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