XMPP Server Project

Milestone 1 Report

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

A. Introduction

These days, Instant Messaging plays an important role in people's life, both personal and professional. Specially in business, availability and low cost of the servers is an important factor in IM servers. Being able to interoperable with other companies' server is also a preferred feature. Studying and implementing some possibilities to this problem is the aim of this project.

Three versions of the server and two of the client were developed. In the client the difference between the two was the protocol used: UDP versus TCP while the server had two TCP implementations: one using several threads for each connection, and other using only one with non-blocking sockets.

The server was implemented on the Java Virtual Machine, since one of the requirements was using JavaNIO for non-blocking sockets. The language of choice was Scala due to its hybrid Object-Oriented and Functional nature along with its natural XML native type. As for the server, same language was chosen, since much of the code was the same as the server.

B. Internal architecture of the server

B1. Common Architecture

The server is composed by two layers: The communication layer, and the processing layer. The first one differs in the three versions of the server that will be detailed below. The second one is responsible for parsing the request. The main class that is responsible for this is XMPPServerParser, that parses the XMPP stream and acts accordingly, that keeps each client's state in a Session class. There are two main objects involved in this: SessionManager and UserManager that create, change and destroy Sessions and Users, and the last one also connects to the Database (SQLite3).

It's important to highlight that each XMPPServerParser receives a OutChannel class (that differs in all three implementations) that has a write method used to reply to the user when needed.

B2. TCP Multithreading

The server's main code is an infinite loop where it accepts new connections and spawns a new thread to handle each new socket. This new thread just inserts into the temporary buffer whatever comes in the socket.

SessionManager registers the OutChannel for each client, and the associated JID to output from other connections and allow one client's thread to write in another client socket.

B3. Non-blocking TCP

This version of the server takes advantage of the Java NIO API. An instance of the Selector class is used to manage Socket-related events. At the beginning only the OP_ACCEPT event is registered at the default port (5222) and each time a user connects, a OP_READ event is registered in the selector for that particular socket. In the read event, the socket output is inserted into a buffer provided by Java NIO, decoded and then inserted into the parsing buffer associated with that socket.

Just like in TCP, there is a OutChannel class for use with Java NIO that works almost the same as the multithreaded version, but uses a intermediate buffer, according to NIO's specs.

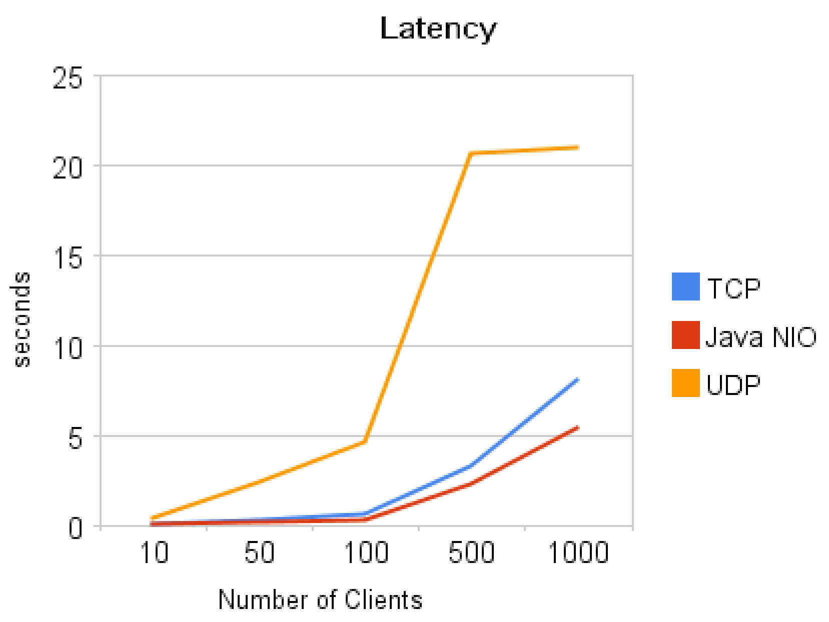
B4. UDP

The UDP architecture in the server is pretty similar to Non-blocking TCP's. There is only one main thread that saves the XMPPServerParser instance for each UDP Datagram origin. For each Datagram received, the content is added to the parsing buffer.

Also, an OutChannel class is also provided that send information to each client, through the socket used to send the requests.

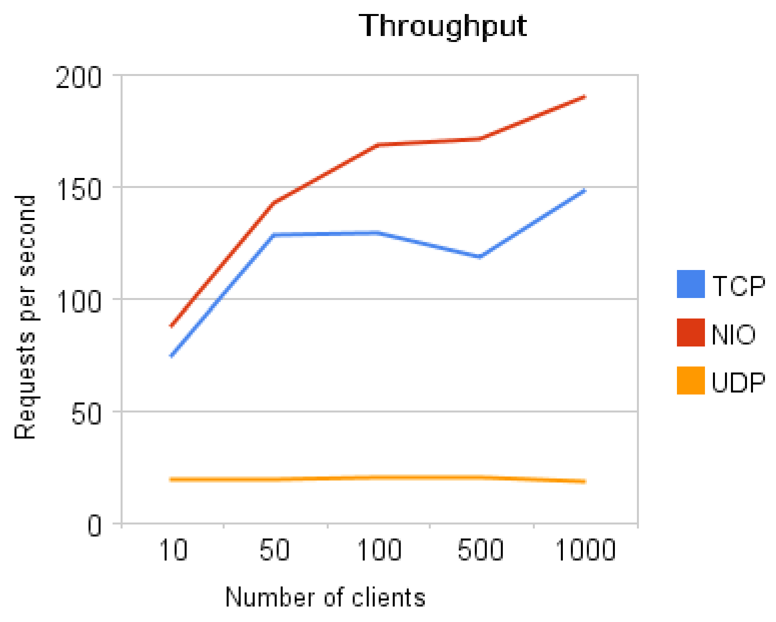
C. Benchmark Study

This benchmark study targets two measures: Throughput and Latency. The latter one was measured in the client side, counting the time from which the client requests something, and receives the response to that request.



As you can see through the graph, all of them grow when the number of clients is increased. Java NIO version grows almost at the same rate as the MultiThreaded Version, but slight lower. This lower latency may be due to the lack of CPU transition between threads that TCP has. UDP is far more slower because a new Datagram is created each time a request or a response is send, while in TCP versions the socket is already open.

As for throughput, it results from the average of requests answered by the server at the end of execution of the stress test.



The UDP version of the server is limited to 20 requests per second, whatever the number of clients is. The problem here might be the limited buffer to store UDP messages when one is being processed.

The TCP versions grow at a similar pace, just like in the latency, but the non-blocking version sure is more responsive than the blocking one. Once again the trouble of managing so many threads slows the server down. And when testing to only a few more than 1000 clients, the NIO holds a few more clients before running out of memory.

Overall the non-blocking TCP version was considered the best, since it scales better than the others, keeping a better responsiveness and lower latency.

Note: The results obtained here do not matter by its absolute value since it depends on the machine where the tests ran, its status and the java virtual machine used. However relative results between the three versions are valid and useful to select which version to deploy on a live environment.

D. Exception handling in the sockets

Since the non-blocking TCP version got the better results in the above benchmark, a few improvements were made to ensure its quality.

In the client-side both IOExceptions and EOFExceptions are being taken care of, and result on a creating a new session with the server, reconnecting and establishing a new connection. There is however a small wait of 10 seconds between each try, since when stressing the server, this value allowed a larger number of clients to successfully connect to the server.

Two more exceptions handlers could be added if more time was given to this project. The first one was to detect which client closed the session and broadcast it's status according. At this moment, a *quit* is required at the client side. The second handler would be one that detects if a connection is broken, and saves that message in a queue to later resend it when the client reconnects. Both this changes needed a rethinking of the general architecture to support server to server connections.

E. Integration with Jabber and GTalk.

This project followed the XMPP protocol, implementing the Client to Server, XMMP IM, Sessions, Authentication, Register and Presences. Although the Server to Server was not implemented, one can use our server with a regular XMPP client, Psi was used successfully, and our client with a server that supports non-encrypted PLAIN authentication[1].

F. User Manual

If using a UDP server you should use the udp_client.jar. If using a TCP server, you should use the tcp_client.jar.

Launching the client: java -jar xxx_client.jar username password.

As a default two accounts are created: teste:teste and qwerty:qwerty. More accounts can be creating using the jabber-register extension. (Psi works as an example). It is also recomended to use a GUI client like Psi since we cannot guarantee the asynchronously of XMPP in a console-based client.

When the client successfully connects to the serve, you can see your roster online. From now on several commands are available:

quit - closes the stream.

send <jid> <message> - sends a message to all resources with that jid.

add <jid> - adds a certain JID to your roster and subscribes to it's presence.

---

1. Using a PLAIN authencation through a non-encrypted channel will result in a account theft by someone sniffing the network. This is why such compatible server are so rare. Implementing TLS would be the preferred solution.

del <jid> - removes a certain JID to your roster and unsubscribes to it's presence.

set <status message> - changes your presence show message to what you define.

G. Installation and Configuration Manual

Well, to install the server you just have to copy the selected jar (tcp_mt_server.jar, tcp_nio_server.jar or udp_server.jar) into somewhere along with the folder db (with dev.db inside). Just run it afterwards.

No need for configuration, but in development mode, you might want to take a look to Config.scala.

H. Description of the Tests Made to the Application

For each featured developed in the server , it was tested with the Psi client in different database states. This would include add and subscribing, removing and unsubscribing, changing presence and sending messages.

The client was implemented to mimic Psi's XML stanzas to ensure XMPP compliance. Regular tests were ran to ensure each feature was correct, again altering between different database states. Interoperability between command-line client and Psi was also tested.

Finally to perform the benchmark a few stress tests were applied. Running multiple clients from the same origin, in the same account and in different accounts, and exchanging messages between them was also tested.