Erlang Chat Server

Simon Fowler

December 14, 2015

1 Introduction

In this report, we describe the design and implementation of a session-typed chat server, `mse-chat`, using the `monitored-session-erlang` [1] session types framework for the Erlang programming language.

More of an introduction to the `monitored-session-erlang` framework is contained within the DNS Server case study.

2 Description

`mse-chat` is a chat server built on the `monitored-session-erlang` framework. The structure of the application follows standard Erlang/OTP practices, and is arranged in a supervision tree. The supervision tree is shown in Figure 2.

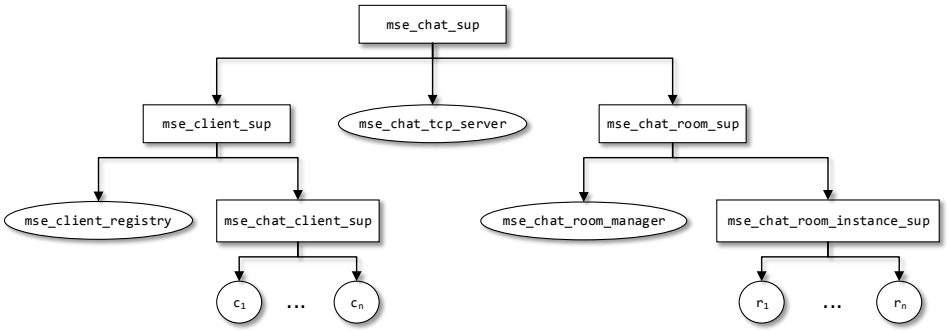


Figure 1: Supervision Tree for `mse-chat`

As is typical, we have a root supervisor, `mse-chat-sup`. We have two subsystems: a client subsystem, consisting of a client registry and a client supervisor: the client supervisor spawns child processes to handle communication with the user, but does not restart them should they fail. The `mse_chat_tcp_server` listens on a socket and accepts new clients, spawning a new `mse_chat_client` when a client connects.

Clients can create and join chat rooms. Chat rooms are represented by `mse_chat_room_instance` actors and recorded by the `mse_chat_room_manager` actor.

Excluding supervisors, the different types of actors are as follows:

`mse_chat_client` (written as $c_1 \ldots c_n$)
: An actor spawned for each chat client. Contains the socket used to communicate with the client.

`mse_chat_tcp_server`
: Listens on a socket, and spawns a new `mse_chat_client` actor when a new connection is made.

`mse_chat_room_instance` (written as $r_1 \ldots r_n$)
: Represents a chat room. Contains details about the current clients joined to the chat room, and can register / deregister clients, and broadcast chat messages.

mse_chat_room_manager
: A registry for chat rooms. Stores a mapping from chat room name to process ID. Can create, lookup, and list chat rooms.

The protocol proceeds as follows:

- A client connects to the server.
- An `mse_chat_client` actor is spawned to handle incoming requests from the new client.
- A client can either create or join a room.
  - If a client chooses to create a room, it sends a request to `mse_chat_room_manager`, which responds by either notifying the client that the room has been created successfully, or that the room already exists.
  - If a client chooses to join a room, it sends a request to `mse_chat_room_manager` to ascertain whether the room exists. If so, then the client is registered with the room. Once registered with the room, any chat messages sent should be delivered to all other clients registered with the room.
- The client can leave the session at any time. When it leaves, it should be deregistered from any rooms to which it is registered.

The chat server example differs from the DNS server as sessions are longer: whereas a DNS handling session existed only while the request was being fulfilled, a chat server persists for as long as the client remains connected to the server. Similarly to the DNS server example, not all roles are inhabited at the start of the protocol, as a client joins the chat room only after specifying the room name.

Upon receiving the room PID, the client creates a new subsession for interactions within the chat room. In this session, it is important to note that communication is *bidirectional*: the client can send chat messages to the chat room, and can also receive messages from the chat room. When a user sends a chat message to the room, it should be broadcast to all participants of the room. This is encoded using the `par` block, where each scope contains actions that can be performed in an interleaved fashion.

The subsession initiation is implemented using the custom `initiates` construct:

```
Role initiates ProtocolName(Roles) { SuccessBlock } handle(FailureName) { FailureBlock }
```

The `initiates` construct initiates a subsession for the protocol `ProtocolName` with the given roles (in this case, `ClientThread` is an internal invitation, inviting the actor playing the `ClientThread` role in `ChatServer` to play the `ClientThread` role in `ChatSession`, and an external invitation for another actor to play the `ChatRoom`).

The `SuccessBlock` block describes the interactions to occur should the subsession complete successfully, whereas the `handle` constructs allow different interactions to take place should the subsession terminate abnormally. In this case, we allow a client to join a different room should the chat room go offline (note that `ParticipantOffline` is the error raised when an actor in a subsession is terminated while it is still needed in the subsession).

A participant may disconnect at any time. If this is the case, then they should be deregistered from any chat rooms to which they belong.

3 Scribble Protocol

We specify the communication within the system using two protocols: `ChatServer`, which describes the interactions prior to joining a chat room, and `ChatSession`, which describes the interactions within the chat session. The session types are written in the Scribble protocol description language [2, 4].

```
module src.com.simonjf.ScribbleExamples.ChatServer.ChatServer;

type <erlang> "string" from "" as Username;
type <erlang> "string" from "" as Password;
type <erlang> "tuple" from "" as ClientData;
type <erlang> "string" from "" as String;
type <erlang> "string" from "" as RoomName;
type <erlang> "list" from "" as StringList;
type <erlang> "string" from "" as PID;


global protocol ChatServer(role ClientThread, role RoomRegistry) {
    rec ClientChoiceLoop {
        // When not in a room, a user can either join, create, or list the rooms.
        choice at ClientThread {
            lookupRoom(RoomName) from ClientThread to RoomRegistry;
            choice at RoomRegistry {
                roomPID(RoomName, PID) from RoomRegistry to ClientThread;
                ClientThread initiates ChatSession(ClientThread, new ChatRoom) {
                    continue ClientChoiceLoop;
                } handle (ParticipantOffline) {
                      continue ClientChoiceLoop;
                }
            } or {
                roomNotFound(RoomName) from RoomRegistry to ClientThread;
            }
        } or {
            createRoom(RoomName) from ClientThread to RoomRegistry;
            choice at RoomRegistry {
                createRoomSuccess(RoomName) from RoomRegistry to ClientThread;
            } or {
                roomExists(RoomName) from RoomRegistry to ClientThread;
            }
        } or {
            listRooms() from ClientThread to RoomRegistry;
            roomList(StringList) from RoomRegistry to ClientThread;
        }
        continue ClientChoiceLoop;
    }
}

global protocol ChatSession(role ClientThread, role ChatRoom) {
    // Communication is bidirectional: a client can send messages to the
    // server (either chat messages or control messages), and the server
    // can send messages to the client (incoming chat messages)
    par {
        rec ClientLoop {
            choice at ClientThread {
                outgoingChatMessage(String) from ClientThread to ChatRoom;
                continue ClientLoop;
            } or {
                leaveRoom() from ClientThread to ChatRoom;
            }
        }
    } and {
        rec ServerLoop {
            incomingChatMessage(String) from ChatRoom to ClientThread;
            continue ServerLoop;
        }
    }
}
```

3.1 Local Projections

3.1.1 ClientThread

```
module src.com.simonjf.ScribbleExamples.ChatServer.ChatServer_ClientThread;

type <erlang> "string" from "" as Username;
type <erlang> "string" from "" as Password;
type <erlang> "tuple" from "" as ClientData;
type <erlang> "string" from "" as String;
type <erlang> "string" from "" as RoomName;
type <erlang> "list" from "" as StringList;
type <erlang> "string" from "" as PID;

local protocol ChatServer at ClientThread(role ClientThread,role RoomRegistry) {
    rec ClientChoiceLoop {
        choice at ClientThread {
            lookupRoom(RoomName) to RoomRegistry;
            choice at RoomRegistry {
                roomPID(RoomName,PID) from RoomRegistry;
                ClientThread initiates ChatSession( ClientThread , new ChatRoom ) {
                    continue ClientChoiceLoop;
                } handle (ParticipantOffline) {
                    continue ClientChoiceLoop;
                }
            } or {
                roomNotFound(RoomName) from RoomRegistry;
            }
        } or {
            createRoom(RoomName) to RoomRegistry;
            choice at RoomRegistry {
                createRoomSuccess(RoomName) from RoomRegistry;
            } or {
                roomExists(RoomName) from RoomRegistry;
            }
        } or {
            listRooms() to RoomRegistry;
            roomList(StringList) from RoomRegistry;
        }
        continue ClientChoiceLoop;
    }
}

local protocol ChatSession at ClientThread(role ClientThread,role ChatRoom) {
    par {
        rec ClientLoop {
            choice at ClientThread {
                outgoingChatMessage(String) to ChatRoom;
                continue ClientLoop;
            } or {
                leaveRoom() to ChatRoom;
            }
        }
    } and {
        rec ServerLoop {
            incomingChatMessage(String) from ChatRoom;
            continue ServerLoop;
        }
    }
}
```

3.1.2 RoomRegistry

```
module src.com.simonjf.ScribbleExamples.ChatServer.ChatServer_RoomRegistry;

type <erlang> "string" from "" as Username;
type <erlang> "string" from "" as Password;
type <erlang> "tuple" from "" as ClientData;
type <erlang> "string" from "" as String;
type <erlang> "string" from "" as RoomName;
type <erlang> "list" from "" as StringList;
type <erlang> "string" from "" as PID;

local protocol ChatServer at RoomRegistry(role ClientThread,role RoomRegistry) {
    rec ClientChoiceLoop {
        choice at ClientThread {
            lookupRoom(RoomName) from ClientThread;
            choice at RoomRegistry {
                roomPID(RoomName,PID) to ClientThread;
            } or {
                roomNotFound(RoomName) to ClientThread;
            }
        } or {
            createRoom(RoomName) from ClientThread;
            choice at RoomRegistry {
                createRoomSuccess(RoomName) to ClientThread;
            } or {
                roomExists(RoomName) to ClientThread;
            }
        } or {
            listRooms() from ClientThread;
            roomList(StringList) to ClientThread;
        }
        continue ClientChoiceLoop;
    }
}
```

3.1.3 ChatRoom

```
module src.com.simonjf.ScribbleExamples.ChatServer.ChatServer_ChatRoom;

type <erlang> "string" from "" as Username;
type <erlang> "string" from "" as Password;
type <erlang> "tuple" from "" as ClientData;
type <erlang> "string" from "" as String;
type <erlang> "string" from "" as RoomName;
type <erlang> "list" from "" as StringList;
type <erlang> "string" from "" as PID;

local protocol ChatSession at ChatRoom(role ClientThread,role ChatRoom) {
    par {
        rec ClientLoop {
            choice at ClientThread {
                outgoingChatMessage(String) from ClientThread;
                continue ClientLoop;
            } or {
                leaveRoom() from ClientThread;
            }
        }
```

```
    } and {
        rec ServerLoop {
            incomingChatMessage(String) to ClientThread;
            continue ServerLoop;
        }
    }
}
```

4 Implementation using `monitored-session-erlang`

The first step (as with any application using `monitored-session-erlang`) is to define a configuration file specifying the roles which each actor may play in the different protocols.

Listing 1: Configuration file for `mse-chat`

```
-module(mse_chat_config).
-compile(export_all).

config() ->
  [{mse_chat_client, [{"ChatServer", ["ClientThread"]},
                      {"ChatSession", ["ClientThread"]}]},
   {mse_chat_room_manager, [{"ChatServer", ["RoomRegistry"]}]},
   {mse_chat_room_instance, [{"ChatSession", ["ChatRoom"]}]}].
```

Listing 1 shows the `monitored-session-erlang` configuration file for `mse-chat`. The `mse_chat_client` actor can play the `ClientThread` role in both the `ChatServer` and `ChatSession` protocols; the `mse_chat_room_manager` actor can play the `RoomRegistry` role in the `ChatServer` protocol; and the `mse_chat_room_instance` actor can play the `ChatRoom` role in the `ChatSession` protocol.

The implementation encodes some interesting communication patterns: firstly, the inter-actor communication is driven by incoming TCP messages from an external client, and thus messages must be dispatched to the correct session; and secondly different sessions may *interact*: a message from one client to a chat room triggers a message to be sent to other clients in that chat room.

The `mse_chat_tcp_server` accepts incoming connections, and creates a new `mse_chat_client` to handle incoming messages: in Erlang, the `gen_tcp` socket library dispatches incoming TCP messages to the client as messages.

Recall that the `mse_chat_client` can partake in both `ChatServer` and `ChatSession` protocols. Upon receiving messages from the remote host—in this case, a chat client program—the process must ensure that a message is sent to the correct session. For example, a 'create room' packet must be handled by the `ChatServer` session, whereas a 'send chat message' packet must be handled by the `ChatSession` session. In order to do this, we make essential use of the `become` co-operative role-switching capability. When a client actor is started, it initiates a `ChatServer` session. This is registered using the `main_thread` key. Additionally, when the client has joined a chat room, it begins a `ChatSession` session, and this is registered using the `chat_session` key.

```
ssactor_conversation_established("ChatServer", "ClientThread", _CID, ConvKey, State) ->
  error_logger:info_msg("Conv established~n"),
  conversation:register_conversation(main_thread, ConvKey),
  {ok, State};
ssactor_conversation_established("ChatSession", "ClientThread", _CID, ConvKey, State) ->
  conversation:register_conversation(chat_session, ConvKey),
  {ok, State}.
```

Upon handling an incoming packet, the actor switches to the appropriate role using the `become` function, and can subsequently send a message in the appropriate session. The cases for room creation and chat are as follows:

```
...
if Command == "CHAT" ->
     [_|SplitChatMessage] = SplitMessage,
     ChatMessage = string:join(SplitChatMessage, ":"),
     conversation:become(MonitorPID, chat_session, "ClientThread",
                         chat, [ChatMessage]),
     State;
   Command == "CREATE" ->
     [RoomName|_Rest] = PacketRemainder,
     conversation:become(MonitorPID, main_thread, "ClientThread",
                         create_room, [RoomName]),
     State;
...
```

The `become` function subsequently invokes the `ssactor_become` callback, which is provided with the `ConvKey` for the particular session that is needed:

```
...
ssactor_become("ChatServer", "ClientThread", create_room, [RoomName],
               ConvKey, State) ->
  handle_create_room(ConvKey, RoomName),
  {ok, State};
ssactor_become("ChatSession", "ClientThread", chat, [Message],
               ConvKey, State) ->
  handle_chat(ConvKey, Message, State),
  {ok, State};
...
```

The `handle_create_room` and `handle_chat` functions send the appropriate messages to the room registry and the chat room respectively.

Another interesting pattern to discuss is how messages can be delivered to all clients in a chat room. Recall that a session is associated with a unique identifier; we register and store this identifier when a client joins a chat room:

```
ssactor_conversation_established("ChatSession", "ChatRoom", CID, ConvKey, State) ->
  conversation:register_conversation(CID, ConvKey),
  ClientList = State#room_state.room_members,
  NewClientList = [CID|ClientList],
  NewState = State#room_state{room_members=NewClientList},
  {ok, NewState}.
```

Consequently, we have a list of unique session identifiers, with each session identifier registered to its `ConvKey`. By iterating over the list of stored session IDs, we can switch to the appropriate session, and send the message to all participants as required.

```
handle_broadcast_message(ConvKey, SenderName, Message, State) ->
  RoomMembers = orddict:to_list(State#room_state.room_members),
  lists:foreach(fun(CID) ->
                      conversation:become(ConvKey, CID, "ChatRoom",
                                          broadcast, [SenderName, Message])
                  end,
                RoomMembers).

ssactor_become("ChatSession", "ChatRoom", broadcast, [SenderName, Message], ConvKey, State) ->
  mse_chat_client:chat_message(ConvKey, SenderName, Message),
  {ok, State}.
```

4.1 Failure Model

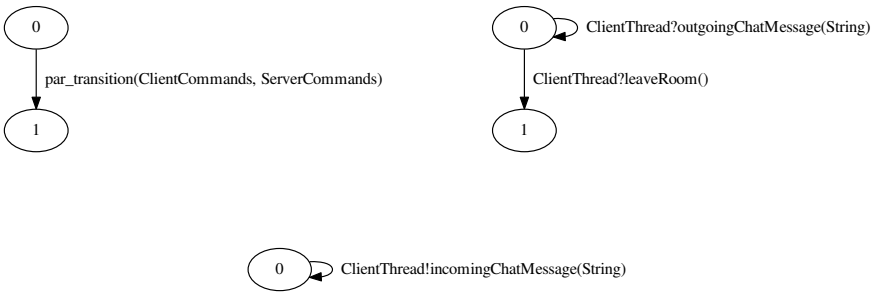
An `mse_chat_client` is linked to the socket connected to the client application. Should the socket be closed, then the `mse_chat_client` actor will terminate. Push-based failure detection then detects that the session cannot continue, and invokes the `ssactor_conversation_ended` callback in the chat room instance, which removes the session ID from the list.

Should the chat room terminate while a participant is involved, the system will end the subsession with the reason `ParticipantOffline`, which is handled by the `handle` block. At this point, the user can join another room.

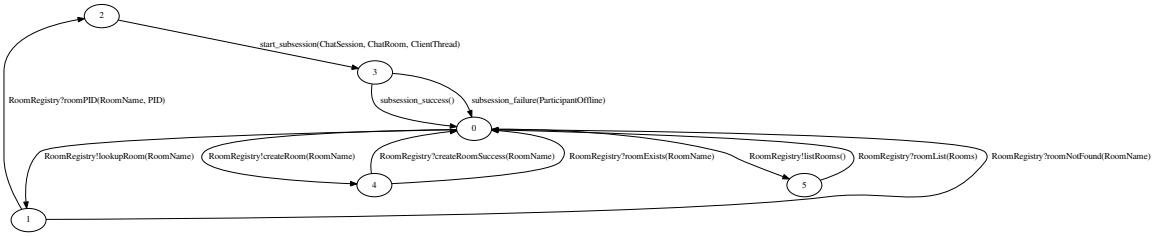
5 FSMs

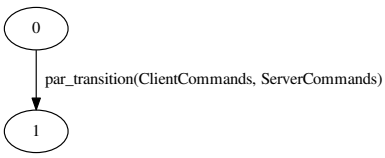
In this section, we detail the monitor FSMs for each of the roles. Note that we make use of the nested FSM optimisation of Hu et al. [3] in order to efficiently encode parallel composition: instead of encoding `par` blocks by defining states for each possible interleaving, we encode each branch of a parallel composition as its own FSM. The 'join' transition (denoted by `par_transition` in the diagrams) may be taken when each nested FSM is in a final state.

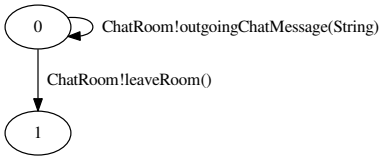
5.1 ChatRoom

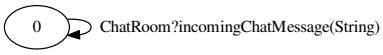


5.2 ClientThread

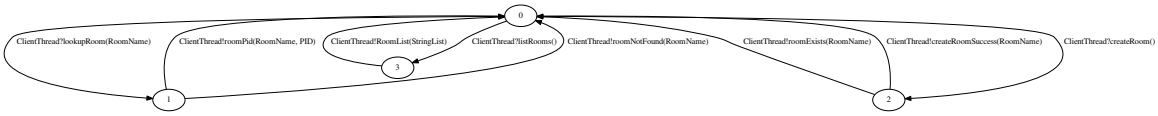




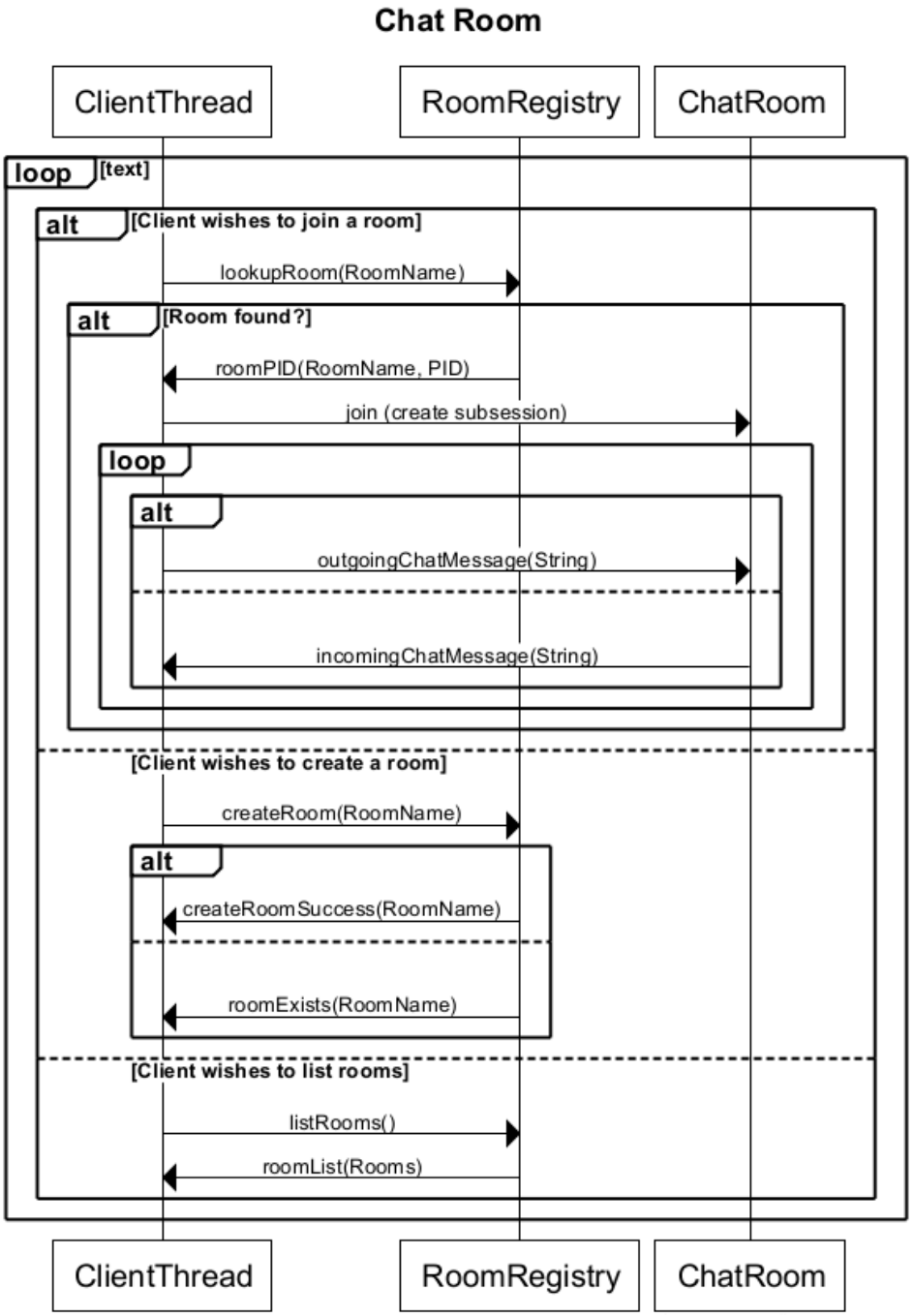




5.3 RoomRegistry



6 Sequence Diagram



References

[1] Simon Fowler. Monitoring Erlang/OTP applications using multiparty session types. Master's thesis, August 2015.

[2] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. *Scribbling Interactions with a Formal Foundation*, volume 6536 of *Lecture Notes in Computer Science*, chapter 4,

pages 55–75. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-19055-1. doi: 10.1007/978-3-642-19056-8_4. URL http://dx.doi.org/10.1007/978-3-642-19056-8_4.

[3] Raymond Hu, Rumyana Neykova, Nobuko Yoshida, Romain Demangeon, and Kohei Honda. Practical interruptible conversations. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification*, volume 8174 of *Lecture Notes in Computer Science*, pages 130–148. Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-40787-1_8. URL http://dx.doi.org/10.1007/978-3-642-40787-1_8.

[4] The Scribble Team. Scribble Language Reference. http://www.doc.ic.ac.uk/\~{}rhu/scribble/langref.html, 2013.