Erlang UvA Summer School

Arjan Scherpenisse

July 5, 2013

1 A simple chat system

Use `spawn/1` and `register/2` to start a named server process which maintains a list of connected clients in its state. Every message that the server received, needs to be sent to every other client. Using `register/2` you can give the server process a name, so each client connect to it. The client prints every received message.

You should create two modules, one for the client and one for the server; e.g. `chat_client` and `chat_server`.

`chat_server` should export a function `start/0` which spawns the server process and registers it.

`chat_client` should export functions `connect/0` to connect to the server, and `message/1` to send a message to all connected clients.

Launch different terminals with different Erlang shells to test your code: one terminal for the server, and one for each client. Use `"erl -sname <nodename>"` to start the erlang shells so the erlang processes are connected. Use the syntax {`Processname, NodeName`} `!  Message` to send a message between nodes. For instance: {`server, server@localhost`} `!  "foo"` to send `"foo"` to the registered process called `server` in the `"erl -sname server"` terminal.

You can use `net_adm:ping(nodename@localhost)` to test if your terminal is connected to the other nodes.

2 Possible extensions

1. Before sending, echo the message locally using `io:format/2`. Then, send the message to every other client **but** the sending client, to disable local echo.

2. Require a nickname which clients have to enter when registering/connecting. Send this nickname along while broadcasting client's messages.

3. In the server, maintain a history of the last N messages (with timestamps!) that have been sent, and send these upon establishing connection.

4. Implement the concept of *rooms*: clients can join specific rooms and only receive messages sent to those rooms. Find a nice way to maintain the list of rooms and clients on the server.

5. Use the rebar build system to build your app instead of calling `c(filename).`, and use the OTP guidelines to make a proper `gen_server` out of the chat server.

6. For more chat inspiration: look at the features of the IRC protocol, or, even better, the XMPP (Jabber) MUC (Multi User Chat) specification to implement more features in your chat system.

7. Make the server/client communicate with eachother over TCP/IP like a proper internet server; use the `gen_tcp` OTP module. This is a major step: you need to implement proper protocol parsing, no more passing around Erlang terms between client/server...

8. Implement a web interface for your module, implementing realtime web chat using websockets or longpolling. Woah! This sounds more complicated than it is. Ask Arjan about Zotonic, and/or check out the zchat module from Google code.

9. For extra brownie points, document your code, put it in a DVCS, publish it on Github, tweet about it, et cetera :-)