Chat Server

Paradigms of Distributed Systems, MEI/UM
January 10, 2015

Ana Paula Carvalho[1] and Vtor Enes Duarte[2]

[1] Minho University, Portugal
pg25335@alunos.uminho.pt
[2] Minho University, Portugal
pg27754@alunos.uminho.pt

Abstract. This project aimed at building a chat server (server and client).
The server was built using relevant paradigms for the several components, namely actors, message-oriented and resource-oriented. This made the project easily scalable.
The client uses a text protocol to interact with the other users. There is also administrators who run the available chat rooms. It is also possible to observe the system by subscribing relevant chosen events.
All the required features were implemented and some other as database support, private messages history and GUI client were added.

1 Introduction

This project aims at building a **chat service (server and client)**. The server are the multiple components of the system that interact among each other while being hidden from the user. The client complies all the ways to access the services available by the server. In this case, the client is usually a chat user that comunicates with the server via a protocol (text-based).

Project Goals. This project aims at building a distributed chat service with the following features:

- **User registration**, given name and password; registration removal; a user should be authenticated to use the service;
- **Choice of room** (from existing ones), to which text messages will be sent;
- Sending of **private messages** to other connected users;
- Have a simple **text-based protocol** to allow simple chat clients, being usable by telnet;
- Have a **REST API** for management and description: e.g., room creation/removal, list of rooms, list of users in room;
- Have a **notification API** to allow subscribing to relevant events: room creation/removal, user joining/leaving room;

Report structure. Bearing this in mind, this report will fistly address the theory behind this work in section 2; It will then in 3 explain the system's architecture and implementation choices; In section 4 there will be some use cases and a description of some problems found; And, in section 5 a final appreciation of the work will be made, together with some suggestions for improvement.

2 Paradigms and Tools

2.1 Message oriented programming

A message based system is composed by elements glued by messaging patterns. A very important message pattern is the publisher-subscriber. The publisher produces messages which behave like events and subscribers register interest in receiving certain events. So this can be interpreted like a event notification system.

ZeroMQ - 0MQ. 0MQ acts like an open-source concurrency framework that gives access to sockets that carry atomic messages across various transports. It allows to connect sockets N-to-N with patterns like fan-out, pub-sub, task distribution, and request-reply. Its asynchronous I/O model gives you scalable multicore applications, built as asynchronous message-processing tasks. In this project, 0MQ will be used to reproduce an event notification system for relevant events on the chat service using the publisher-subscriber pattern.

2.2 Actor model

Actors are essentially well encapsulated active objects, which can only communicate by sending one another immutable messages asynchronously. Whatever state an actor holds internally, it cannot be accessed from outside the actor except by sending a message to the actor and receiving its reply. An actor can choose the behaviour for processing the message received, including no reply. Considering the special case of a chat, an actor seems like a very intuitive concept to implement since the core of a chat service is sending and receiving messages. An actor is an abstract lightweight entity so it can be created in large numbers.

Quasar. The way this project implements actors is via Quasar. Quasar is a Java library that provides high-performance lightweight threads - called fibers -, Go-like channels, Erlang-like actors, and other asynchronous programming tools [3]. There can be millions of fibers in an application. All actors extend the Actor class. A simple way to start an actor is by calling *actor.spawn()* which assigns the actor to a newly created fiber and starts it. **Spawning an actor is a very cheap operation** in both computation and memory.

2.3 Representational State Transfer - REST

REST is an architectural style used as a set of guidelines for creating web services. REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems [1]. Operations in REST manipulate resources through representations which in turn are manipulated through standardized media-types. Server state is only resource related (not client session related) leading to greater scalability and fault-tolerance.

Dropwizard. Dropwizard straddles the line between being a library and a framework. It uses:

- Jetty HTTP library to embed an incredibly tuned HTTP server
- Jersey for RESTful web applications;

3 Implementation

The project was developed using Git as a version control system. All code and issues reported can be found in GitHub. The project is divided in four Main Projects:

- **ChatServer** - the core of the project: actors, notification API, REST API and database;
- **ChatClient** - the GUI Client and Admin;
- **NotificationClient** - ZeroMQ based project that allows the client to subscribe the most relevant events in the Chat;
- **Common** - utilities and common classes to projects.

3.1 Chat Server

Most of this project is actor-based. When the server starts, several actors are spawned: two Acceptors, a Room Manager, a Main Room, a Notification Manager and an overall Manager.

Acceptors. Each Acceptor is listening on a different port, one for clients that use our text protocol, the other one for the GUI client. This happens mainly because some of the best features are only available in the latter. Every time any of these actors accepts a new connection, an actor **User** is spawned.

User. Each User has an actor named **LineReader** dedicated to read from his socket. When this actor sends a message (lines it reads from the socket) to the user, it checks if the line starts with the character **:**. If so, it checks whether it is a valid command and acts upon that. If not, and when the user is connected to a room, it sends the line to that room.

Every command sends a message to other actors, and most of them need an immediate answer. To achieve this, a special entity, **Pigeon**, was created. This Pigeon is an important tool in our tool set. It carries a message to any actor and always comes back with an answer.

Manager. To better understand this, another essential actor, the **Manager**, is needed. This actor is in charge of user registration, authentication and removal and private messaging. When the user tries to create an account, this is accomplished with a Pigeon that carries a message to the Manager and retrieves the reply to the User, allowing or preventing the action. After the log in, also achieved with this technique, the user is automatically connected to the Main Room. This room exists by default and it cannot be deleted by any administrator. At this point, the user might try to change to another room. We do this creating another Pigeon but the recipient will be the **Room Manager**.

Room Manager. The Room Manager, besides controlling room changes, is also responsible for room creation and removal. Such requests are made using the **REST API**, which will be describe further in the document. When the Room Manager receives a message soliciting a room change, if the rooms exists, it contacts the **Room** actor, notifying that a user will connect soon. The Room actor counts how many users will connect soon. This will be very important for the REST API. The Room Manager and each Room actor have another important task: send a message to the **Notification Manager** every time a user enters or leaves a room and every time a room is created or removed.

Notification Manager. Every time this actor receives messages, it fowards them to possible connected Notification Clients that subscribed the event that message represents. This is accomplished using 0MQ pub-sub pattern.

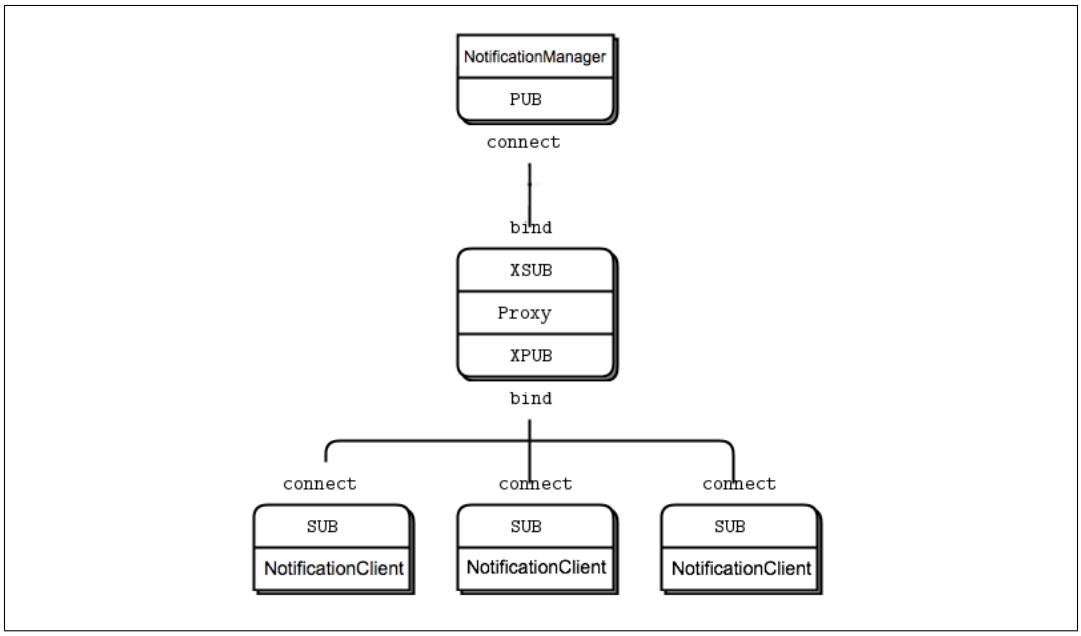

Fig. 1: Pub sub

As we can see in Figure 1, we only have one publisher but we are prepared to add new publishers in the future very easily. The subscribers use our **Notification API** which we'll discuss further.

REST API. This project also has a layer that consists in RESTful API using JSON as a serialization format. For detailed information, check the appendix A. The choice of JSON instead of the standard XML results in an increase of performance in terms of serialization and deserialization because JSON is a much more lightweight format. For serialization and deserialization purposes, Gson was mainly used.

When implementing this, we faced a problem. We needed to guarantee that some API calls were only made by admins. For that, this calls request an Header Parameter named **Volatile-ChatServer-Auth**. This acts as an authentication token. To create this token, we encode a JSON with two properties, username and password, using Base64 encode algorithm. If the username belongs to an admin, and the password is correct, it's a valid token. This mechanism provides no confidentiality protection for the transmitted credentials. They are merely encoded with Base64, but not encrypted or hashed in any way. Therefore, is typically used over HTTPS, which we're not doing. This difficults the use of this calls using a simple client as curl, so we recommend the use of our Chat Client.

There is a situation we took special attention when implementing this API. When a user asks to change room, he will be given the ActorRef to that room. Meanwhile, using this

API, the admin might try to delete the same room. The order to delete can arrive before the message from the user notifying his entering, and if so, the user has an ActorRef to a room that will be deleted. This problem was solved with the counter mentioned before: how many users will enter soon. The attempt to delete the room will only succeed if the room has no users and this counter is zero.

OrientDB We're using OrientBD [2] as database. OrientDB is an Open Source NoSQL DBMS with the features of both Document and Graph DBMSs. Our system use it as a Graph Database. Although we do not have a persistence layer, we believe it's a good feature because we can recover the server state if for some reason it crashes.

Capsule To deploy our service as a standalone application we use Capsule [4]. This allows to package our application into a single JAR and deliver it to anyone who wants to use it. This happens to the Chat Server, Chat Client and Notification Client projects.

3.2 Chat Client

The Chat Client was built for end users. A simple client (e.g. telnet) enjoys almost the same chat features as the GUI client using our text protocol 1. Other features, as listing rooms, choosing a room and listing users from a room are only accessible using the GUI client. These are calls to our REST API. To also achieve this, the simple client has to use a command line tool like curl or another http client.
Another feature is the Inbox. Each client can send private messages to other chat users (even if they are offline) and load their inbox history.

Text-based protocol The text-based protocol allows that even simple clients as **telnet** and **nc** can enjoy our chat. All protocol components - commands - start with **:**. The protocol is as simple as possible so that it would not become too unpleasant for the simple clients.

Command	**Description**
:h/:help	Lists all available commands
:create username password	Creates a new user
:remove username passsword	Removes an existing user
:login username password	Logs an existing user
:logout	Logs the user out of the service
:cr/:changeroom newRoom	Changes the current room to newRoom
:private username message	Sends a private message to a user by username
:inbox	Displays the private messages received

Table 1. Text protocol for a simple client.

Admin. The Admin is a regular chat client with special permissions: it can create new rooms, delete existing ones, promote users to admin and remove their privileges. In the full version, the Admin can easily achieve this because the work of encoding his credentials

is done by the application. When using curl he has to use some Base64 encoder for that, which is very distasteful. If the username and password are admin, after encoding the JSON {"username":"admin", "password":"admin"} the curl command to create a room is

```
curl -X PUT -H
"Volatile-ChatServer-Auth : eyJ1c2VybmFtZSI6ImFkbWluIiwgInBhc3N3b3JkIjoiYWRtaW4ifQ=="
localhost:8080/room/ROOM
```

3.3 Notification Client

This client uses our Notification API to subscribe relevant events. This API is very similar to REST API endpoints.

:sub rooms	subscribes rooms creation and removal
:unsub rooms	unsubscribes
:sub room/name	subscribes entries and exists in the room with that name
:unsub room/name	unsubscribes

Table 2. Notification API

If in the future our service grows, and we have more publishers notifying other events, this client demands no change.

3.4 Common

In this project we have the common classes of the whole project. One particular class, Saying, has the replies to all user actions. Each Saying method returns a String, which are used both in Chat Server and Chat Client. It also has representations of important entities that serve as a way to exchange information between the several components.

4 Results

The first thing a user sees after connection to our system is the Authentication page.

The user has the option to log in, but only after creating an account. He can also remove an existing account. The system catches all this misuse alternatives and helps the user to the right path.

After the user is logged in, it is automatically in a chat room, the Main room, which is the default room of the service. In the simple text-based client, the user can type *:help* to see the available commands. The GUI client has a panel that emulates just that: a text-box to write chat messages; a text area where the chat messages show up; a list of existing rooms or online users in a room; a button to go to his inbox; and, if it is an admin user, a button to go to admin options.

If the users clicks in the inbox, it can view the history of previous conversations with other users (:inbox command for the simple client lists only messages received) or send a private message to a user it never did before (:private user message for the simple client).

If an admin user clicks on the admin button option, it finds a panel where he can add or delete a room and give or take admin privileges to other users.

In summary, all predictable error behaviour by the user is supported and information is displayed. During the implementation, other possible bugs were tested and, the ones found, were corrected.

5 Conclusions and Future Work

This project was successful in emulating a Chat service comprising both a server and a client.

The project also has many features that make it scalable and easily maintainable. One issue that prevented the project to be more manageable was the way messages are in Quasar. They're not as flexible as, e.g., Erlang messages, in which we're not limited to the number of elements of the message neither their type.

Despite the fact that no major bugs were found, there is still room for improvement. Some tests should be developed to ensure that the system is running properly, to verify database performance and the communication between the GUI client and the server never fails. We think that a major improvement would be integrate the Notification Client in the Chat Client.

In conclusion, all the goal were meet and some further improvements were implemented. Therefore, we consider this a successful demonstration of a Chat Server.

References

1. Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
2. LTD Orient Technologies. OrientDB.
3. Parallel Universe. Quasar documentation.
4. Parallel Universe. Capsule.

A API REST

Endpoint	**/rooms**	
HTPP method	**GET**	
Response Codes	200	OK
Response	{"rooms":["Main, ShareLatex, Erlang"]}	

Table 3. List rooms

Endpoint	**/room/{name}**	
Parameters	name - name of the room	
HTPP method	**GET**	
Response Codes	200	OK
	404	If the room does not exist
Response	{"name":"Main","users":["ana", "vitor"]}	

Table 4. List users in room

Endpoint	**/room/{name}**	
Parameters	name - name of the room	
Headers	**Volatile-ChatServer-Auth** - authentication	
HTPP method	**PUT**	
Response Codes	201	Created
	401	Unauthorized
	409	If the room already exits

Table 5. Create room

Endpoint	**/room/{name}**	
Parameters	name - name of the room	
Headers	**Volatile-ChatServer-Auth** - authentication	
HTPP method	**DELETE**	
Response Codes	200	OK
	401	Unauthorized
	404	If the room does not exist
	412	If the room still has users

Table 6. Delete room

Endpoint	**/admin/{username}**	
Parameters	username - username of the user	
Headers	**Volatile-ChatServer-Auth** - authentication	
HTPP method	**PUT**	
Response Codes	201	Created
	401	Unauthorized height
	404	If the user does not exist

Table 7. Make admin

Endpoint	**/admin/{username}**	
Parameters	username - username of the user	
Headers	**Volatile-ChatServer-Auth** - authentication	
HTPP method	**DELETE**	
Response Codes	200	OK
	401	Unauthorized
	404	If the admin does not exists

Table 8. Detailed REST API.

B OrientDB

Vertices

- User: username, password, registrationDate, loggedIn, active;
- Room: name, creationDate, active;
- Message: from, to, text, date.

Edges

- Messages: from Room to Message: A room has all messages sent in that room linked with an edge; We're not linking the message to the user who sent it because there's no use for that now.
- PrivateMessages: from User who sends to Message and from Message to User who receives: User's incomings edges are messages received; User's outgoing edges are messages sent.

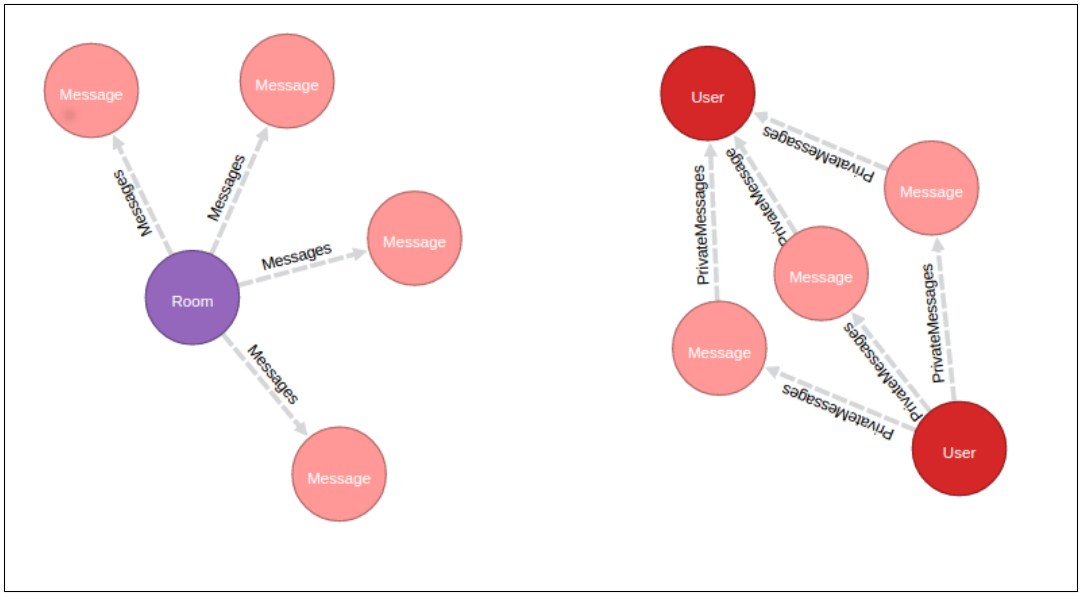

Fig. 2: OrientDB graph example.