither be, the implementation of the JXTA part of the application is behaving satisfactorily as can be seen after a few updates, where leader reports are decreasing their advertisement count by one, specifically the recapped report of *CZ1_1* where the faulty node was.

First node failure

Killing the first node, not only one leader but the root leader is breaking away from the system. As with the pervious *crash*, ZooKeeper's awareness is almost instant, and very soon, both the third node in *CZ1_0* and the fourth node in G1_0 detect it went down (note that unlike the second node, this one had presence in two groups, and being root it was leader in both of them).

The first will find that it has to become leader of *CZ1_0* since the leader just disappeared and he is the last member of this group, therefore it will proceed with a promotion to *G1_0* to watch the other survivor in the system.

The fourth node also detects the leader is gone, and finds out that no promotion is required because it has to assume the role of root, and as soon

as it realises this, it starts gathering reports and announcing root information.

Since the first node just joined *G1_0*, and the fourth is the root leader of this same group, the root reports from the fourth will reveal information from both itself and the first node.

Fourth node failure

As the fourth node leaves the system, the first, which is the last one remaining, takes leadership of *G1_0* and assumes root status, reporting only the information that comes from its own publishing.

6.3 Configuration 2: 2 nodes per group

With this test configuration the general goals of the previous one will be confirmed (structure build-up, correct information flux, promotions...). Additionally, the group size will be limited at two per group, this will allow the structure to have a faster vertical growth, which puts the system under a different pressure than the previous one. In a regular, large configuration, the common would be to have many nodes that belonged to two groups, but given the reduced set most of them assume special roles such as bottom leaf nodes (which only have one group unless promoted), or root (which virtually has three groups, although there is no promotion or real existence for the third). By having a faster growing architecture there will be more space for some of the peers to act as regular links between two groups without having special tasks, allowing the system to have to up three group levels instead of just two like in the previous experiment.

It is worth mentioning that the structure being analyzed here is fully experimental and for testing purposes. Having two nodes per group is a valid and working configuration which offers many interesting changes to evaluate, however it proves to be the minimum possible structure.

A structure that allows one node per group is not possible, since groups are linked from a higher and a lower level, making they require at least two slots for those connections. Having those two available like in this experiment makes it possible for something valid to be built, but the structure efficiency is terrible, so it is not an expected configuration to be seen in production servers.

6.3.1 Structure construction

Since the growth of this configuration is more complex than the previous one and fault tolerance was already checked, this experiment will focus on the building of the system. Two figures (6.4 and 6.5) following the progress can be seen in between the explained step by step analysis.

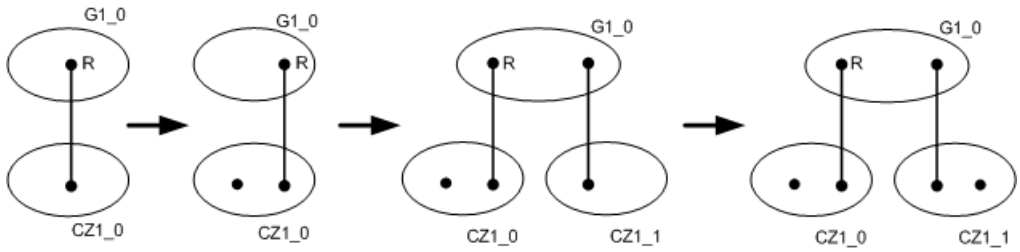

Figure 6.4: Build-up progress of the second configuration

First node entry

The first addition to the system is trivial as usual, the node joins its given group *CZ1_0* and promotes to *G1_0*, becoming leader of the system.

Second node entry

There is no secret to the second node joining the system either, it joins *CZ1_0* and stays there watching for the possible failure of the first node. The new node will report information in *CZ1_0*, that will be gathered by the first one and reported up in *G1_0* altogether with its own information.

Third node entry

Given the small size of the groups, when the third node attempts to join the system it already is issued a new group instance since the previous one is full. The node joins *CZ1_1* as told, promoting itself to *G1_0*. The root reports will now show that two nodes are reporting to *G1_0*, those being the first and the third, with this late one watching over the first's leadership.

Fourth node entry

When the fourth node joins the system, nothing much relevant happens. It will be told to join the instance *CZ1_1*, which is a bottom group and already has a leader, so it will silently join this group and publish self reports, gathered by the leader of the group.

At this point there is four nodes in the system, filling all three groups in it. Two are in a single-group status at the leaf level (the second and the fourth),

and two are acting as a link from the leaves to the root group (the first being root himself, and the third). The leaf nodes are reporting information at the lowest groups of *CZ1_0* and *CZ1_1*, while the other two are gather collecting it and reporting to *G1_0*.

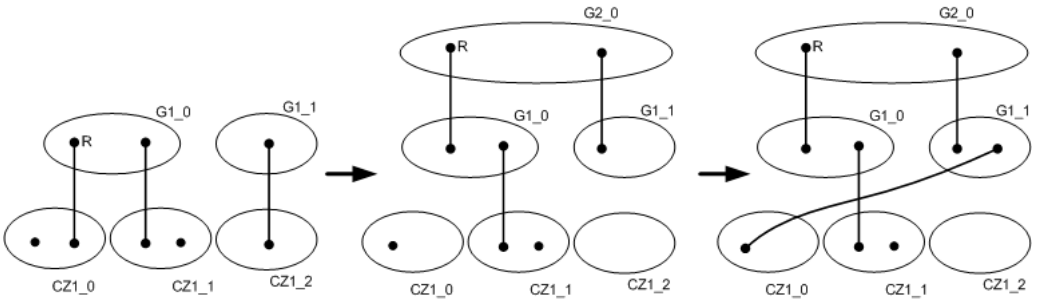

Figure 6.5: Build-up continuation of the second configuration

Fifth node entry

Up until now the other nodes have been joining the system with easiness, however with the fifth node major changes occur. The structure was full, so it is inevitable that a new group appears, and as will be seen the other nodes might need to emerge and reposition themselves to allow a bigger hosting for new entries.

During all the structural changes publishing and gathering of advertisements has been consistently working, however to focus on the interesting facts and not to lose the thread of the changes, they won't be fully explained until the situation is stable, in fact most of those actions are only done once since most states and roles are transitional.

The node, which has to join *CZ1*, creates and joins the instance of *CZ1_2* after seeing that *CZ1_0* and *CZ1_1* are full. Being alone it becomes leader

of this group by promoting to *G1*.

However the only instance of *G1*, *G1_0*, is also full, so it will be required to create and join *G1_1*.

At the appearance of *G1_1* more changes are required. In one hand, the leader of *G1_0* will detect its group is not a root group anymore, so it will have to step down from his root role, not only that but it will also detect the need of a whole new level, *G2*.

Meanwhile, similar to the previous case of its first promotions in the lower level of *CZ1_2*, the new node will find itself alone in the group and also promote up to *G2*.

At this point two nodes (the first and the fifth) will be promoting to *G2_0* in parallel, and the one that joins this group in ZooKeeper first, will be the leader of *G2_0* and the new root node.

In this execution the first node became leader of *G2_0* and therefore of the whole system, however it is important to have in mind this sensation of order is purely coincidence.

With the rapid promotion of the new node to the top it can be observed that nodes are always inclined to float up in the structure, leaving its first group *CZ1_2* empty, but available for possible newcomers.

Similar situations happen in the oldest branch of the structure where the first node who was acting as a link between *CZ1_0* and *G1_0* had to leave *CZ1_0* to get promoted as a link between *G1_0* and *G2_0*. In this case though, *CZ1_0* wasn't left empty, and the single node left there (the second that joined the system) detects the leader left the group, making it the first candidate. With this, it gets promoted to act as a link between *CZ1_0* and

G1, and it is worth seeing that the group it gets promoted to is not the same one where the old leader was (*G1_0*), but the next one which already exists thanks to the last node to join the system, *G1_1*.

This happened because, even though the leader left the bottom group, it stayed in the upper one, holding the spot that the last promotion should have taken intuitively. In the new group there is a leader already, so the last arrival will just be watching the fifth for possible crashes.

A stable configuration has finally been reached, and inspecting ZooKeeper it can be seen that every group has two nodes, besides *CZ1_0* which only has one, and *CZ1_2* which is empty.

The publishing actions don't give much information, however it stays consistent. Every node is publishing recaps of some way besides the fourth which is the only node with a single-group role. Every recap is made out of one node, since groups can host a maximum of two nodes, and the reporting one always publishes his own information on the upper group, so the recaps from the lower levels will never be greater than one. The root node is reporting recaps from two nodes, which consist of his own report of *G1_0*, and the fifth's report from *G1_1*.

Finally, without going into much detail since it is the same procedure over and over, killing the root node (the first one) makes the other node in the top level to become root as can be seen in the figure 6.6. The fall of this node has another effect on the branch it was fetching information from, where lacking a link from *G1_0* to *G2_0*, the third node who was watching the first has to promote itself. This will leave the fourth node alone at the bottom group of *CZ1_0*, which will promote itself to *G1_0* without much trouble since there

was a free slot.

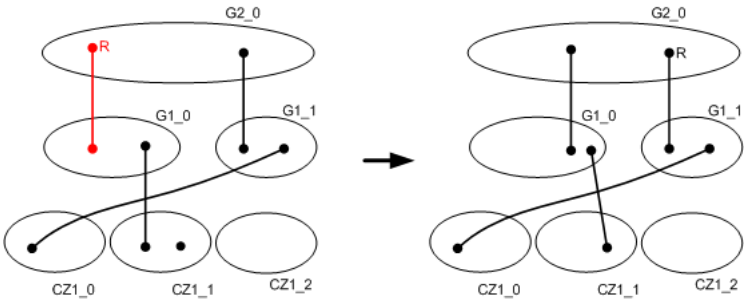

Figure 6.6: Failure of the root node in the second configuration

Chapter 7

Project management

7.1 Introduction

Given the nature of the project, where a strong focus on research takes place, project management might differ a bit from other works under circumstances with a lower risk factor.

This risk comes from the fact of trying to find a solution to a problem that hasn't been tackled previously. This involves a previous study of the possible ways to take, and a choice of support tools and frameworks. One could argue the use of third party software to reach an objective instead of fully developing a fully integrated system, but this is required to speed up the process and be able to focus on the most important part of the project, specially so in this one, where backtracking in case of reaching a dead end would suppose a major drawback.

Having all this in mind, a project plan will be given, but it is expected to be heavily modified as work advances to overcome possible setbacks.

7.2 Methodology

No specific methodology will be followed for the development of this project, however the essence of this custom methodology will resemble the well known the Agile development methods.

Overall, the Agile development methodology promotes teamwork and no long-term planning. Advancing with small work increments that allow the developers to consolidate or reconsider the work done up until then and will minimize the impact of a forced recoil.

When facing a problem, or previous to major implementation or decision making times, the following steps will usually be taken as a general rule:

1. Analyze the problem
2. Think of possible solutions and future impacts of those (affecting other components or generating more problems)
3. Put those solutions on the table with the rest of the team for a group analysis
4. In case of deciding to make use of third party software, invest extra time on estimating the learning costs and possible negative aspects of this choice
5. In case of approval the implementation will be done and tested (with previous isolated testing if required), and if satisfactory results are achieved, the procedure will be repeated for new problems

In the end this has proven to be a very successful way of working, specially considering the timetables that were to be taken into consideration and that are explained in the next section.

7.3 Work plan and scheduling

This project doesn't count with a full dedication schedule since the person in charge is undertaking college classes and working at the same time. This also means that during intensive times (such as college examinations or deliveries,

and work deadlines) dedication to this work might be pushed back, or even stopped until time allows to pick it up again.

In order to fit the described methodology, weekly meetings with the team will be established to discuss the decisions taken and as a general development checkpoint, this way any disagreement among the team can always be discussed in time, and next objectives can be set; it is also a great way to keep members that are not actively developing the system up to date.

Weekly meetings were constant, even in absence of any new development or advances in the project, to avoid losing the thread of the status, and even while doing that at times some days had to be dedicated to figure out at what point the project was left at, specially in long periods of stillness. All those up and downs sometimes proved frustrating for the developer, having to remember previous work or repeating already done parts of the project. Even though sometimes it is inevitable, it is of utter importance to be able to divide the work in independent parts and never leave one of those in a critical development point, in risk of having to backtrack to the very start of the task, documentation is not always enough.

The project started in October 2011 with a long-term aim to have some results by the end of 2012 considering the dedication wouldn't be full. A first basic schedule 7.1 was presented as a general guide, however as can be seen in the second schedule 7.2 it varied a lot in the real scenario, with a lot of task repetitions and overlapping (time units are split in periods of about 15 days). The major difference that can be seen is the expectations of time dedication to JXTA versus the final count. As it has been explained, obviously ZooKeeper and the other architecture were not in the plan since

they happened on the go.

The work took more turns than expected and given the insufficient dedication estimated during work periods, it was required to step out of the typical schedule and cover it up with extra and unplanned days.

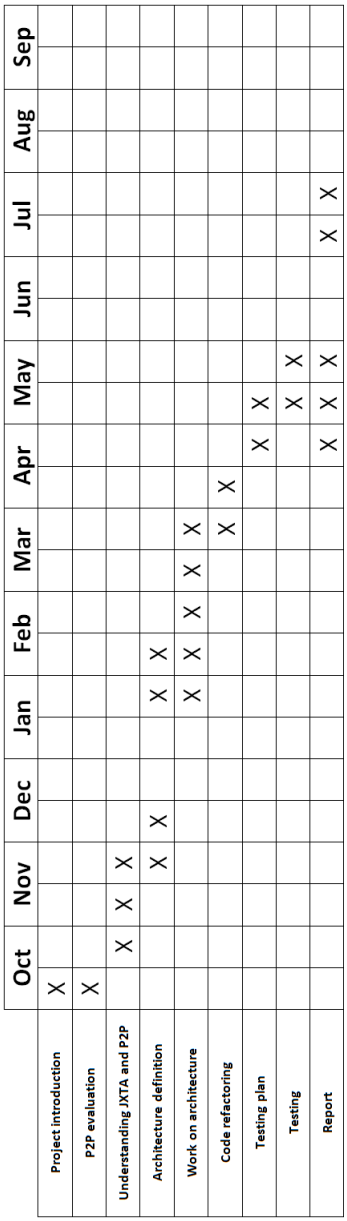

Figure 7.1: Approximation of the planned schedule

A list of the followed roadmap is shown below, note that those steps are in a general scale and each of them have its correspondent substeps, which aren't necessarily less important or heavy (in work terms), than their parent items.

Following the list comes a figure 7.2 with the schedule, where intervals are presented in two per month (15 days), and considering alterations between the hours per day, where most of the time the development was done in part-time.

- Meeting sessions to introduce the project, state the problem and define indicative objectives. Some explorative work was previously done on this field and this was analyzed and taken into account, a strong motivation to use a peer-to-peer system was put on the table from the beginning.

- Different P2P solutions were considered and in the end it was decided to go on with the candidate so far, JXTA. At this point, a specific JXTA roadmap was set with small testing objectives to get the developer familiarized with the new platform. Those objectives consisted of general testing but always aimed at what was thought to be useful for the project, such as group management and service implementations.

- Some dedication had to be put to the environment setup, cluster configuration and system administration. This was notable at the start of the project but it has been active for the whole progress of the work as systems changed, new requirements appeared, etc.

- Once enough understanding of JXTA was acquired, a period to come up with possible architecture was established, and multiple meetings were issued to debate and polish the options. Even though a final solution wasn't found (the point of the project is to try various architectures anyway), a core configuration was agreed.

- With the newly acquired JXTA experience and the architecture target, a more aimed JXTA development towards a real solution was started. This came up with multiple issues and setbacks by the peer-to-peer framework, which as a discontinued project lacked massive documentation and had many undocumented bugs.

- Besides the technical issues with JXTA, other requirements that couldn't be achieved with the P2P framework appeared, and a new time for analyzing how to overcome those surged. After some discussion it was decided to use Apache ZooKeeper, a service aimed to develop applications with strong distributed coordination characteristics.

- Following the same steps as with JXTA, a period of familiarization with ZooKeeper was programmed, and in this case the learning curve was much easier and faster than with the previous, the documentation and reliability of Apache's system excelled that of JXTA's.

- More meetings followed to review the new architecture models composed with ZooKeeper, and then solid development for the system was started.

- At this point the issues with JXTA were overflowing so much, that

alternatives were starting to be seriously considered, however given the time shortage and having a solid enough configuration it was decided to stick with JXTA, leaving it prepared enough so that if in a future a framework change was required it wouldn't suppose a major overhaul.

- At this point the tools for completing the project objectives seemed to be enough, and the development was progressing mostly only with worries concerning architectural aspects, which were covered in the regular meetings until the system was finished.

- With only code polishing left, a new set of meetings was agreed to plan the experiments set and final testing, parallel to the intensive report write-up which was partially started in the previous months.

7.4 Budget

7.4.1 Salaries

For a better understanding of the salary it has broken down into tasks and roles. Different worker profiles are required, and all of them should be proficient in working with abstract problems and with tools that aren't necessarily comfortable or well backed up with documentation.

The first table 7.1 contains the relation between tasks and assignments with the spent hours, while the second 7.2 lists the salaries for each of the roles with an aggregated value.

Task	Role (hours)
Project management	Project manager(80)
Peer-to-Peer evaluation	Analyst(20)
Understand JXTA	Analyst(150) Programmer(180)
Architecture definition	Analyst(100)
Work on architecture 1	Analyst(30) Programmer(70)
ZooKeeper understanding	Analyst(30) Programmer(10)
ZooKeeper implementation	Analyst(20) Programmer(40)
Work on architecture 2	Analyst(40) Programmer(100)
Refactor	Analyst(30) Programmer(40)
Test plan	Analyst(60)
Testing and setup	Administrator(120)
Total	1120

Table 7.1: Dedication per role/task

Role	Hours	Cost (€/h)	Total cost (€)
Analyst	480	20	5400
Programmer	440	15	7560
Administrator	120	12	2040
Project manager	80	30	1600
Total			20040

Table 7.2: Worker salaries

7.4.2 Software expenses

Every software solution used in the application is under a free open-source license, so the expenses on this field are nonexistent. The system it was tested on, and the whole platform where it was developed was also based on the same characteristics.

7.4.3 Hardware expenses

The physical resources required to develop and test this project were: a laptop to be used as workstation, and a distributed system for the testing and experiments.

The distributed system used in this project was a small cluster of less than 10 physical nodes, and most of the times only three to five of them were used. The table of expenses is an orientation to the minimum system that should be acquired to have a representative enough environment, however the college provided the team with a small cluster of those characteristics.

There are alternatives to having multiple servers, however moving away from a real environment when working with distributed systems can be very prone to trouble, and a huge investment in configuration. Virtualization is one possible solution, and another one, which comes with extra expenses, would be to use the services Amazon provides.

In fact, further testing using server instances in Amazon was considered to easily increase the number of nodes, however due to logistics and configuration, time availability and having representative enough results, the plan was disregarded.

Item	**Cost (€)**	**Units**	**Total Cost (€)**
Laptop	800	1	800
Node	500	4	2000
Switch	100	1	100

Table 7.3: Hardware expenses

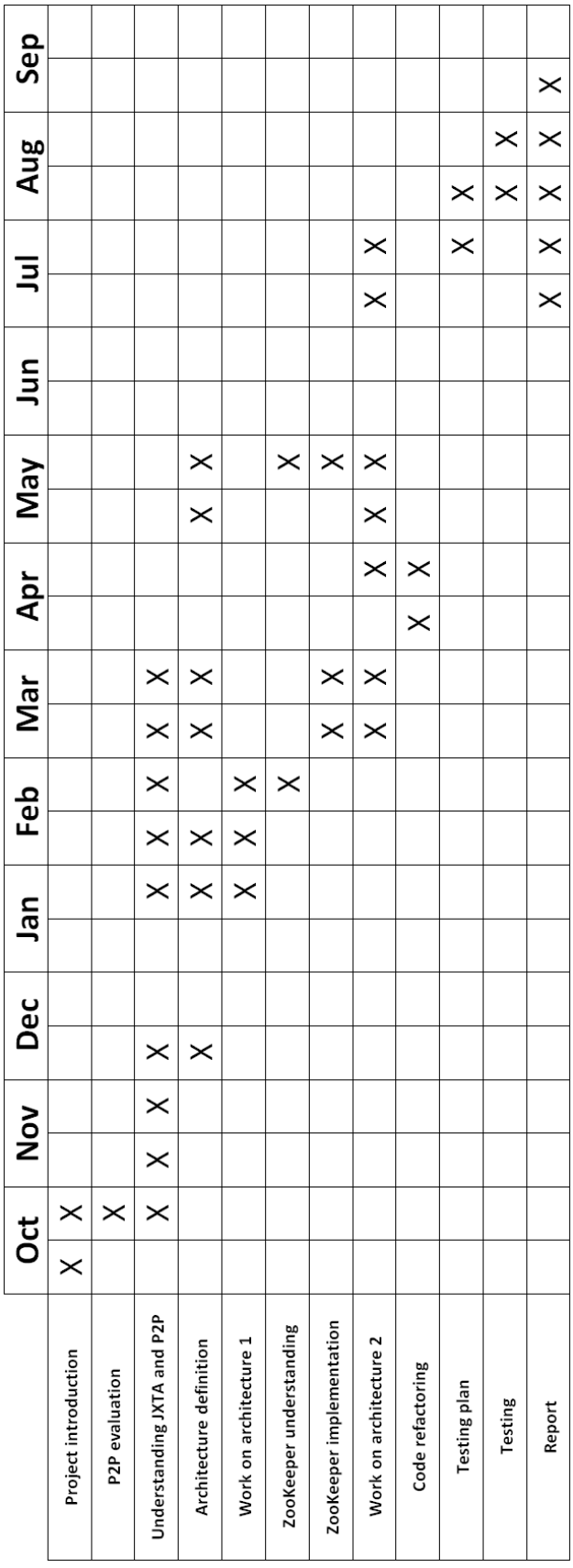

Figure 7.2: Schedule the project followed

Chapter 8

Conclusion

8.1 Summary

This document has presented the development roadmap of a system that can be configured to the settings that best adapt to a distributed system, and that can have its network parameters tweaked to determine the responsiveness and how up to date it is within a given snapshot. The system is meant to be used for monitoring purposes in a way that the ensemble doesn't collapse, however any information can be transferred with it.

Even though no specific network utilization metrics have been taken (and this could cover part of another smaller project, taking metrics of a peer-to-peer network among a distributed system is not a trivial task), it is safe to assume due to the nature of the peer-to-peer network model that a decent configuration with this project will not bottle-neck as easily as any other configuration with centralized data gathering points.

Besides the network improvements due to a more distributed and static nature of the data, it is worth mentioning that another important addition to the system is the fault detection of the nodes, and the recovery it answers with to keep a stable structure without needing to rebuild it all.

Since the application wasn't meant to be a product, no interface has been designed or created for it, however the transition of this software to a final product shouldn't require much effort, considering that it has proven to work for every scenario that has been tested, both documented and not documented in this file, and when choices of configuration presented, the worst case scenarios have always been chosen.

8.2 Final thoughts

8.2.1 Peer-to-peer networks in distributed systems

Peer-to-peer paradigms as a solution to the management of distributed systems has been a popular trend for the past decade, however this terrain is still too unexplored to deposit absolute trust in this field.

P2P architectures provide robust environments of ridiculous sizes without being centralized, and so far they have been found to be a great solution to content distribution networks.

However, managing a distributed system has different requisites, and peer-to-peer architectures are so dynamic it becomes virtually impossible to have a picture of the status of the network. Having an estimation of the network status can be enough for some cases, such as content distribution, but in this project this was a strong requirement that had to be backed up with ZooKeeper, altogether with unreliable communication, consensus scenarios and many uncertainty issues.

It has yet to be seen as more solutions come out, but it is worth considering if it is significant to rely on the peer-to-peer paradigm given the complexity and possibilities it comes with, and with all the drawbacks that have to be covered up with help from other perspectives.

8.2.2 JXTA alternatives

JXTA has been the most controversial decision along the project without doubt. Even though time was invested to decide which platform would be

best, in the end JXTA was chosen. This was because there were no real alternatives and some work with JXTA had been done before, however this previous experience with the platform proved to be too shallow to give any advantage in this project.

Having a difficult learning curve should not be enough to push back a developer from using a certain software, however this was probably the least important inconvenience. The impressions of JXTA were that it was a working project that kept getting new releases but later on, already deep into development it was found out that the project had been abandoned. Bugs kept appearing, there was inconsistence between the documentation and the builds, and the documentation was not only poor, but almost inexistent. Most information came from third parties, and most of it was experimental, in fact many of the sources that were saved during the project, were found to be gone when access was attempted later on. Every inconvenience that was encountered during the development could be fixed, or dodged by taking another perspective or alternative implementation, so the status of JXTA didn't have any effect on the project's quality, however there was an impact on the development where a lot of time and effort had to be invested to first discover those malfunctions, and then to decide and put a fix to them.

It is up to how adventurous the developer is, but using JXTA in the current status is not an easy recommendation. Depending on the will to keep working with P2P, FreePastry [2] is an open-source platform thought specifically to perform research and development in peer-to-peer networks and applications, and after some reading and information gathering it seems to be much more documented, active and successful than the discontinued

JXTA.

8.2.3 Exploiting Apache ZooKeeper's capabilities

Apache ZooKeeper was not included in the plans that were made in the beginning, however once it was spontaneously used to deal with leader decision scenarios, the platform appeared surprisingly good, light, and easy to use.

If opting for a solution that doesn't include peer-to-peer, it is highly possible that the structuring part can be achieved solely with ZooKeeper. As it was mentioned in previous chapters, many difficulties were solved with JXTA because the project wanted to avoid high load and responsibilities falling on ZooKeeper, but for almost every issue that too troublesome, or plainly impossible, to fix with JXTA, quick and clean solutions came to mind in the field of ZooKeeper, for it is a platform thought to coordinate distributed systems, and not just deal with leader election situations as it is mostly used.

The major disadvantage of dropping JXTA and delegating everything to ZooKeeper is that the project would lose its messaging capabilities, which are a strong requirement. This could be covered in many ways, from using plain network sockets, to using messaging libraries, which are more popular and accessible than JXTA since they focus on just messaging.

Further study should be investment to adopt this as an option, but at some points where JXTA was about to be dropped in the project, YAMI4 [3] was considered. This is a messaging set of libraries to help develop distributed system applications with the characteristics matching the ones in this project,

which is maintained and fully documented, as well as used by some known platforms.

8.3 Future work

Just as with any other research project, the possibilities are usually many, and this is no special case. The field in which this project is found is relatively new and poorly explored, and distributed systems are not that common in everyday use so that multiple alternatives exist where one can choose from.

Many implementation details that weren't crucial for a first version have to be reviewed, such as purging of obsolete advertisements which now are left around the system (this is not a functional problem since they're being ignored thanks to the versioning systems, but thinking long-term it is good for P2P networks if obsolete data is purged little by little).

Evaluation and testing could be much more interesting if it was done in realistic environments with many nodes, this would allow to confirm that some aspects of the system such as scalability are good enough. Long term execution of the system would also be a great test to see if there is degradation of the system performance over time. The same infrastructures could be used to evaluate what this project was really developed for, a comparison between architectures to analyze which configuration works better in a given distributed system.

Even though it is not a requirement to balance the structure since it doesn't have performance consequences, it is a common practice in such systems and needed for many reasons. A implementation on this issue was

done, and it wasn't precisely easy, so it would be worth it to have it further tested and modified or improved.

Even though the application is working without trouble with the default or most common network parameters, the system that has been developed is huge, and many of its parts should be analyzed to find possible optimizations. Many JXTA settings can be changed (such as advertisement life time, expiry times, update rates...) and the impact of those on the network performance can be massive. Besides those, many other internal settings can also be tweaked, for example the heartbeat rates of ZooKeeper. Special attention can also be given to the performance and the optimal rates of rendez-vous and relay peer nodes in JXTA per network, peergroup, etc. [1]

In small executions such as the ones that have been seen in this report, tracking on the system can be kept with relative easiness, however as the system grows it would become impossible for a developer to know what is going on with simple message prints on a console. This may seem shallow or purely aesthetic, but when dealing with distributed systems a good managing and informative interface is usually mandatory, and it is not usually trivial to implement.

Besides all that has been said until now, by reading the previous section of final thoughts 8.2 all those possibilities and combinations of them can be considered as deeper future projects so that they don't stay just as a mentioned possibility.

And after all those tasks, even if the perfect solution to this problem was found, many doors would remain open, and by closing one, even more always open. The amounts of data, data transfer rates and nodes, increase everyday

leaving not so old practices obsolete many times faster than what it took to reach them.

8.4 Personal note and acknowledgements

Personally, this project has served as a great introduction to a real distributed systems problem. The extended length of available dedication, allowed me to interiorize many issues and complications that under shorter circumstances would have proved much more difficult, specially with the gradual learning curve of JXTA.

I consider it a shame that there was so many periods of interruption during the development as can be seen in the real schedule, because each of these meant a step back on the development that required to invest quite a bit of time to recover the last checkpoint.

Nevertheless, the experience acquired in this field given by the development of this work is something that I consider of great value, seeing I couldn't learn to move around such environments in many other situations, and right now facing other similar problems I get the feeling I wouldn't look at them with the same eyes I put when I was presented with the one in this document one year ago.

I have to thank Yolanda Becerra, David Carrera and Jordi Torres for the chances and trust they have deposited in me, and their support and understanding during the interrupted course of this project, where the efforts to recover the latest checkpoint were not only mine but also shared with them.

Bibliography

[1] Emir Halepovic, Ralph Deters, and Bernard Traversat. Performance evaluation of jxta rendezvous. Technical report, University of Saskatchewan and Sun Microsystems Inc., 2004.

[2] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. Technical report, Microsoft Research Ltd and Rice University, 2001.

[3] Maciej Sobczak. *Programming Distributed Systems with YAMI4*. Lulu.com, 2012.

[4] Jérôme Verstrynge. *Practical JXTA*. Lulu.com, 2008.

[5] Navaneeth Krishnan Juan Carlos Soto Daniel Brookshier, Darren Govoni. *JXTA: Java P2P Programming*. Sams Publishing, 2002.

[6] Brendon J. Wilson. *JXTA*. David Dwyer, 2002.

[7] Bernard Traversat Li Gong, Scot t Oaks. *JXTA in a Nutshell*. O'Reilly Media, 2002.

[8] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.

[9] Jxta java standard edition v2.5: Programmers guide. Technical report, Sun Microsystems, 2007.

[10] Dr. Ian Wang. P2ps (peer-to-peer simplified). Technical report, Cardiff University, 2003.

[11] Gabriel Antoniu, Mathieu Jan, and David A. Noblet. Enabling jxta for high performance grid computing. Technical report, INRIA (Institut National de Recherche en Informatique et en Automatique), 2005.

[12] Emir Halepovic and Ralph Deters. The jxta performance model and evaluation. Technical report, University of Saskatchewan, 2004.

[13] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony Rowstron. Topologyaware routing in structured peertopeer overlay networks. Technical report, Microsoft Research and Rice University and Purdue University, 2002.