XMPP Server Project

Milestone 3 Report

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

A. Introduction

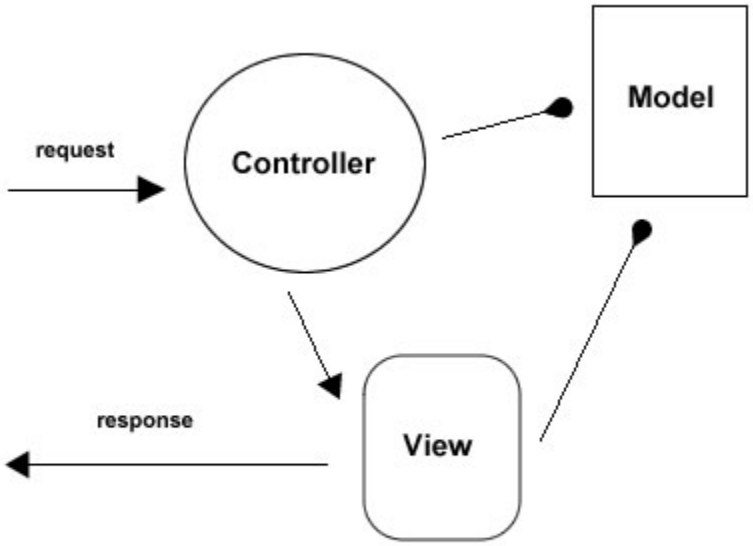
In the previous phases, we developed a chat server and then made it work in a federation of servers in order to scale. But computer engineering is not only about performance and numbers, but also about interacting with the user. A command line interface is okay for testing and debugging, but for the end-user is not usable as it should.

In this milestone, the solution proposed is a rich internet application backed up by one (or more) web servers that handle the HTTP to XMPP communication. Web applications have the advantage of not requiring any deploying, just sending the link. This is also an advantage with upgrades that are instantly done by upgrading the server. There are however some restraints with this solution: the webserver is acting as a client also, and it's something traditional desktop clients wouldn't require. A solution in between would be making the webserver also a XMPP server communicating with all the others through RMI (or even S2S).

B. Web-based architecture

B1. MTV pattern

When writing software, the domain separation is an important issue that can't be forgotten. There are a few design patterns that can be followed. MVC (model view controller) is an example of a widespread one where the business logic remains in the controller, the interface details in the view, and the data in the models.



Although this works fine in a desktop environment, when making complex web-based applications, this might not be the best choice. Although popular frameworks are using this pattern (Rails, Symfony, Mason), there are always some changes to the pattern that happen.

In this project, I followed Django's MTV approach[1], in which the views do the business logic, and may call templates that handle the format produced (being HTML, XML, JSON, YAML, ATOM, etc...) and also uses models for accessing data. Controllers are used only for routing the requests for the right view.

B2. RESTful API

A web-chat is a dynamic application, that requires AJAX to be usable. When using this kind of approach, where the HTML+CSS+JS combo acts as a standalone client, a RESTful approach is preferred.

Our server exposes a REST API, where some URLs accept HTTP's four verbs: GET, PUT, POST and DELETE. The first one to retrieve information, the second to create, POST to change it and the last one to delete some information.[2]

An advantage of this approach is that the traffic to our servlet is only real information, and the UI can be provided by a faster webserver like Lighttp or NGinx.

Today's browsers only support GET and POST verbs, so in addiction to the native doPut and doDelete methods, Rails-like PUT and DELETE over POST is also supported.

---

1. For more details on this approach visit http://jeffcroft.com/blog/2007/jan/11/django-and-mtv/
2. A more detailed paper on this pattern is available in http://ajaxpatterns.org/RESTful_Service.

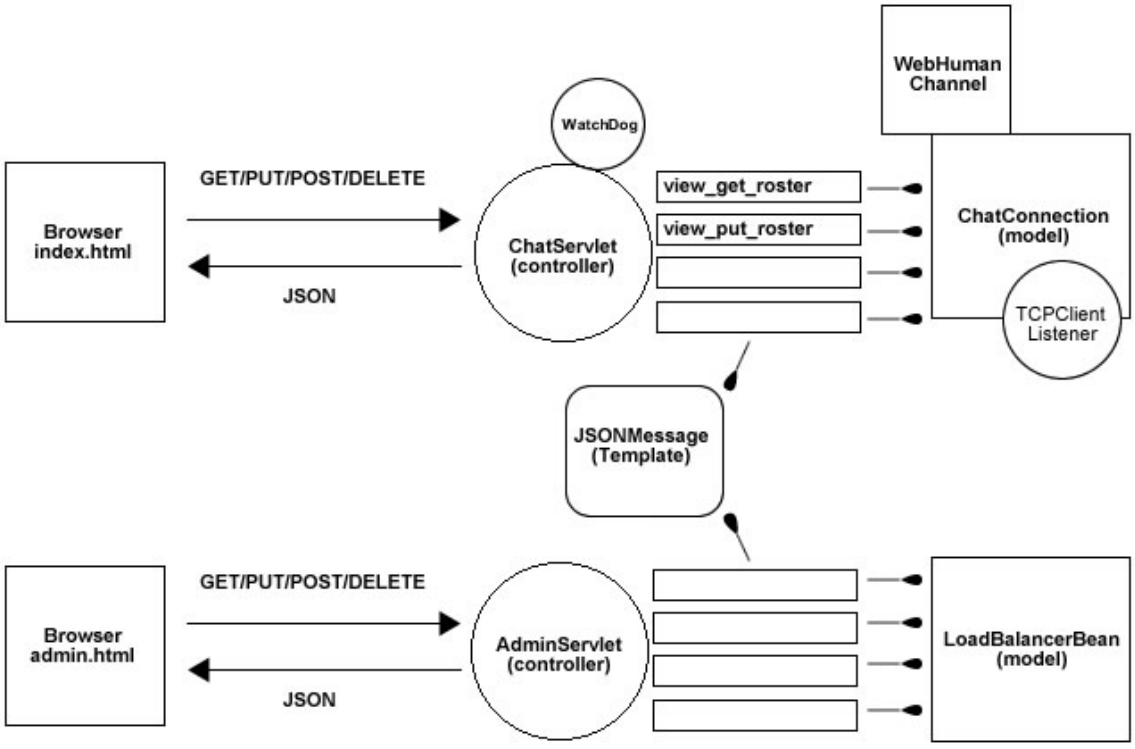
In this project, JSON was chosen for the data transfer over XML because it's more lightweight, and it's faster to parse in JavaScript.

B3. Comet

IM communication requires low latency communication, and in those cases PUSH is preferred over POOLing. The XMPP is the standard used in those situations, instead of HTTP. However in these days, HTTP is also adopting PUSH, and some solutions are being developed. Until HTML5 browsers become mainstream, along with web-sockets[3], we can use Comet for that purpose. There are a few methods to apply Comet[4] but right now the only standard method is using Cometd[5] from the Dojo Foundation.

I spent several time trying to get Cometd working with Tomcat, but right now there are only a few tests available and documentation is lacking. Although pooling is more expensive for the server, that's the approach used due to the current limitations of the platform required.

B4. Application Architecture



---

3. The draft spec is available at http://www.whatwg.org/specs/web-apps/current-work/multipage/comms.html#network
4. http://en.wikipedia.org/wiki/Comet_(programming) has a detailed list on them.
5. http://cometdproject.dojotoolkit.org/

This is the general architecture of the webserver. The user interface it totally independent from the logic, that is handled by the server. The UI is send to the client in the first initial request (being index or admin.html) along with all the CSS and JS dependencies.

There are two different controllers there (one servlet for each) since there are two different applications that handle different things. Each servlet has the views required for each url and for each method applicable. In the code those are functions, since it's the most correct representation, just like in Django.

Views can access models to change something. In the admin application, a vanilla bean was used to encapsulate the LoadBalancer instance running in the Naming Service. As for the Chat application, a ChatConnection represents the model but in order to be compatible with the rest of the XMPP architecture it is not truly a Java Bean.

Any view can use the shared JSONMessage, a small template for a regular message, but writing the JSON in an OOP approach is also an alternative for the views. Since no HTML is generated in the server, there was no point in using JSP.

C. Integration between the Web Server and the Chat server

As mentioned before, the communication with the Chat server is not done in the servlet, but rather in a special class created for that purpose. A ChatConnection is created per session and exposes an API to the views.

Since HTTP is stateless, we are associating the ChatConnection instance with the SessionID provided by the Servlet API. All of them are stored in a HashTable in the servlet instance in which a WatchDog thread is checking them for recent activity. If a connection has no web-based requests for 10 seconds, the WatchDog closes it. Since the user pools the Connection every 3 seconds, this actually works fine for detecting dropped connections.

This along with the need to store the received information from the server in a WebHumanChannel is the reason the ChatConnection is not a Java Bean. The WebHumanChannel is given to XMPPClientParser to store the output of the connection. The ChatConnection then fetches that information on demand.

In everything else, the ChatConnection is pretty similar to the TCPClient. Each one has a ListenerThread hence the need for the watchdog to close them.

D. Integration between the Web Server and the Naming Service

The LoadBalancerBean exists per application (that is, servlet instance). When created, it connects to the RMI instance of the LoadBalancer running in the Naming Service, using the Connector from phase 2. It then proxies every request to the original LoadBalancer.

A few more methods were added to manage accounts (create, delete and change password).

E. User's Manual

To use the Chat application, just enter the address of your server in your browser (Ex. http://127.0.0.1:8080/). Then enter you user name and password, and if you want, chose to connect to Jabber.org instead of your federation.

Inside, you can change your status in the drop-down, click a JID to chat with him, add and remove contacts.

To use the Admin application, access the admin.html in your browser (Ex. http://127.0.0.1:8080/admin.html) and switch between the tabs to display information.

In the server's tab, you can click the Number of Clients to show the list of online JIDs in each one. In the accounts tab, you can create an account, change the password of an existing one or just delete an account.

E. Installation Manual

Requirements:

- tomcat 6 installed (http://tomcat.apache.org/)
- scala installed (http://www.scala-lang.org/)

First, change the tomcat and scala home directories in the ANT's build.xml in the root of the project.

Then run ant compile to generate all the code required.

In the build folder, run the rmiregistry.
Back in the project root, run ant ns to start the Naming Service.
In the build folder run scala BootServer <port> to start as many servers as you like.
You can also run scala BootClient to run a command-line client, or scala BootClient --jabber to connect to jabber.org.

Next, having your tomcat running, create a softlink from the ROOT folder to the public one inside this project. Add the build path to the CLASSPATH of your tomcat and restart it.

Everything should run fine here, although it depends on the configuration of your tomcat.

G. Tests performed to the Application

- Chat Application
    - JSON messages are sent correct.
    - Connects to the Naming Service
    - Connects to the TCP Server
    - Connects to the Jabber Server
    - Gets the right roster
    - Adds/removes a contact and changes the Database
    - Sends presence when status is changed.
    - Sends/receives messages to/from a JID using Web and/or CLI
    - Receives requests and accepts or not.
    - Open a new window in a new chat.
    - Client is closed when browser is closed after a while
- Admin Application
    - Server and Client information is the same as in the NS.
    - Create/Delete/ChangePassword reflects in the database.