Seminar Report: Chatty

João Neto and Igor Zavalyshyn

September 30, 2013

1. Introduction

This seminar allows us to implement and test a simple type of the distributed system with one or several servers and connected clients. In our case it's a chat service. This example clearly shows us distributed systems in action and gives us an opportunity to think about the problems we may face (e.g. what happens if the number of clients increases or the server fails). Moreover, seminar lets us learn the basics of Erlang language.

2. Work Done

Source code is in attachment. Code has been supplemented with the comments to understand each line and it's action.

3. Experiments

Part 1 - Single server and several clients

Figure 1 shows the results of testing the server with several connected clients. As you can see, messages from one client are delivered to the rest of the clients at once. Name of the message sender ("From" value) is displayed before the message text. Server also notifies all the clients when new client is connected ("JOIN" event) or disconnected ("EXIT" event).

In this scenario the whole service relies on a single server. If it crashes nobody will be able to use the chat.

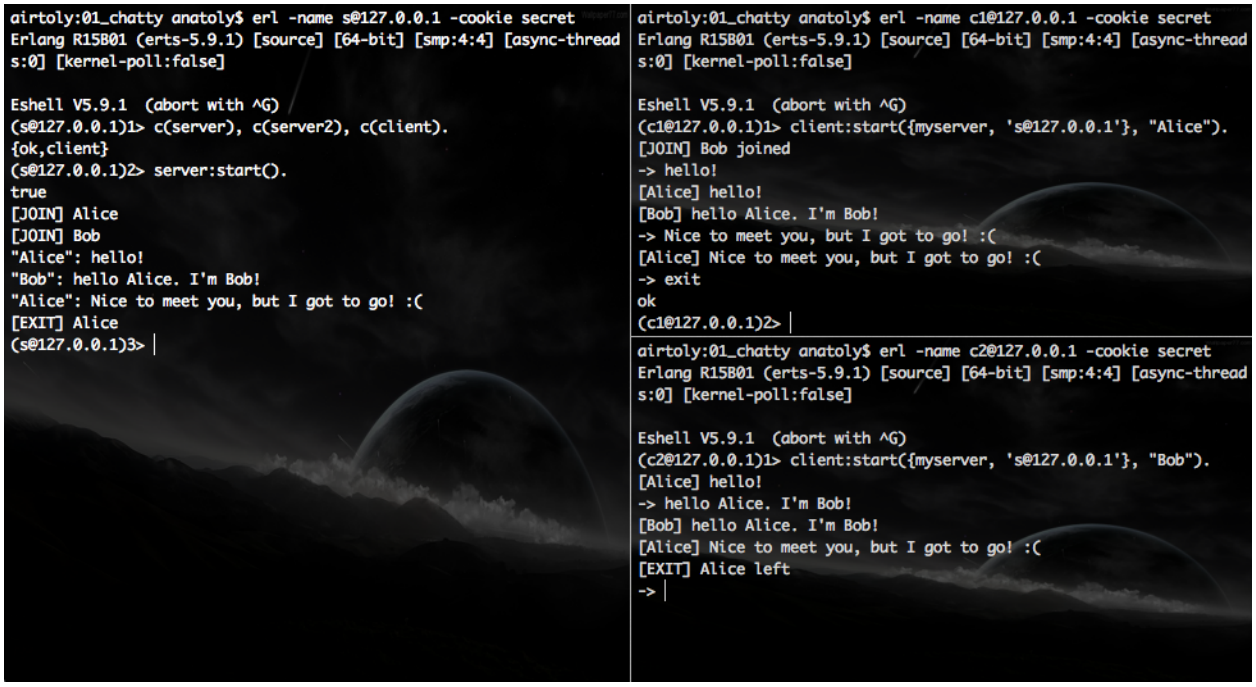

Fig. 1 - First implementation of the chatty service, with a client-server architecture

Part 2 - Several servers and clients

Figure 2 shows the results of testing the service with several servers and clients connected to each of them. As we can see messages are delivered correctly, even between the clients that are connected to different servers.

When new server connects the rest of the servers are notified about that. Name of this server is added to the list of replicated servers (‘Servers’ variable). Each server has this list and uses it to send messages to other servers.

If we try to simulate the server crash (in our case we just closed the terminal window of one particular server instance), we see that service keeps on working. Clients connected to other servers will be able to chat with each other, but those who were connected to the “dead” server will be left apart. This is because all the servers keep track of every server connected to the chat network.

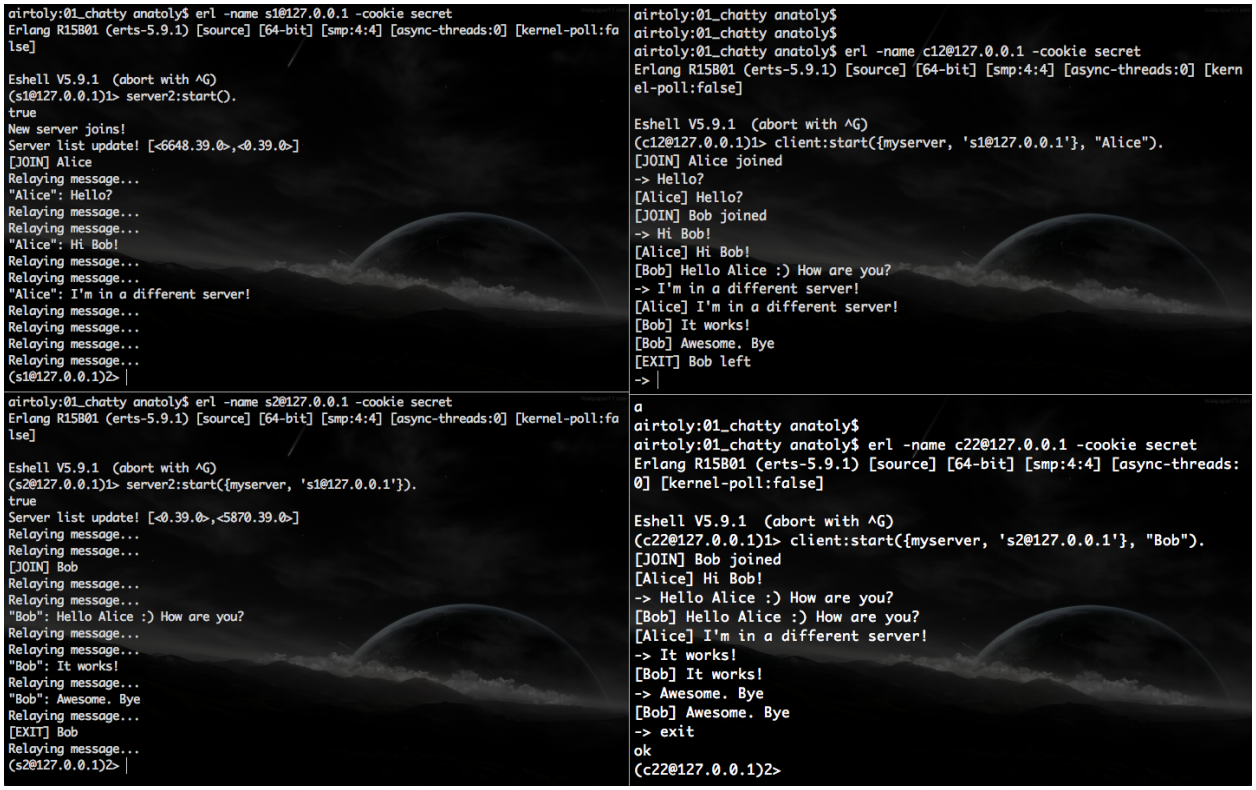

Fig. 2 - Multiple clients in multiple servers

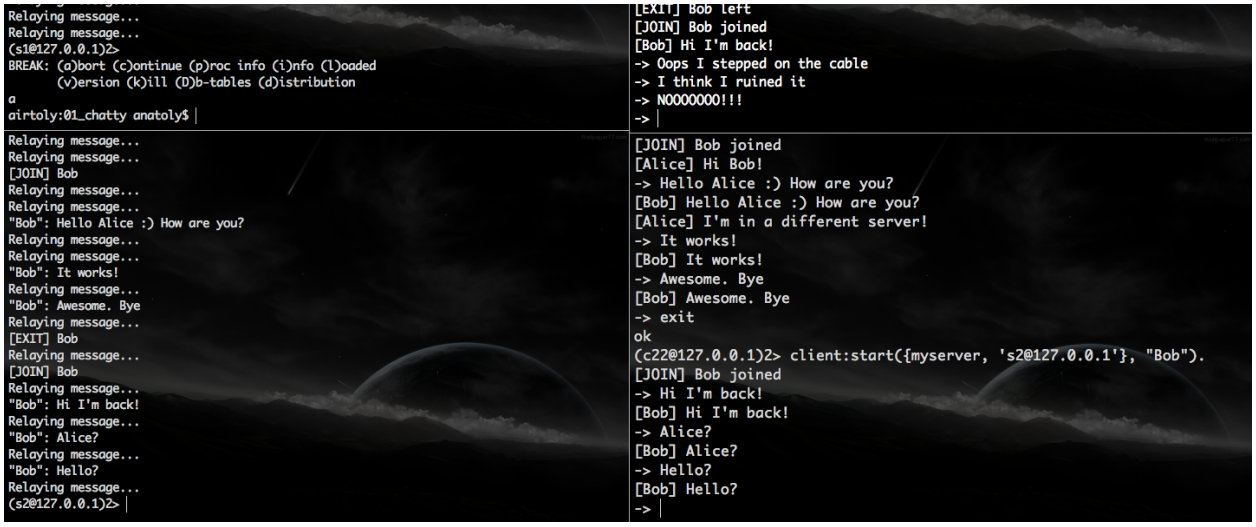

Fig. 3 - Service behaviour when one of the servers crashes

4. Open Questions

Part 1 - Single server and several clients

i) Does this solution scale when the number of users increase?
We only have one server that handles the requests from all the clients. If the number of clients and as a results the total number of requests to the server increases, it will require additional CPU power. In case of working with hundreds of clients it's better to use several distributed servers connected with each other. This will also make the whole service more stable and scalable.

ii) What happens if the server fails?
The whole service will stop working. All the clients will be unable to chat.

iii) Are the messages from a single client to any other client guaranteed to be delivered in the order they were issued? (hint: search for the 'order of message reception' in Erlang FAQ)
Yes, if we are talking about the messages sent from one client to another, messages will be delivered in the order they were sent.

iv) What about the messages from different clients (hint: think on several clients sending messages concurrently to another one)?
In this case we can not guarantee that messages will be delivered in the same order as they were sent. Different factors (e.g. latency, faulty channels) would lead to the difference in time of delivery.

v) What happens if a user joins/leaves the chat while the server is broadcasting a message?
Server will process the event of a client joining or leaving the chat after the current operation is completed (in this case it's broadcasting the message). In other words, JOIN/LEAVE event will be placed on the server's process queue.

Part 2 - Several servers and clients

i) What happens if a server fails?
If one of the server fails only those users connected to the failing server will lose connectivity but the rest will remain connected to the service.

ii) What happens if there are concurrent requests from servers to join or leave the system?
All the requests will be processed sequentially, by order of arrival.

iii) What are the advantages and disadvantages of this implementation regarding the previous one?

The main advantage of the second scenario is that chat service will keep on working even if one of the servers fails (there is potentially no single point of failure). The disadvantage in this case is that clients of the “dead” server are not able to continue using the service and are unreachable for the rest of the clients. The best solution would be reconnecting to other server if a client detects a failure.

Moreover, this new implementation does not manage the resources decently. If a server dies, it’s identifier will remain in the servers list. If N servers die, this results in unnecessary N messages being relayed for every message received.

There are also some scalability issues. Even though each server only ‘talks’ with a subset of all the clients, there is also the overhead of ‘talking’ to all the other servers.

5. Personal Opinion

Interesting and funny way to understand Erlang’s syntax and the basics of distributed systems. It shows how powerful Erlang is, and how easy is to build something that would be a lot more complex in other languages (e.g. C) or more verbose (e.g. Java). We believe it should be included in the next year - it is an interesting subject for starting in Erlang.