XMPP Server Project

Milestone 2 Report

Alcides Fonseca
amaf@student.dei.uc.pt
2006124656

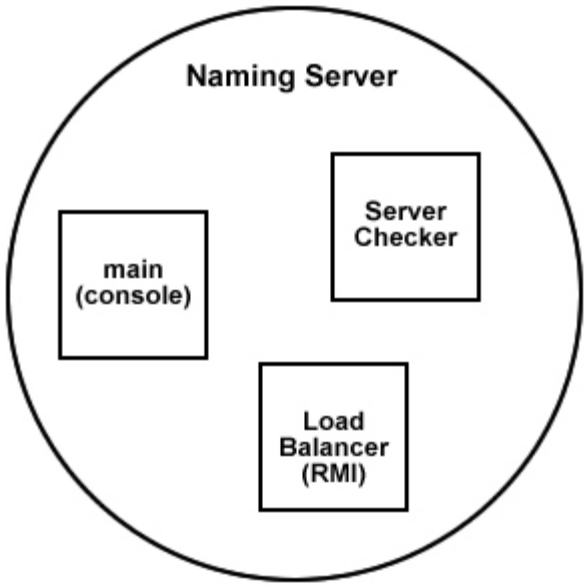
A. Introduction

In the first milestone a IM network was built just with one server. This is a very strong limitation, since in our benchmarks it couldn't handle more than 2000 clients at the same time. In the real world this is a very important issue, and the solution is to replicate the servers in a network, using XMPP's distributed nature.

The user should not know of this replication, and the state should be equal in the one-server situation. So in this new scenario, the user data (logins, contacts) is stored centrally in the Naming Server entity. This is also responsible for the Load Balancing, so we can get the maximum out of our hardware, and not overload any server.

In this solution, the only limitation to the number of users is the number of machines you can buy, and the number of accesses to the NamingServer requesting user data (Roster or authentication).

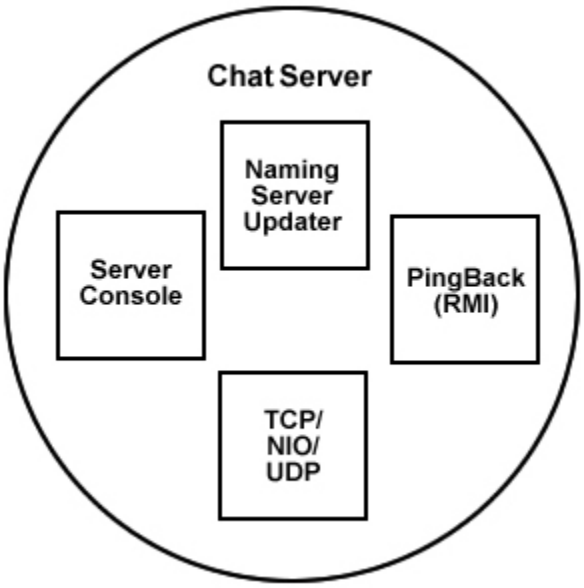
B. Internal architecture of the Naming Service



The NamingServer process has three important threads running. The first one, the one implemented in the main method, is just the console interface for the halt/restart/exit commands requested. Before entering that loop, it registers the LoadBalancer instance in the RMIRegistry. From then on, it runs on a separate thread, handled by the RMI middleware. This LoadBalancer is the responder to clients, when they are looking for servers, and also for the latter in order to communicate with the Database and update their settings.

In order to check if the servers are really online, we have another thread, that every minute (can be adjusted in the configurations) checks if a server has updated. If not, it pings that server and if it doesn't answer, that server is removed from the server list.

C. Internal architecture of the Chat Server



In the first phase of the project, we wrote three implementations of a XMPP server (and client). Now, any of the three can be used in this new model. Since the NIO version was the best one, it was the one used in the development. There is also the server console, that just accepts the requested input from the user (restart/halt/exit).

Regarding the RMI communication, the NamingServerUpdater thread just updates the Naming Server every 1 minute with the local information regarding CPU, Memory and Network usage. The PingBack is a RMI actor that just answers the ping from the Naming Server, and updates the Server Information (explained above).

D. Management of the Federation of Servers

At a given time there could be a N number os servers registered in the Naming Service. The NamingServiceUpdater thread in the Chat Server connects to the Naming Server and joins the federation. It can also withdraw from it, stopping giving updates and communicating with the other servers. All this communication is made from the Server requesting the RMI methods on the Naming Service LoadBalancer instance.

E. Monitoring of the Server

The LoadBalancer keeps an updated list of servers currently in the federation along with the statistic information for each of them. This list is continuously being pruned by the ServerChecker in the Naming Service.

All the servers that haven't been updating their information for the double of the refresh time[1] are pinged. If there is no response from the Server, it is considered dead, and removed from the list of Online Servers.

F. Load Balancing and Fail Over

As stated before, the LoadBalancer instance selects the best server for a new client connection. Since the equality of the server machines is not assured nor the fact that this is the only service being handled by any specific machine, the number of clients a server has no influence in our algorithm.

Between the CPU, Network and Memory usage, we select the bottleneck of each server in percentage. A fold left function is applied, and the one with the lowest bottleneck is the one selected.

If server A has 30% of CPU usage, 40% of occupied memory and 70% of network load, its bottleneck is the 70% of network load. Server B has 60%, 30% and 25% for the same components, so the bottleneck is the 60% of the CPU usage. So if this were the only servers in the list, Server B would be the one given to the client, since its bottleneck (60%) is lower than A's (70%).

Regarding the fail over mechanism, every time a connection from client to the server goes down, it asks the Naming Service for another Server using this algorithm.

If the Naming Service does not respond to a client or a server, this last one connects to the following Naming Service in the RMI registry list (after filtering the Naming Services there). It also unbinds the Naming Service that's down, so next connections don't find it.

G. Persistency of Data

The data storage is exactly the same as in the first phase of the project, but the Database object is now on the Naming Service, and the Chat Server just as a RemoteDatabase that acts as a proxy, using some RMI methods on the LoadBalancer object. A sqlite database was used for the development, but should be switched to a mysql or postgres database in a production environment.

The information about the servers in the federation is not stored in disk, since it should be dynamic and be frequently updated. It is however pushed to the clients when some change occurs. This kind of approach really improves network traffic instead of the usual pooling system. Pooling is acceptable with ServerData, since it's always a fixed number of values, but when it comes to the JID-Server list, it might be big at some point of the system growth, so using push here is an important improvement.

H. Exception Handling

Exception Handling is very important to detect close connects (TCP or even RMI). This allow us to keep a correct information of servers online in the Naming Server, of Clients online in the Server, and retry connections in the Client/Server RMI communication.

---

1. If a server updates every second, the maximum difference caused by server/ namingservice asynchronously is 2 seconds.

In the development of this project I felt the need to create my own exception: NoServerAvailableException, so Clients know that there is no Server available for them, and catch that exception, so it doesn't break anything in the communication, just retry later.

I. User Manual

See the README for more information. Running the jar is the best solution for running this. Compiling from source using ant is also possible, but you need to set your SCALA_HOME path in the build.xml.

The NS doesn't receive any parameter.

The Chat Server receives as the first parameter the port in which it will run.

The Client can receive a "--jabber" flat to connect to jabber.org. Otherwise it will connect to our local federation.

NS and Server both support the halt/restart/exit commands as requested. The client is the same from the first milestone.

J. Installation Manual

Installation is the same as in the first milestone. Important issues are the permissions of database, and running RMIRegistry with the path of the project classes. Running the jars are the only thing required: first one (or more) NamingServices, then one (or more) Servers, and then all the clients.

K. Test Specs

In the development of this project, every feature was tested before stepping to the next. Only the relevant tests are here enumerated:

* NS detects when RMIRegistry is down.
* Client and Server detect when no NS is available.
* Client and Server detect that NS is down, and try to reconnect using the filtered rmiregistry list.
* NS detects that a Server hasn't updated for a while
* NS pings the Server (with success, or without removing it from the list).
* Server detects broken socket with Client, and removes it from the list (and sends the new presence to anyone)
* Client detects broken socket with Server, and requests a new server from the NS (can be the same, if it's back again).
* Server changes NS without clients being aware.
* Client connects successfully with Jabber.org
* Roster operations successfully testes both with Jabber.org and local federation.