An Erlang Implementation of Multiparty Session Actors

Simon Fowler
The University of Edinburgh
Edinburgh, UK
simon.fowler@ed.ac.uk

By requiring co-ordination to take place using explicit message passing instead of relying on shared memory, actor-based programming languages have been shown to be effective tools for building reliable and fault-tolerant distributed systems. Although naturally communication-centric, communication patterns in actor-based applications remain informally specified, meaning that errors in communication are detected late, if at all.

Multiparty session types are a formalism to describe, at a global level, the interactions between multiple communicating entities. This article describes the implementation of a prototype framework for monitoring Erlang/OTP `gen_server` applications against multiparty session types, showing how previous work on multiparty session actors can be adapted to a purely actor-based language, and how monitor violations and termination of session participants can be reported in line with the Erlang mantra of ‘let it fail’. Finally, the framework is used to implement two case studies: an adaptation of a freely-available DNS server, and a chat server.

1 Introduction

Programming concurrent and distributed systems is a challenge—the introduction of concurrency and distribution introduces issues such as deadlocks, race conditions, and node failures; issues which simply do not arise when developing single-threaded applications.

The actor model is a model of concurrency introduced by Hewitt et al. [11] and discussed in the context of distributed computing by Agha [1]. *Actors*—entities with a unique ID and a message queue called a mailbox—react to incoming messages in three ways: by asynchronously sending a finite set of messages to other actors; spawning a finite number of new actors; and changing how they react to future messages.

The actor model provides an ideal theoretical basis for programming languages designed for programming robust and fault-tolerant distributed systems. Programming languages such as Erlang [2] and Elixir take actors as primitive entities, implemented as lightweight processes which communicate only through explicit message passing. By eschewing shared memory as a method of co-ordination, applications are naturally built in a style fostering fault isolation, scalability, and modularity. Of particular note is Erlang’s approach to failure handling—an approach succinctly described as ‘let it fail’, wherein processes should terminate upon encountering a fault, and be restarted by a supervisor process.

Moving to a communication-centric programming paradigm has its own issues, however: in particular, how do we document the communication patterns expected within an application, and how do we ensure that the application conforms to these communication patterns? *Multiparty Session Types* [14] are a type formalism in which a *global type* describing interactions between participants in a session—a series of interactions between participants—can be projected into *local types* describing the interaction from the point of view of an individual participant. It is then possible to check conformance either statically using a type checker, or dynamically through runtime monitoring techniques.

We build upon the work of Neykova and Yoshida [17], who describe a conceptual framework in which actors are treated as entities which are simultaneously involved in multiple possibly-interacting sessions.

Their conceptual framework is realised as a library building upon the Cell actor framework in Python.

While actors can be emulated, Python remains an imperative language with mutability and shared memory concurrency. This article seeks to demonstrate the applicability of this conceptual framework in Erlang, which as well as taking actors as primitive, is a functional language which forbids co-ordination via shared memory. In addition, we investigate how dynamic monitoring of communication against multiparty session types may be integrated with the 'let it fail' methodology of Erlang.

1.1 Contributions

The contributions of this article are:

- A tool, `monitored-session-erlang`[1], for monitoring communication in Erlang applications, based on the session actor framework of Neykova and Yoshida [17]. The implementation uses native actor functionality as opposed to emulating actors using AMQP and Python Greenlets, and has a simpler invitation workflow as a result. Additionally, we allow actors to take part in multiple *instances* of a session, which is necessary to implement server applications.
- An extension of the Scribble protocol description language to use *subsessions* [9] for introducing participants midway through a session, and for structuring possibly-failing sessions.
- A discussion of ways to detect and act upon failures, including a two-phase commit to ensure messages are accepted in a multicast, and reachability analysis to ascertain whether a failed participant is required in the remainder of a session.
- Two larger examples of session-based communication using the framework: a chat server, and an adaptation of a freely-available DNS server.

2 An Overview of Multiparty Session Actors

Multiparty session types are a formalism to allow a protocol—a series of typed interactions between multiple participants—to be described as a *global type*. As the name suggests, global types are a *global* formalism, describing all of the interactions in that particular protocol. Global types can be projected into *local* types, which describe the protocol from the point of view of a single participant.

In the traditional view of runtime monitoring of multiparty session types, each process is monitored by a single monitor, which checks incoming and outgoing messages to see whether they conform to a local type. The actor model, on the other hand, lends itself to a different model of monitoring. Actors are naturally event-driven: upon processing a message from a mailbox, an actor can send a finite set of messages; spawn a finite set of new actors; and change the way it behaves upon encountering the next message. A more appropriate style of monitoring pioneered by Neykova and Yoshida [17] is to treat actors as entities that can take part in multiple sessions. The core ideas behind multiparty session actors are that:

- Actors may be involved in multiple sessions simultaneously.
- Actors may play multiple roles in each session.
- A message received in the course of one session may trigger an interaction in another session.

In this setting, actors become containers for monitors and handlers for incoming session messages. Actors may be involved in multiple roles in multiple sessions, with the ability to co-ordinate between them, and with actor-wide state shared amongst the handlers for each session.

To describe protocols, we use Scribble [21], a human-readable protocol description language based

[1] `monitored-session-erlang` can be found online at `http://www.github.com/SimonJF/monitored-session-erlang`. Chat server: `http://www.github.com/SimonJF/mse-chat`, DNS server: `http://www.github.com/SimonJF/erlang-dns`.

on the theory of multiparty session types. Scribble is realised as a Java-based toolchain[2], including components for parsing, validating well-formedness, and creating local projections of global types.

2.1 A Chat Server

We illustrate the main concepts of the multiparty session actor method of designing actor-based applications in the context of our Erlang implementation, `monitored-session-erlang`.

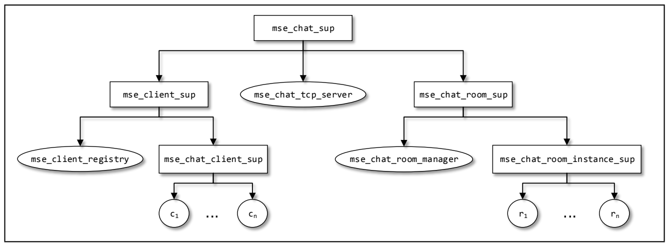


Figure 1: Supervision Tree for Chat Server

Erlang/OTP applications are structured for reliability using a design pattern known as *supervision hierarchies*, where workers (actors which perform computations) can be restarted by *supervisors* if they terminate. Instead of attempting to recover from an unexpected or erroneous state, actors are designed to terminate and be restarted. Figure 1 shows the supervision tree for the chat system. The nodes in boxes denote supervisor actors, which do not participate in interactions themselves, but restart their child actors should the child actors terminate. Of interest to the protocol are three actors: `mse_chat_client` ($c_1, \dots, c_n$ in the diagram), representing a chat client actor; `mse_chat_room_instance` ($r_1, \dots, r_n$ in the diagram), representing an instance of a chat room; and `mse_chat_room_manager`, which maintains a registry of chat rooms.

The `mse_chat_tcp_server` actor listens on a socket and accepts new clients, spawning a new `mse_chat_client` to handle requests from the new client. A client can either create or join a room by sending a message to `mse_chat_room_manager`, which checks whether the room exists. Once a client is registered with the room, any chat messages sent should be distributed to all other participants in the room. The client can leave the session at any time and should be deregistered from any rooms to which it is registered.

```
global protocol ChatServer(role ClientThread, role RoomRegistry) {
 rec ClientChoiceLoop {
  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);
   } 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){
 par {
  rec ClientLoop {
   choice at ClientThread {
    outgoingChatMessage(Msg) from ClientThread to ChatRoom;
    continue ClientLoop;
   } or { leaveRoom() from ClientThread to ChatRoom; }
  }
 } and {
  rec ServerLoop {
   incomingChatMessage(Msg) from ChatRoom to ClientThread;
   continue ServerLoop;
  }
 }
}
```

Figure 2: Global Protocols for Chat Server

Figure 2 shows two Scribble protocols describing the chat server. The `ChatServer` protocol describes the interactions between a client and the room registry before joining a room. Interactions are of the form

[2] http://www.github.com/scribble/scribble-java

`messageName(Payload)from FromRole to ToRole1, ..., ToRoleN`, meaning that the role with name `FromRole` sends a message with name `messageName` and payload type `Payload` to roles with names `ToRole1` ... `ToRoleN`.

A common pattern in Erlang involves sending process IDs. We make use of a small extension to Scribble to implement the theory of *nested protocols*, or *subsessions* [9] which allow a child session to be initiated with some participants invited from the current session, and some invited externally. This is implemented using the `initiates` construct (Line 7), stating that `ClientThread` starts the `ChatSession` protocol, externally inviting another actor to fulfil the `ChatRoom` role.

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

We begin by creating a configuration file associating roles with actors: `mse_chat_client` can play `ClientThread` in both `ChatServer` and `ChatSession`, `mse_chat_room_manager` plays `RoomRegistry` in `ChatServer`, and `mse_chat_room_instance` plays `ChatRoom` in `ChatSession`.

Instead of using send and receive primitives directly, Erlang developers make heavy use of *OTP behaviours*, which provide boilerplate functionality and require a developer to implement a number of callbacks in order to provide application logic. As an example, the `gen_server` behaviour abstracts over Erlang's communication primitives to provide an event loop, invoking callbacks such as `handle_cast` to process incoming messages. To participate in sessions in `monitored-session-erlang`, actors must implement the `ssa_gen_server` behaviour, described further in Section 3.6.

We begin by looking at the implementation of `mse_chat_room_manager`. When an actor is spawned, the `ssactor_init` callback is invoked with the arguments and an *initiation key* for starting sessions, and should return the initial actor state. When an actor is invited to join a session, the `ssactor_join` callback is invoked, and the return value defines whether the invitation is accepted or declined. The `ssactor_handle_message` callback is invoked when a session message has been received and accepted by the monitor.

```
ssactor_init(_Args, _InitKey) ->
  #room_manager_state{rooms=orddict:new()}.
ssactor_join(_, _, _, State) ->
  {accept, State}.
ssactor_conversation_established("ChatServer",
    "RoomRegistry", _CID, ConvKey, State) ->
  {ok, State}.
ssactor_handle_message("ChatServer", "RoomRegistry",
    _, _, "lookupRoom", [RoomName], State, ConvKey) ->
  handle_get_room(ConvKey, RoomName, State),
  {ok, State}.

handle_get_room(ConvKey, RoomName, State) ->
  RoomDict = State#room_manager_state.rooms,
  case orddict:find(RoomName, RoomDict) of
    {ok, RoomPID} ->
     mse_chat_client:found_room_pid(ConvKey,
        RoomName, RoomPID);
    error ->
     mse_chat_client:room_not_found(ConvKey, RoomName)
  end.
```

Figure 3: Excerpt from `mse_chat_room_manager` implementation

Figure 3 shows an excerpt from the implementation of `mse_chat_room_manager`. The actor creates a new map structure in the `ssactor_init` callback to hold rooms that will be created later; always accepts invitations to join sessions; and does nothing when the session is established. The `ssactor_handle_message` callback details how the `lookupRoom` message in the `ChatServer` protocol is handled: the actor queries the map between room names and PIDs, sending the PID to the requesting actor if it is found, and sending a "room not found" message if not. In line with Erlang development practices, message passing is abstracted as an API call: as an example, the `mse_chat_client:room_not_found` function is defined as `room_not_found(ConvKey, RoomName)->conversation:send(ConvKey, ["ClientThread"], "roomNotFound", [RoomName]).`

The `conversation:send` function sends a session message `roomNotFound` with payload `RoomName` to `ClientThread`; `ConvKey` (provided by `ssactor_handle_message`) is used to identify the correct monitor. The implementation of the client follows a similar pattern. We show an excerpt of the implementation of the client in Figure 4; note that a `ChatServer` session is initiated in `ssactor_init`.

```
1  ssactor_init([ClientID, ClientSocket], InitKey) ->
2    State =
3     #client_state{client_id=ClientID,
4       client_name=undefined, client_socket=ClientSocket,
5       init_key=InitKey},
6    conversation:start_conversation(InitKey,
7      "ChatServer", "ClientThread"),
8    inet:setopts(ClientSocket, [{active, true}]),
9    State.
10
11 ssactor_conversation_established("ChatServer",
12     "ClientThread", _CID, ConvKey, State) ->
13   conversation:register_conversation(main_thread, ConvKey),
14   {ok, State};
15
16
17 handle_message(Message, State) ->
18   InitKey = State#client_state.init_key,
19   SplitMessage = string:tokens(Message, ":"),
20   [Command|PacketRemainder] = SplitMessage,
21   NewState =
22     if Command == "JOIN" ->
23        [RoomName|_Rest] = PacketRemainder,
24        conversation:become(InitKey, main_thread,
25          "ClientThread", join_room, [RoomName]),
26        State;
27        [ ... ]
28     end,
29   {noreply, NewState}
30
31 ssactor_become("ChatSession", "ClientThread", chat,
32                [Message], ConvKey, State) ->
33   handle_chat(ConvKey, Message, State),
34   {ok, State};
```

Figure 4: Excerpt from the `mse_chat_client` implementation

Recall that the `mse_chat_client` participates in both the `ChatServer` and `ChatSession` protocols. Upon receiving messages from a chat client program, the process must ensure that a message is sent in the correct session. For example, a 'create room' packet from the client application must be handled by `ChatServer`, whereas a 'send chat message' packet must be handled by `ChatSession`.

To accomplish this, we make essential use of the ability to switch between roles. In our implementation (shown in Figure 4), we can *register* a session to a key (Line 13), and can use this key to switch to the session (Line 24). As an example, the `ClientThread` session is registered with `main_thread` key, and switches to this session in order to send a `lookupRoom` packet to the room registry. The `mse_chat_room_instance` actor uses a similar approach to broadcast messages to the chat room.

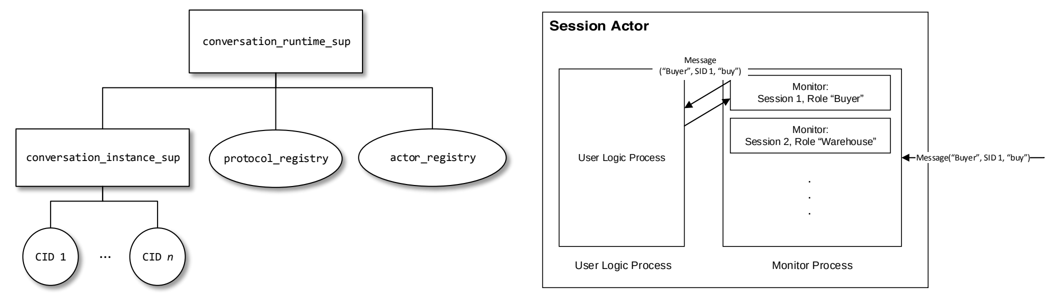
3 Design and Implementation of `monitored-session-erlang`

3.1 System Overview

The `monitored-session-erlang` system is implemented as an Erlang library. The supervision tree for the system is shown in Figure 5a: `conversation_runtime_sup` is the root supervisor of the system, which restarts top-level processes should they fail. The `protocol_registry` process associates protocols and roles with FSMs used for monitoring, and the `actor_registry` process maintains a list of active actors which are registered to take part in sessions.

Each session is associated with a *coordinating process* (depicted as `CID 1 ...CID n` in Figure 5a); coordinating processes are used to coordinate actions such as setting up a session and failure handling. Coordinating processes are arranged as children of the `conversation_instance_sup` process, but are not restarted should they fail. As a high-level overview, the system works as follows:

- Local projections of protocols are used to generate monitors based on communicating finite-state machines (CFSMs). A configuration file defines which roles, in which protocols, actors may fulfil.
- When an actor is spawned, it is added to the `actor_registry`, allowing it to participate in sessions.
- A session *initiator* begins a session. At this point, eligible actors are invited to fulfil each role in the protocol. When all roles are fulfilled, the participants of the session are notified that the session has been initiated successfully and are provided with the monitor FSM to use. Conversely, if it is not possible to fulfil all of the roles then all actors registered in the session are notified of the failure.
- Session messages are sent using a session API, and processed by session actors. All communication using the session API is mediated by monitors, and messages which do not conform to the protocol


conversation_runtime_sup
conversation_instance_sup
protocol_registry
actor_registry
CID 1
...
CID *n*
Session Actor
Message ("Buyer", SID 1, "buy")
Monitor: Session 1, Role "Buyer"
Monitor: Session 2, Role "Warehouse"
User Logic Process
Message("Buyer", SID 1, "buy")
User Logic Process
Monitor Process

(a) `monitored-session-erlang` supervision tree (b) Erlang Session Actors

Figure 5: Components of `monitored-session-erlang`

are rejected, with an exception thrown in the sender.

- Due to the supervision tree structure within the Erlang applications, we *cannot assume that participants are alive for the duration of the session*. Consequently, we provide failure detection mechanisms, which detect when the session can no longer safely proceed.
- When the session is over (or an actor ends it prematurely due to an error), a participant calls a function which notifies other participants in the session that the session has ended.

3.2 Erlang Session Actors

A considerable insight due to Neykova and Yoshida is that instead of having a single monitor, session actors can fulfil multiple roles in multiple protocols. At an abstract level, we can think of an Erlang session actor as containing seven components: a process ID, a term currently being evaluated, a mailbox, an actor state, a monitor lookup table, a routing table mapping session / role pairs to actor PIDs, and a message handler function. Each monitor can be uniquely identified using a pair of a session ID and a role.

Role Registration Neykova and Yoshida [17] associate actors with roles using Python decorators, which is an appealing and intuitive method of associating message handlers, protocols, and roles together. It is unclear how it would be possible to allow multiple *instances* of sessions, which is an important requirement when writing server applications. For example, in the chat server, a single `mse_chat_room_registry` actor can take part in multiple instances of the `ChatServer` protocol to connect multiple different clients to chat rooms. Instead, in `monitored-session-erlang`, actors are associated with roles using a configuration file.

Process Structure When working in the setting of a functional, actor-based language with immutable variables, it is convenient to have separate processes for monitoring and user logic (Figure 5b).

The solution used in `monitored-session-erlang` is to provide a *session key* or `ConvKey` to the user. A session key is a 3-tuple (M,R,S) where M is the process ID of the monitor process, R is the name of the role that the participant is playing in the current message handler, and S is the process ID of the coordinating process for the current session. To the user, the session key is an opaque, abstract value. Passing the session key as a parameter to the send operation allows the correct monitor to be identified in order to check the outgoing message and update the monitor state. The monitor state will be updated in the monitoring process, and consequently there is no requirement for linearity tracking.

3.3 Session Initiation

A strength of the multiparty session actor framework is that messages can be sent to a role name instead of requiring a concrete actor address, but this introduces the issue of associating roles with endpoints at the beginning of the protocol. The work of Neykova and Yoshida [17] makes essential use of the

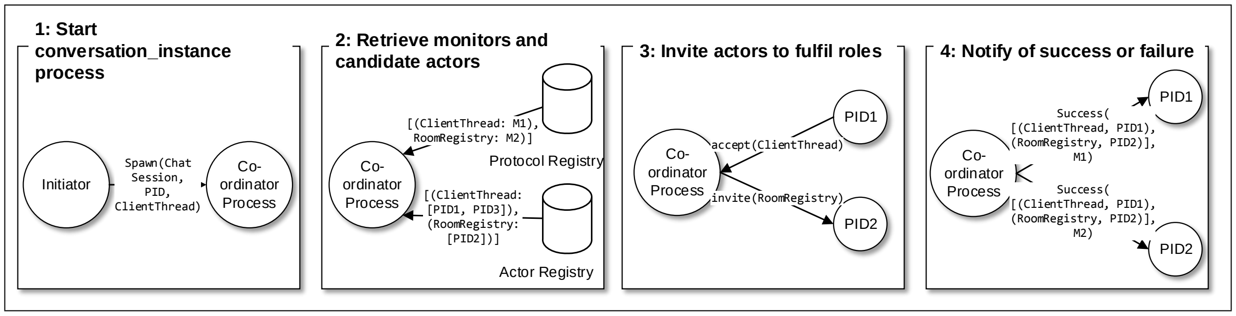

1: Start conversation_instance process
Initiator
Spawn(Chat Session, PID, ClientThread)
Co-ordinator Process
2: Retrieve monitors and candidate actors
[(ClientThread: M1), RoomRegistry: M2)]
Protocol Registry
[(ClientThread: [PID1, PID3]), (RoomRegistry: [PID2])]
Actor Registry
Co-ordinator Process
3: Invite actors to fulfil roles
Co-ordinator Process
accept(ClientThread)
invite(RoomRegistry)
PID1
PID2
4: Notify of success or failure
Co-ordinator Process
Success([(ClientThread, PID1), (RoomRegistry, PID2)], M1)
Success([(ClientThread, PID1), (RoomRegistry, PID2)], M2)
PID1
PID2

Figure 6: Actor Invitation Workflow

Advanced Message Queueing Protocol (AMQP), which provides abstractions known as *exchanges* to distribute messages to other AMQP entities. Session initiation requires four exchanges, and all session communication is routed through a single exchange per session instance.
As Erlang is an actor-based language, we do not use AMQP, resulting in a simpler invitation workflow. We require two centralised registries: a `protocol_registry`, which contains precomputed monitor FSMs for each role in a protocol, and an `actor_registry`, which associates active session actors with the roles they may fulfil. More concretely, the `actor_registry` is a map $\mathit{Protocol\ Name} \mapsto (\mathit{Role} \mapsto \mathit{Actor\ PID})$. The procedure for initiating a session (shown in Figure 6) is as follows:

1. A session actor—the *session initiator*—requests that a session is initiated, specifying a protocol name, and the role it wishes to take in the protocol.
2. A `conversation_instance` process is spawned to co-ordinate session actions.
3. The `conversation_instance` process contacts the `protocol_registry` process to retrieve the list of roles and monitors used in the process, and contacts the `actor_registry` process to retrieve the list of actor process IDs which may fulfil the roles in the session.
4. The `conversation_instance` process invites eligible actors to fulfil each role. Once all roles have been fulfilled, each actor is notified, invoking the `ssactor_conversation_established` callback. If it is not possible to fulfil a role, for example because all active session actors have declined the invitation to fulfil the role, then the invitation process is aborted. All actors already invited to fulfil roles in the protocol are notified, resulting in the invocation of the `ssactor_conversation_error` callback.

3.4 Monitoring

Messages are checked against monitors based on communicating finite-state machines [4]. Firstly, global types are projected into local types, and the local types are used to construct monitors based on communicating finite state machines using an algorithm based on that of Deniélou and Yoshida [10].

Transitions between states are predicated on send and receive operations. The monitor generated for the `ChatServer` protocol from Section 2.1 projected at the `RoomRegistry` role is shown in Figure 7.

The monitor generation algorithm also makes use of the nested FSM optimisation described by Hu et al. [15] to ensure that generated monitors are polynomial in the size of the global type in the presence of parallel composition. Instead of generating states for all possible interleavings, the algorithm generates a separate monitor for each block of interactions composed in parallel, and the outer monitor can only progress when all nested monitors are in a terminal state.

A notable difference to standard CFSMs is that, following the design of Scribble, we allow transitions to be predicated on sending a message to a *set* of recipients, in a multicast fashion. The failure detection mechanisms in Section 4.1 ensure all recipients accept the message.

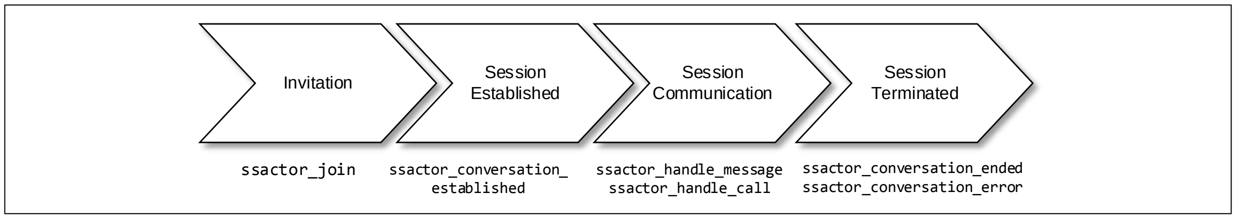


Figure 8: Session Lifecycle

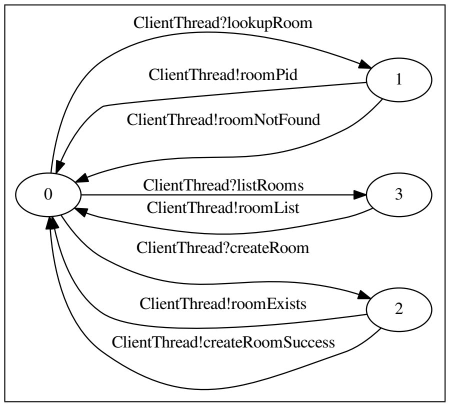


Figure 7: Monitor for ClientThread role of ChatServer protocol

Monitor generation takes place when the actor system is started, and the generated monitors are stored in the `protocol_registry` actor. The second aspect of monitoring is the monitoring runtime: once a monitor has been generated, it may be used to check incoming and outgoing messages against the local specification for a type. The monitor process for an actor contains a hashtable mapping session ID / role pairs to monitors. Checking a message against a monitor involves checking whether any transitions can be made from the current monitor state. If not, then an exception is raised.

3.5 Sending Messages

A user may send a monitored session message by calling the `conversation:send` function. In order to send a message, four pieces of information are required: a session key `ConvKey`; a list of recipients; a message name; and a list of values. Recall that the monitoring process is *external* to the logic process, and the session key contains the role name and the session ID, which uniquely identify a monitor.

When `conversation:send` is called, a message is sent to the actor monitor process, which retrieves the appropriate monitor using the role name and session ID. Should the message be accepted by the monitor, the role will be resolved to the PID of the monitor of the receiving process by the routing table. At this point, a synchronous call will be made to the remote monitor to ascertain whether the message can be accepted: if so, both monitors are advanced, and the actor will proceed. The monitoring process is synchronous, in order to allow errors to be reported to the user.

If a monitor rejects a message, an exception is thrown. Such behaviour is consistent with the Erlang design ideology of letting a process fail if its behaviour deviates from a specification. As a result, `monitored-session-erlang` *extends the let-it-fail ideology to communication patterns.*

3.6 The `ssa_gen_server` Behaviour

In keeping with the Erlang/OTP method of designing applications, `monitored-session-erlang` provides a behaviour, `ssa_gen_server`, which contains callbacks that should be implemented by Erlang session actors. Figure 8 describes the session lifecycle, and the expected callbacks.

During session initiation, actors eligible to fulfil a role will be invited to participate in the session, triggering the `ssactor_join` callback. The expected return value consists of a pair of either the atom `accept` or `decline`, and an updated actor state. Once all roles in the protocol have been fulfilled, the `ssactor_conversation_established` callback is invoked. At this point, the actor can begin to communicate using session messages. When a message is received, `ssactor_handle_message` is invoked.

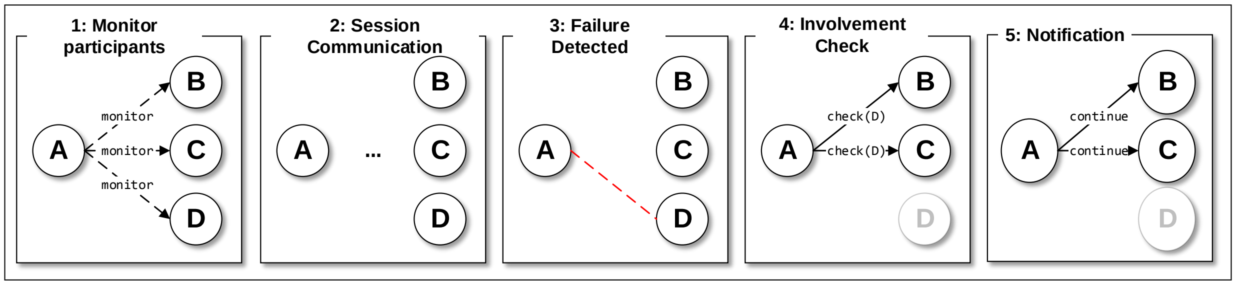


Figure 9: Push-Based Failure Detection

Once all communication has finished an actor may end the session, which invokes `ssactor_conversation_ended` in all participant sessions. Alternatively, `ssactor_conversation_error` is invoked if an error occurs and the session cannot continue (for example, an actor that is required in the session terminates).

4 Failure Detection and Handling

A common assumption for implementations of either session-typed languages, or monitoring frameworks for applications using session types, is that processes persist throughout the course of the session.

Unfortunately, this assumption does not hold true in Erlang applications. An important design pattern in Erlang applications is to arrange processes in *supervision hierarchies*, allowing processes to fail and be restarted by their supervisors when they encounter an unrecoverable fault. Consequently, it is not possible to assume that a process is running throughout the entirety of the session. In this section, we detail how failures within a session can be detected, the circumstances in which a session can continue in spite of the termination of a participant, and a modular method based on *subsessions* to facilitate error handling.

To do so, we rely on two tools provided by Erlang: firstly, Erlang provides the possibility to emulate *synchronous calls*, with a call returning an error or timing out if a remote actor has terminated or is unreachable; and secondly, Erlang provides the possibility to register for reliable *notifications* should a process terminate, known as *monitoring* processes. The latter method, not to be confused with monitoring the communication between actors, is reliable both in the case of a single node, and in the case that a remote node becomes unreachable.

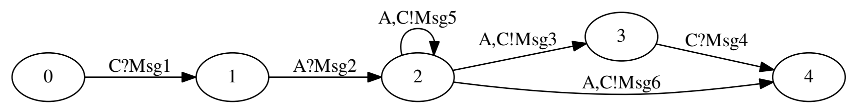
4.1 Failure Detection

Should a process in the session fail, the failure should be detected, as it may be the case that the process which has failed is playing a role which is involved in the remainder of the session.

To this end, we describe two methods of failure detection: push-based, which involves using the Erlang `monitor` functionality to detect when a participant is no longer available, and pull-based, which uses reliable sends and a two-phase commit protocol.

Push-Based Push-based failure detection uses Erlang's reliable termination detection functionality to notify other participants in the session that an unrecoverable failure has occurred. Figure 9 shows the main stages of the push-based failure detection system: firstly, the co-ordinator process for a session monitors each participant in the session to be notified if any of the processes terminate. Should a failure in any of the processes be detected, then the co-ordinator process sends a request to each other participant of the session to determine whether, from the point of view of each participant, the terminated actor is involved in the remainder of the session. More specifically, a role *r* is *involved* in a session if there exists a transition reachable from the current monitor state where *r* is the sender or receiver in a communication.

After the involvement check is complete, then participants are notified of the result. Should all actors respond that the terminated participant is not involved in the remainder of the session, then the session may continue as before; if the terminated participant is involved in the remainder of the session or any of the other actors in the session are unreachable, however, then it is not possible to continue, and the remaining participants are notified of the session's termination.



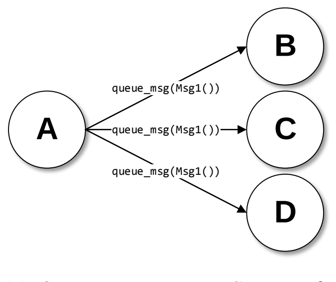
(a) Example Monitor

State	Reachable Roles
0	[A, C]
1	[A, C]
2	[A, C]
3	[C]
4	[]

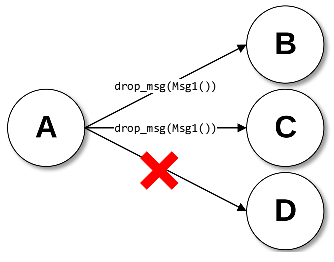
(b) Reachability table

Figure 10: Monitor Reachability Analysis

Push-based failure detection includes a check to see whether a given role is involved in the remainder of the session. Figure 10a shows a monitor and a table detailing the roles reachable at each state[3]; the reachability algorithm only needs to be run once, upon monitor generation, and also records the IDs of any nested FSMs used in order to detect roles inside `par` blocks. Checking whether a role is involved at the current point in the session is achieved by a lookup of the current state ID.



(a) Queue Messages: Successful



(b) Queue Messages: Failure

Figure 11: Pull-based failure detection

Pull-Based As Scribble protocols allow messages to be sent to multiple participants, it is desirable to ensure that messages are only delivered if all processes receiving the message are active. To do so, pull-based failure detection uses a two-phase commit to ensure that all recipient processes are available, and that all recipient monitors accept the message.

The first stage of pull-based failure detection is to send a synchronous message, `queue_msg`, to each recipient monitor process (Figure 11a). Either the call will succeed, returning `ok`, indicating that the call was successful and the message was accepted by the remote monitor; the call will succeed, but returning `error`, indicating that the remote process was available but the remote monitor rejected the incoming message; or the call will fail. When a message is queued, it is assigned a unique identifier. Should a message be accepted, it is stored in a table; should all messages be delivered successfully, a second, asynchronous message will be sent to *commit* the message.

If the a queue message fails for any participant however (Figure 11b), then the message cannot be delivered successfully. If the failure is due to a message rejection, then it is possible for the session to continue: a `drop` message is sent to all participants, the messages are discarded from the queue, and the failure is synchronously reported to the sender. The session cannot continue if a process is unreachable.

Discussion Push-based approaches allow failures to be detected as soon as they occur, and allow the sessions to continue should the failed role not be involved in the remainder of the session. Pull-based approaches only report failures when a failed role is needed, but do not require co-ordination amongst processes to detect whether it is safe to continue. On the other hand, however, pull-based detection

[3]Some readers may recognise this monitor as that of buyer 2 in the two-buyer protocol [14].

approaches fall short when an actor terminates while processing a message; consider a protocol `X() from A to B, C; Y() from B to A, C;`.

Suppose `X` is delivered successfully, but B terminates prior to sending `Y` to `A`: in this case, there would be no way of detecting the failure. Such a situation can be detected using push-based detection. Pull-based detection also falls short if a process terminates between queueing and committing a message. Push-based failure detection falls short should a message handler involving the failed role be executed while the safety check is in progress. Suppose `A` terminates while `B` processes message `X`. Without pull-based detection, there no guarantee that the failure will be detected before `Y` is sent to A and C, with only C receiving the message. Consequently, it is useful to use both methods of failure detection together to ensure that failures are eventually detected (using push-based detection) and that they are detected should the process fail before the safety check is complete (using pull-based detection).

4.2 Subsessions for Exception Handling

In unpublished work[4], Neykova and Yoshida apply *subsessions* to dynamically introduce actors into roles, in particular demonstrating the technique using a Fibonacci benchmark. The session actor framework requires all roles in a protocol to be fufilled upon session initiation, but due to the common Erlang practice of storing process IDs in registries and passing them in messages, it is often the case that it is not known which actor should fulfil which role until later in the protocol. As an example, consider again the chat server: a user sends a room name to the room registry which, if the room exists, responds with the room's process ID. It is only at this point that we know which actor should fulfil the `ChatRoom` role!

Subsessions are a modular abstraction which allow such a pattern to be encapsulated. Interestingly, at the end of their paper on nested protocols, Demangeon and Honda [9] state:

> *Yet exceptions are absolutely necessary when specifying real-world protocols. We believe that nested protocols give a simple way to handle exceptions, by making explicit blocks of computation.*

This is especially poignant given a setting where actors can terminate in the middle of a session: by splitting protocols into subprotocols, it is possible to repeat parts of a session with possibly-different participants should a participant in a subsession terminate.

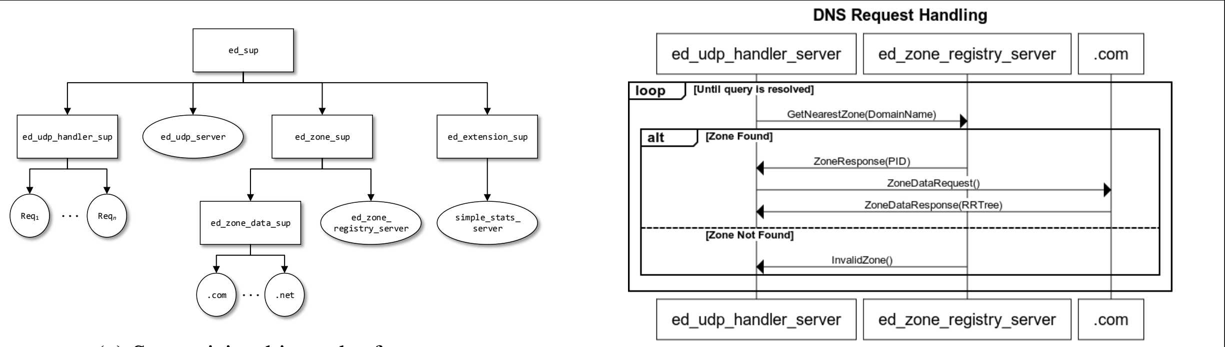
Demangeon and Honda also describe a method by which subsessions may return results, which we can adapt to implement a simple failure handling mechanism. We allow participants in a subsession to state that the subsession has failed through the `conversation:subsession_failed` function, which takes an argument stating the failure. We can then introduce the `initiates` construct:

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

This construct states that role `InitiatorRole` initiates an instance of the `ProtocolName` protocol as a subsession. If the subsession completes successfully (that is, if the session finishes and an actor calls `conversation:subsession_complete`), then the protocol proceeds as `SuccessBlock`. If the process calls `conversation:subsession_failed` with the argument `FailureName` however, then the protocol proceeds as `FailureBlock`. A process can have an unlimited number of `handle` clauses. If no failures are expected, then `initiates` can be used without an explicit `SuccessBlock`.

The `initiates` construct indicates that the session initiator initiates the session, and makes a choice based on the result. To every role other than the initiator, the construct is simply projected as a choice directed by the initiator role: safety follows from enforcing the same restrictions on `SuccessBlock`

[4] http://www.doc.ic.ac.uk/~rn710/sactor/main.pdf



(a) Supervision hierarchy for `erlang-dns`

(b) Messages sent to fulfil a DNS lookup request

Figure 12: DNS Server Case Study

and each `FailureBlock` as in Scribble choice blocks. We provide a distinguished reason for failure, `ParticipantOffline`, which is returned should a session be aborted due to failure detection.

```
ClientThread initiates ChatSession(ClientThread,
    new ChatRoom) {
  clientLeftRoom() from ClientThread to Logger;
  continue ClientChoiceLoop;
} handle(Kicked) {
  clientKicked() from ClientThread to Logger;
  continue ClientChoiceLoop;
} handle(ParticipantOffline) {
  roomTerminated() from ClientThread to Logger;
  continue ClientChoiceLoop;
}
```

As an example, let us introduce a `Logger` process to the `ChatServer` protocol, and state that a client thread should send a message to the logger with the reason for leaving. We introduce two `handle` clauses: if the subsession ends with `Kicked`, then a moderator has expelled the client from the room, whereas if the subsession ends with `ParticipantOffline` is called, then the actor playing the `ChatRoom` role has terminated. In both cases, the appropriate messages are sent to the logger, and the client is free to join another room.

5 Evaluation

5.1 DNS Server Case Study

The `erlang-dns` project[5] is an Erlang/OTP server for the Domain Name System (DNS). Figure 12a shows the supervision hierarchy of the server: of interest to protocol are zone data servers (shown as '.com' and '.net'), which map domain names to IP addresses; `ed_zone_data_server`, which maps domain names to zone data servers; and UDP handler servers (shown in the diagram as $\texttt{Req}_i$), which handle requests.

Upon system initiation, the `ed_udp_server` process opens a UDP acceptor socket and listens for incoming requests. When a query is received, an `ed_udp_handler_server` process is spawned to handle the request, and it is at this point that the session is started. Figure 12b shows the messages sent when fulfilling a DNS request: the UDP handler server contacts the zone registry to ascertain whether or not the zone exists. If not, the server returns `InvalidZone`, at which point a DNS response packet to this effect is generated and sent back to the client. If the zone is found, the zone registry returns the PID of the zone server, which is queried for information about the zone. At this point, if the IP address can be resolved from the request, then it is returned to the user, but it may also be necessary to perform a recursive lookup.

Protocol Figure 13 shows the Scribble protocols for `erlang-dns`. There are three roles: `UDPHandlerServer`, fulfilled by the `ed_udp_handler_server` instance initiating the session; `DNSZoneRegServer`, fulfilled by `ed_zone_registry_server`; and `DNSZoneDataServer`, fulfilled by a zone data server able to handle the request. It is not possible to fulfil the `DNSZoneDataServer` role upon session initiation as the actor to invite depends on the result of the request to the zone registry; consequently, we have a main protocol `HandleDNSRequest`,

[5] https://www.github.com/hcvst/erlang-dns

```
global protocol HandleDNSRequest(role UDPHandlerServer, role DNSZoneRegServer) {
  rec QueryResolution {
    FindNearestZone(DomainName)
      from UDPHandlerServer to DNSZoneRegServer;
    choice at DNSZoneRegServer {
      ZoneResponse(ZonePID) from DNSZoneRegServer to UDPHandlerServer;
      UDPHandlerServer initiates GetZoneData(
        UDPHandlerServer, new DNSZoneDataServer);
      continue QueryResolution;
    } or { InvalidZone() from DNSZoneRegServer to UDPHandlerServer; }
  }
}

global protocol GetZoneData(role UDPHandlerServer, role DNSZoneDataServer) {
  ZoneDataRequest() from UDPHandlerServer to DNSZoneDataServer;
  ZoneDataResponse(RRTree) from DNSZoneDataServer to UDPHandlerServer;
}
```

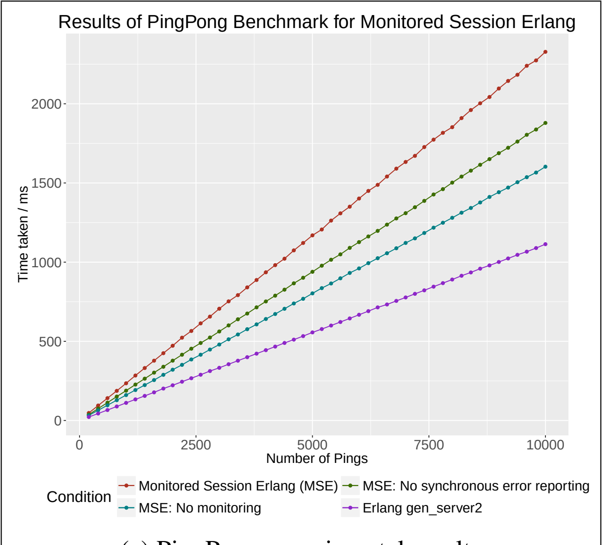
```
config() ->
  [{ed_zone_data_server, [{"GetZoneData", ["DNSZoneDataServer"]}]},
   {ed_zone_registry_server, [{"HandleDNSRequest", ["DNSZoneRegServer"]}]},
   {ed_udp_handler_server, [{"HandleDNSRequest", ["UDPHandlerServer"]}, {"GetZoneData", ["UDPHandlerServer"]}]}].
```

Figure 13: Scribble Protocol and Configuration File for `erlang-dns`

and a subprotocol `GetZoneData` which is initiated should `ed_zone_registry_server` respond with a PID.

Implementation To adapt `erlang-dns` to use `monitored-session-erlang`, we firstly define a configuration file to associate each actor with the role that it plays in each protocol. Next, we adapt each of the actors which were formerly instances of `gen_server` to be instances of `ssa_gen_server`. Vitally, *no changes are made to the supervision structure*, as it is *orthogonal* to the monitoring of messages.

There are two main ways that the session implementation diverges from the original implementation. Firstly, once the zone data server PID has been returned, we start a new subsession to invite the zone data server and retrieve the zone data. Secondly, we emulate a synchronous call with two asynchronous messages. As DNS is most commonly implemented over UDP, there is no guarantee that a response will ever be received. Should a failure occur within the DNS server, there is little point in trying to fulfil the remainder of a request. Instead, it is better to allow the supervisor to restart the component and let the request time out.



(a) PingPong experimental results

Figure 14: Experimental evaluation of overheads

5.2 Overheads

We measure the overheads of `monitored-session-erlang` using the `PingPong` benchmark: an actor *A* sends a message `ping` to an actor *B*, which responds with a message `pong`. We measure several scenarios: “Erlang `gen_server2`” refers to an implementation not using `monitored-session-erlang`, where actors communicate using the `gen_server:cast` function. The remaining three scenarios use `monitored-session-erlang`: “MSE” refers to an implementation using the full system; “MSE: No synchronous error reporting” refers to an implementation without synchronous reporting of monitoring errors, and “MSE: No monitoring” refers to an implementation without monitoring.

The different scenarios demonstrate different aspects of the system: in contrast to the original work on session actors, sending a message involves synchronous calls to both the source and destination monitors to immediately report failures.

Should either of the checks fail, an exception is raised, allowing the computation to be aborted as soon as a message is rejected by a monitor. Employing an asynchronous approach results in two fewer messages, but errors have to be reported as separate messages, meaning that the remainder of the handler must run before an error report can be processed. The final variation uses `monitored-session-erlang` with monitoring disabled: the overheads incurred in this scenario are as a result of the external monitor process and the resolution of role names to actors.

Figure 14a shows the experimental results of the four basic experimental scenarios[6]. The `gen_server2` implementation is fastest at 0.111ms per iteration, whereas the full `monitored-session-erlang` system has a mean time per iteration of 0.23ms, giving a final overhead per iteration of 0.12ms (or 0.06ms per messsage). The overheads can be explained by the additional messages sent between the monitors in order to detect and report monitoring errors.

6 Related Work

Session types were originally introduced by Honda [12] and later expanded upon by Honda et al. [13] to model interactions between two communicating parties. Honda et al. [14] propose multiparty session types to model interactions with more than two participants. A *global type* is projected to a *local type* for each participant. Conformance to local types can be checked statically, or monitored at runtime.

Deniélou and Yoshida [10] discuss the connections between multiparty session types and communicating finite state machines [4]: we use a variant of this algorithm in the monitor generation phase. Chen et al. [6] and Bocchi et al. [3] formalise the theory of runtime monitoring, with monitors and routing tables as first class entities, and reductions of monitored processes predicated on labels emitted by a labelled transition system on local types. The SPY framework [18] can monitor communication against multiparty session types in Python, and runtime monitoring can also be used to enforce timing constraints [19].

Capecchi et al. [5] describe a process calculus with multiparty session types containing `try..catch` and `throw` constructs, where a set of roles move to a compensation process should an exception be thrown. Chen et al. [7] describe a formal system of *protocol types* based on multiparty session types, where each interaction is annotated with the exceptions that may occur as a result of the interaction, and continuations which are invoked upon a failure occurring. The system is realised by a *transformation* stage which combines projection with an analysis of participants which need to be notified should an exception occur. The formalism is elegant and a particular strength is its decentralised nature. In our setting, the requirement to satisfy the protocol well-formedness conditions for choice blocks means that such notifications must be encoded explicitly in the protocol. In contrast, our approach of using subsessions to structure protocols allows parts of the protocol to be retried with different participants if one terminates or goes offline.

Mostrous and Vasconcelos [16] consider a core session calculus based on Erlang, using Erlang's ability to generate fresh references to design a static type discipline relying on *correlation sets* [20] to associate messages with sessions. The type system has not yet been implemented. Crafa [8] defines a behaviourally-typed actor caclulus, AC, based on Scala's actor primitives, guaranteeing that each input is eventually matched with exactly one output in the system. We expand upon the work of Neykova and Yoshida [17] in monitoring actors according to multiparty session types by investigating how the session actor framework can be applied in the setting of an actor-based functional language without relying on AMQP, and propose methods of handling the case where an actor terminates during a session.

[6]Experimental conditions: Two cluster nodes with 4 16-core AMD Opteron 6376 processors at 2300MHz; 264GB RAM; RTT 0.101ms using `ping`. Scientific Linux 7, Erlang 7.0. Value plotted: arithmetic mean over 100 repetitions, measured after session establishment.

7 Conclusion

Communication is central to software written in actor-based languages. We have described the design and implementation of `monitored-session-erlang`, a framework that allows communication in Erlang applications to be monitored against multiparty session types. Our tool demonstrates the applicability of the conceptual framework of multiparty session actors to actor-based functional languages, motivates subsessions as a fundamental abstraction, and introduces features such as synchronous error reporting and allowing actors to take part in multiple instances of a protocol.

Future work will centre around a formal semantics, and an investigation into how ideas from the dynamic view of session-typed actors can be used in a session-typed concurrent λ-calculus.

Acknowledgements Thanks to Sam Lindley, Garrett Morris, and Philip Wadler for useful discussions, and to the anonymous reviewers for detailed and insightful comments. This work was supported by EPSRC grant EP/L01503X/1 (University of Edinburgh CDT in Pervasive Parallelism).

References

[1] G. A. Agha. Actors: A model of concurrent computation in distributed systems. Technical report, DTIC Document, 1985.

[2] J. Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, 2010. doi: 10.1145/1810891.1810910.

[3] L. Bocchi, T.-C. Chen, R. Demangeon, K. Honda, and N. Yoshida. Monitoring networks through multiparty session types. In *FORTE*, pages 50–65. Springer, 2013. doi: 10.1007/978-3-642-38592-6_5.

[4] D. Brand and P. Zafiropulo. On communicating finite-state machines. *JACM*, 30(2):323–342, 1983. doi: 10.1145/322374.322380.

[5] S. Capecchi, E. Giachino, and N. Yoshida. Global escape in multiparty sessions. *MSCS*, 26(02):156–205, 2016. doi: 10.1017/S0960129514000164.

[6] T.-C. Chen, L. Bocchi, P.-M. Deniélou, K. Honda, and N. Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *TGC*, pages 25–45. Springer, 2011. doi: 10.1007/978-3-642-30065-3_2.

[7] T.-C. Chen, M. Viering, A. Bejleri, L. Ziarek, and P. Eugster. A Type Theory for Robust Failure Handling in Distributed Systems. In *FORTE*, pages 96–113. Springer, 2016. doi: 10.1007/978-3-319-39570-8_7.

[8] S. Crafa. Behavioural Types for Actor Systems. *CoRR*, abs/1206.1687, 2012. URL http://arxiv.org/abs/1206.1687.

[9] R. Demangeon and K. Honda. Nested protocols in session types. In *CONCUR*, pages 272–286. Springer, 2012. doi: 10.1007/978-3-642-32940-1_20.

[10] P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP*, pages 194–213. Springer, 2012. doi: 10.1007/978-3-642-28869-2_10.

[11] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[12] K. Honda. Types for dyadic interaction. In *CONCUR*, pages 509–523. Springer, 1993. doi: 10.1007/3-540-57208-2_35.

[13] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, pages 122–138. Springer, 1998. doi: 10.1007/BFb0053567.

[14] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL*, pages 273–284, New York, NY, USA, 2008. ACM. doi: 10.1145/1328438.1328472.

[15] R. Hu, R. Neykova, N. Yoshida, R. Demangeon, and K. Honda. Practical interruptible conversations. In *RV*, pages 130–148. Springer, 2013. doi: 10.1007/978-3-642-40787-1_8.

[16] D. Mostrous and V. T. Vasconcelos. Session typing for a featherweight Erlang. In *COORDINATION*, pages 95–109. Springer, 2011. doi: 10.1007/978-3-642-21464-6_7.

[17] R. Neykova and N. Yoshida. Multiparty session actors. In *COORDINATION*, pages 131–146. Springer, 2014. doi: 10.1007/978-3-662-43376-8_9.

[18] R. Neykova, N. Yoshida, and R. Hu. SPY: Local verification of global protocols. In *RV*, pages 358–363. Springer, 2013. doi: 10.1007/978-3-642-40787-1_25.

[19] R. Neykova, L. Bocchi, and N. Yoshida. Timed runtime monitoring for multiparty conversations. In *BEAT*. Open Publishing Association, 2014. doi: 10.4204/EPTCS.162.3.

[20] M. Viroli. Towards a formal foundation to orchestration languages. *ENTCS*, 105:51–71, 2004. doi: 10.1016/j.entcs.2004.05.008.

[21] N. Yoshida, R. Hu, R. Neykova, and N. Ng. The Scribble protocol language. In *TGC*, pages 22–41. Springer, 2013. doi: 10.1007/978-3-319-05119-2_3.