Monitoring Erlang/OTP Applications using Multiparty Session Types

Simon Fowler

Master of Science by Research
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2015

Abstract

The actor model has emerged as a programming paradigm particularly suited to programming concurrent and distributed systems. Programming languages based on the actor model consist of lightweight processes which *do not communicate using shared memory*, relying instead on explicit message passing. Consequently, programming languages built on the actor model avoid many of the pitfalls associated with shared memory systems, and can employ design patterns such as supervision hierarchies to remain operational even in the presence of software or network errors.

In actor-based systems, communication is key: but how can we verify that applications conform to their expected communication patterns? In this thesis, we investigate the use of *session types*—a type discipline used for codifying communication protocols—to describe and verify conformance to communication patterns in the actor-based Erlang programming language.

Drawing upon connections between multiparty session types and communicating finite-state automata [29], and the work on session actors by Neykova and Yoshida [52], we introduce a framework for creating Erlang/OTP server applications where messages between processes can be checked dynamically against session types. We discuss issues of failure detection and failure handling within a session, and describe extensions to the standard presentation of multiparty session to show how common communication patterns found within Erlang/OTP applications, such as passing process IDs and making synchronous calls, may be described by session types.

We evaluate the system both empirically and through the implementation of a series of case studies: a travel booking workflow involving error handling; the adaptation of a freely-available Erlang DNS server, and a chat server.

Acknowledgements

I would like to begin by thanking Philip Wadler and Sam Lindley for their supervision, and for their numerous useful suggestions throughout the project.

A great deal of thanks is due to Garrett Morris, for detailed comments on an earlier draft of this work. I would also like to thank Bob Atkey: a discussion at the CoCo:PoPS workshop inspired me to look more in depth at failure handling strategies, which led to the approach described in this thesis.

Furthermore, I thank Edwin Brady and Sam Lindley for encouraging me to attend the SICSA Summer School on Practical Types, in spite of its proximity to the thesis deadline.

My thanks also go to Murray Cole, Amanda Grimm, and everyone involved in the running of the CDT: your tireless efforts have been much appreciated. My colleagues in IF1.07 have also provided much camaraderie and inspiration.

Finally, I would like to thank my parents, Clare and Andy Fowler, for their boundless encouragement throughout the course of the project.

I gratefully acknowledge support from the EPSRC Centre for Doctoral Training in Pervasive Parallelism.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Simon Fowler*)

Table of Contents

Chapter 1

Introduction

Writing concurrent and distributed software is difficult.

Cliché as it may now be to say it, we live in an era where it is no longer sufficient to write single-threaded applications. Physical limits on processors require that performance advances are attained not by adding more transistors or running at a higher clock frequency, but instead by adding more processors.

Writing a multithreaded application involves thinking about a multitude of issues such as race conditions and deadlocks, which simply do not occur when writing single-threaded software. Care has to be taken to identify the appropriate critical sections, the correct concurrency abstractions to use, to ensure that all locks must be released and that there are no circular requests for locks: the list continues.

And that is just for concurrent systems! The crucial introduction of distribution brings yet more challenges to the fore: instead of shared memory accesses, data may not be on the same machine, and accessing the data therefore requires communication. And perhaps most crucially, failures are not an exceptional case: in large systems, they must be expected, and handled. It is easy to imagine how a developer may quickly be overwhelmed with low-level details instead of concentrating on business logic.

1.1 Erlang and the Actor Model

Fortunately, the Erlang programming language [5] has been developed with distribution and failure handling as a primary concern. Erlang is a dynamically-typed functional programming language based on the actor model [3]. The actor model was originally designed as a model of concurrency consisting of single-threaded lightweight processes, or *actors*. Upon receipt of an

incoming message, actors can perform three actions: send a finite number of messages to other actors, spawn a finite number of new actors, and change how to react to subsequent messages. Importantly, the actor model forbids co-ordination through shared memory: all sharing must happen using explicit message passing.

At first glance, this style of programming seems cumbersome and inefficient. After all, it is a substantial departure from the standard style of programming, and message passing will naturally incur some overheads not present when simply reading from shared memory—but it is vital not to overlook the benefits that adopting this programming style will yield.

Since actors are single-threaded, and there is no shared memory, locks are no longer necessary: atomicity is gained for free. Relying purely on communication also makes it easier to structure applications: instead of needing to consider both the case where data is local and the data is remote, data accesses can be handled transparently by the runtime system, without needing to incorporate different programming styles.

1.1.1 Erlang and Fault Tolerance

But Erlang's real strength comes with its failure handling and fault tolerance mechanisms. In distributed systems, failures truly are inevitable. Even classifying failures is difficult: is a node not responding because it is offline semi-permanently, for example due to a power cut, or is it simply taking too long to respond due to increased load?

Built with highly-reliable distributed applications in mind, Erlang encourages certain patterns of development. In particular, the most well-known of these is somewhat unintuitive: if a process encounters a problem for any reason, *let it fail*. When designing Erlang applications, processes are arranged in structures known as *supervision hierarchies*. In this model, processes are either *workers*, which perform computations, or *supervisors*, which are responsible for restarting workers should they fail. Designing applications using supervision hierarchies fosters the design and implementation of highly-resilient and reliable applications: several real-world case studies may be found in Armstrong's PhD thesis [4].

1.2 The Challenge: Safe Communication in Erlang Applications?

When moving to the actor model, communication becomes a central actor of the system.

The supervisor pattern allows processes to terminate when errors occur in the logic of the program: a process terminates, the failure is logged, and a developer can fix the underlying cause of the fault at a later date. At the same time, it is difficult to check when errors occur in

the *communication patterns* described by the program. In spite of communication being central to Erlang applications, communication patterns are often informally described, and the order in which messages are sent is not always checked by program logic.

This brings us to a question: is there a way of applying the same 'let it fail' methodology to violations in communication patterns?

1.2.1 Session Types

One possible solution to this challenge is the use of *session types* [36, 37]. Just as the now ubiquitous notion of a data type codifies the expected structure of data, resulting in memory safety, optimisations, and better application structure, session types are a type discipline which can ensure that applications conform to communication protocols.

As an example, consider the example of a simple calculator server, offering two operations: addition and negation [31]. The & symbol denotes a type representing the offering of a choice to the client, ? denotes a type representing receiving a value, and ! denotes a type representing sending a value.

$$S = \&\{\mathsf{Add} : ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Int},\ \mathsf{Neg} : ?\mathsf{Int}.!\mathsf{Int}\}$$

A client would implement the *dual* session type:

$$C = \oplus\{\mathsf{Add} : !\mathsf{Int}.!\mathsf{Int}.?\mathsf{Int},\ \mathsf{Neg} : !\mathsf{Int}.?\mathsf{Int}\}$$

Note here that all receive types have been replaced by send types, and the branching type & has been replaced by the choice type $\oplus$. In fact, S and C are dual: duality ensures that communication between the two is safe.

Decades of research has spawned many advances in the theory and practice of session types. In particular, Honda et al. [38] extend binary session types to the *multiparty* setting, where multiple participants may take part in a session.

Multiparty session types provide strong guarantees on communication safety. In particular, by allowing *local types* to be projected from a *global* view of the system, multiparty session

types ensure that protocols are free of race conditions (via analysis of causality), and deadlocks. Local types can then be used to either statically or dynamically check conformance of an implementation to a protocol.

Alas, the integration of multiparty session types with Erlang applications is nontrivial. For example, multiparty session types make assumptions that all participants in a session are available at the beginning of the session, and are alive until the conclusion of the session: assumptions which cannot be made due to the ability to pass process IDs between actors, allowing participants to be introduced midway through the session, and as participants may fail and be restarted midway through the session.

The question we ask in this thesis is therefore:

> *Can multiparty session types be used to encode communication patterns in distributed Erlang/OTP applications, and what benefits are there if they can?*

1.3 Contributions

We answer the above question in the affirmative, requiring some extensions to the standard presentation of multiparty session types.

Concretely, the contributions of this thesis are as follows.

- Drawing inspiration from the session actor framework of Neykova and Yoshida [52], we describe the design and implementation of a system, `monitored-session-erlang`, for ensuring that messages in Erlang/OTP applications conform to protocols specified by session types.

  We make several substantial changes to the original session actor framework which is presented as a Python framework using AMQP, primarily due to the fact that Erlang is an actor-based language. By not requiring a middleware layer, we substantially simplify the mechanism by which actors may be invited to fulfil roles, and allow fully-distributed communication between actors. Finally, we change the way by which monitor violations are reported, allowing immediate reporting of failures.

- We identify common Erlang/OTP communication patterns that are not possible to describe using the standard theory of multiparty session types. In particular, we note the common pattern of passing process IDs to dynamically introduce processes into a communication session, and motivate the use of *subsessions* [28]—child sessions spawned by a parent session—as a solution. We also show the importance of encoding synchronous calls, reducing the need to transform actor programs into an asynchronous form, and

statically ruling out cyclic calls.

- Erlang applications are developed in a style where lightweight processes may terminate upon encountering an error. Consequently, processes cannot be assumed to be alive throughout the duration of a session, and sessions must incorporate failure detection and failure handling strategies.

  We describe and discuss two methods for failure detection: *pull-based*, using a two-phase commit protocol, and *push-based*, using Erlang's `monitor` functionality and FSM reachability analysis.

  We also describe a method for handling both application-level exceptions, and failures due to processes terminating, based on subsessions. We demonstrate the technique by implementing a travel booking case study with error handling, specified for the Business Process Model and Notation[1] (BPMN).

- We evaluate the applicability of the framework by investigating two larger case studies: adapting a freely-available Erlang DNS server to use session types, and the implementation of a chat server.

- We empirically investigate the performance of the Erlang session actor framework, showing that overheads, while present, are acceptable.

1.4 Structure of the Thesis

The thesis is structured as follows.

In Chapters 2 and 3, we begin by examining the background and related work of both session types and Erlang applications. In particular, we examine the theory of dynamic monitoring of session types in some depth, and comment on the guarantees afforded to programs as a result.

In Chapter 4, we describe the design and implementation of a library, `monitored-session-erlang`, which allows actor communications to be monitored using multiparty session types. We describe the similarities to, and differences from, existing work; we discuss Erlang-specific implementation concerns such as the integration with the different layers of communication semantics; we outline the structure of the monitor generation and monitor runtime code, and describe the API exposed to the user.

In Chapter 5, we describe two common Erlang communication patterns which cannot easily be expressed using multiparty session types and the standard session actor approach: process ID

[1] http://www.bpmn.org

passing, and synchronous calls. We describe the two types of communication pattern, motivating the use of subsessions to handle process ID passing, and describe a modification to Scribble and the monitoring framework to enable synchronous calls within a message handler.

In Chapter 6, we discuss the failure detection and failure handling mechanisms needed when we cannot assume that the actors inhabiting each of the roles in a session are available for the duration of the session. We describe two main failure detection mechanisms: pull-based, using a two-phase commit technique, and push-based, using the Erlang `monitor` function along with monitor reachability analysis. Furthermore, we describe a method based on subsessions to allow failures to be handled and compensated for, showing how the result of a subsession may be used to decide how a parent process should proceed.

In Chapter 7, we describe the evaluation of the system. We show how multiparty session types can be used to encode and monitor conformance to the communication patterns within a publicly-available DNS server, and a chat server. We subsequently detail an empirical evaluation of the overheads imposed by various aspects of the system.

Chapter 8 concludes, and describes directions for future work.

Chapter 2

Background

2.1 Process Calculi

Process calculi are abstract theoretical models used to describe the behaviour of concurrent processes. Session types were designed as a typing discipline for process calculi based on the π-calculus [48]. Building on previous work on CSP [35] and CCS [47], the π-calculus is a small process calculus based around the idea of *mobility*: the idea that *links* between processes may be sent between processes. These links, known as *names*, form the basic entity of the π-calculus: processes use names to interact, and names may be freely sent between processes.

2.2 Session Types

Session types were originally proposed by Honda [36], and later introduced alongside language primitives [37] to model a series of interactions between two communicating parties.

Session types are named after the notion of a *session*, which is defined as a series of interactions between two communicating parties, with these interactions taking place over a shared, private channel. Interaction patterns are specified as *session types*, which are inhabited by communication primitives. The original work identifies three such primitives: *value passing*, which involves sending and receiving values; *channel delegation*, which involves sending and receiving channels; and *label branching*, which introduces the possibility of flow control by offering multiple possible continuations of a session.

Session type disciplines ensure that the communication along a session channel is safe: where one process expects to send, the other expects to receive. Safety relies crucially on the notion of *duality* – for example, the dual of a send is a receive, and the dual of offering a selection is

P	::=		Processes
		$\bar{x}v.P$	Output
	|	$x(x).P$	Input
	|	$P \mid P$	Parallel Composition
	|	if v then P else P	Conditional Statement
	|	$\mathbf{0}$	Inaction
	|	(νxx)	Scope Restriction
v	::=		Values
		x	Variable
	|	true	
	|	false	

Figure 2.1: Syntax of processes

selecting a branch.

Figures 2.1 —2.5 describe a process calculus and type system for session-typed processes, described by Vasconcelos [61][1]. Processes are similar to that of the π-calculus, providing the ability to send and receive values on channels, along with standard parallel composition, conditional statements, and inaction processes. Interestingly, however, scope restriction takes the form νxx, representing two dual channels, as opposed to a single channel. The type system includes value types which can be qualified as linear, meaning that they must be used exactly once, or unlimited, meaning that they may be used an unlimited number of times. Linearity is important as channels *must not be duplicated*, as the duplication of channels will result in the loss of safety guarantees.

Linearity is enforced through the use of context split and context update operations: the context split operation $\Gamma = \Gamma_1 \circ \Gamma_2$ non-deterministically splits a context Γ into two separate contexts Γ_1 and Γ_2, where the sets of values with linear types in Γ_1 and Γ_2 are pairwise distinct. The context update operation $\Gamma + x : T = \Gamma$ ensures that a value x with a linear type T may only be added to a context Γ if x does not already exist in Γ.

Structural congruence and the reduction rules are largely standard, but reflect the fact that restriction is over two dual channel endpoints x and y instead of a single channel type. The [T-RES] rule ensures that two channel endpoints x and y are dual, and the [T-PAR] rule ensures that no values with a linear type appear in parallel processes. The [T-IN] rule splits the context into two contexts Γ_1 and Γ_2, checking the channel type in Γ_1 and the continuation in Γ_2, ensuring

[1] Note that this is not the original system described by Honda et al. [37], but is a later, more concise presentation, shown here for simplicity.

$$
\begin{array}{lcll}
q & ::= & & \text{Qualifiers} \\
 & | & \mathsf{lin} & \text{Linear} \\
 & | & \mathsf{un} & \text{Unrestricted} \\
p & ::= & & \text{Prefixes} \\
 & | & ?T.T & \text{Receive} \\
 & | & !T.T & \text{Send} \\
T & ::= & & \\
 & | & \mathsf{bool} & \text{Boolean} \\
 & | & \mathsf{end} & \text{Termination} \\
 & | & qp & \text{Qualified Prefix} \\
\Gamma & ::= & \emptyset \mid \Gamma, x : T & \text{Contexts}
\end{array}
$$

$$\overline{q?T.U} = q!T.\overline{U}$$

$$\overline{q!T.U} = q?T.\overline{U}$$

$$\overline{\mathsf{end}} = \mathsf{end}$$

Figure 2.2: Syntax of, and duality relation on, types

Structural Congruence, $P \equiv P$

$$P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad P \mid \mathbf{0} \equiv P$$

$$(\nu xy)P \mid Q \equiv (\nu xy)(P \mid Q) \qquad (\nu xy)\mathbf{0} \equiv \mathbf{0} \qquad (\nu wx)(\nu yz)P \equiv (\nu yz)(\nu wx)P$$

Reduction Rules, $P \longrightarrow P'$

R-Com

$$(\nu xy)(\bar{x}v.P \mid y(z).Q \mid R) \longrightarrow (\nu xy)(P \mid Q[v/z] \mid R)$$

R-IfT

$$\mathsf{if\ true\ then}\ P\ \mathsf{else}\ Q \longrightarrow P$$

R-IfF

$$\mathsf{if\ false\ then}\ P\ \mathsf{else}\ Q \longrightarrow Q$$

R-Res

$$\frac{P \longrightarrow Q}{(\nu xy)P \longrightarrow (\nu xy)Q}$$

R-Par

$$\frac{P \longrightarrow Q}{P \mid R \longrightarrow Q \mid R}$$

R-Struct

$$\frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}$$

Figure 2.3: Structural Congruences and Reduction Rules for Processes

Context split, $\Gamma = \Gamma \circ \Gamma$

$$\emptyset = \emptyset \circ \emptyset \qquad \frac{\Gamma_1 \circ \Gamma_2 = \Gamma \quad \mathsf{un}(T)}{\Gamma, x : T = (\Gamma_1, x : T) \circ (\Gamma_2, x : T)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : \mathsf{lin}\, p = (\Gamma_1,\ x : \mathsf{lin}\, p) \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma x : \mathsf{lin}\, p = \Gamma_1 \circ (\Gamma_2, x : \mathsf{lin}\, p)}$$

Context update, $\Gamma + x : T = \Gamma$

$$\frac{x : U \notin \Gamma}{\Gamma + x : T = \Gamma, x : T} \qquad \frac{\mathsf{un}(T)}{(\Gamma, x : T) + x : T = (\Gamma, x : T)}$$

Figure 2.4: Operations on contexts

Typing rules for values, $\Gamma \vdash v : T$

$$\text{T-True}\ \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \text{T-False}\ \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathsf{false} : \mathsf{bool}} \qquad \text{T-Var}\ \frac{\mathsf{un}(\Gamma)}{\Gamma, x : T \vdash x : T}$$

Typing rules for processes, $\Gamma \vdash P$

$$\text{T-Inact}\ \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathbf{0}} \qquad \text{T-Par}\ \frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash P \mid Q} \qquad \text{T-Res}\ \frac{\Gamma, x : T, y : \overline{T} \vdash P}{\Gamma \vdash (\nu\, x\, y) P} \qquad \text{T-If}\ \frac{\Gamma_1 \vdash v : \mathsf{bool} \quad \Gamma_2 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash \mathsf{if}\ v\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

$$\text{T-In}\ \frac{\Gamma_1 \vdash x : q?T.U \quad (\Gamma_2 + x : U), y : T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x(y).P}$$

$$\text{T-Out}\ \frac{\Gamma_1 \vdash x : q!T.U \quad \Gamma_2 \vdash v : T \quad \Gamma_3 + x : U \vdash P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash \overline{x}[v].P}$$

Figure 2.5: Typing rules for processes

that the channel is typable in the continuation after the input capability has been expended. The case for [T-OUT] is similar.

Later work by Kobayashi [41], advocated and expanded upon by Dardha et al. [27], shows that the π-calculus with session types can be represented in the linear π-calculus [42]—that is, the π-calculus in which each channel must be used exactly once—with variant types. This technique can be used when implementing session-typed languages, or to reason about session-typed languages using a lower-level core calculus.

2.2.1 Multiparty Session Types

Originally, session types were designed to encapsulate a series of interactions between *two* participants. Describing interactions between *multiple* parties using binary session types is far from natural, prone to deadlocks, and cannot capture the interleavings of the session types, losing important information.

Honda et al. [38] describe a session-typed process calculus involving multiple participants, with a global type specifying the sequence of interactions between the participants. Global types can be *projected* to a local type, which is the global type from the perspective of a single participant.

G	$::=$	$p \to \Pi : \langle S \rangle . G'$	Send message
		$\mid p \to \Pi : \langle \{l_i : G_i\}_{i \in I} \rangle$	Send branch selection
		$\mid \mu \mathbf{t}.G$	Recursion
		$\mid \mathbf{t}$	Recursion variable
		$\mid \mathbf{end}$	End
S	$::=$	bool |int |...	Ground types

Figure 2.6: Syntax of global types

Figure 2.6 shows the syntax of global types, as described by Bettini et al. [9][2]. In particular, $p \to \Pi : \langle U \rangle . G'$ denotes a send operation from a participant p to a set of participants contained in Π, with the protocol continuing as G'; and $p \to \Pi : \langle \{l_i : G_i\}_{i \in I} \rangle$ denotes the selection of a branch by p, communicating the choice to all participants Π, with the protocol proceeding as the selected branch.

Multiparty session types work under the assumption of ordered message delivery on a channel, but without any guarantees of ordering between channels. In order to provide this, the system

[2]Note that this is not the syntax of global types given by Honda et al. [38]: in particular, global types are presented without reference to channels. Such a presentation does not decrease expressivity—see [9]. The presentation given by Bettini et al. [9] is the de-facto standard presentation of multiparty session types used in the literature.

requires analysis of causality: for example, the protocol

$$A \to B\langle \mathsf{bool} \rangle.\mathbf{end}$$
$$C \to B\langle \mathsf{bool} \rangle.\mathbf{end}$$

would not be permitted as there is no causal ordering between the receipt of messages m and n by participant B.

Multiparty session types ensure that if all participants satisfy their local types, then the system satisfies the global type. The paper also proves subject reduction and local progress: local progress states that a process will always be able to reduce further (unless it is the inactive process) unless a deadlock has been caused due to interleaving with other protocols.

Bettini et al. [9] and Coppo et al. [25] consider the property of *global progress*. Global progress ensures that protocols remain deadlock-free even in the presence of interleaving amongst multiple multiparty sessions using an *interaction typing system*.

2.2.2 The Scribble Protocol Description Language

Building upon the theoretical foundations of multiparty session types, the Scribble protocol description language is a domain-specific language for describing and validating protocols. Scribble abstracts over concrete participants by making use of the idea of a *role*: an abstract description of a participant's behaviour within a session.

Scribble implements validation and projection functionality: that is, it is possible to ensure that the written protocol forms a valid multiparty session type by checking conditions such as causality, and if so, can project the global types to local types for each role.

The canonical example of multiparty session types is that of the Two-Buyer Protocol, shown in Figure 2.7.

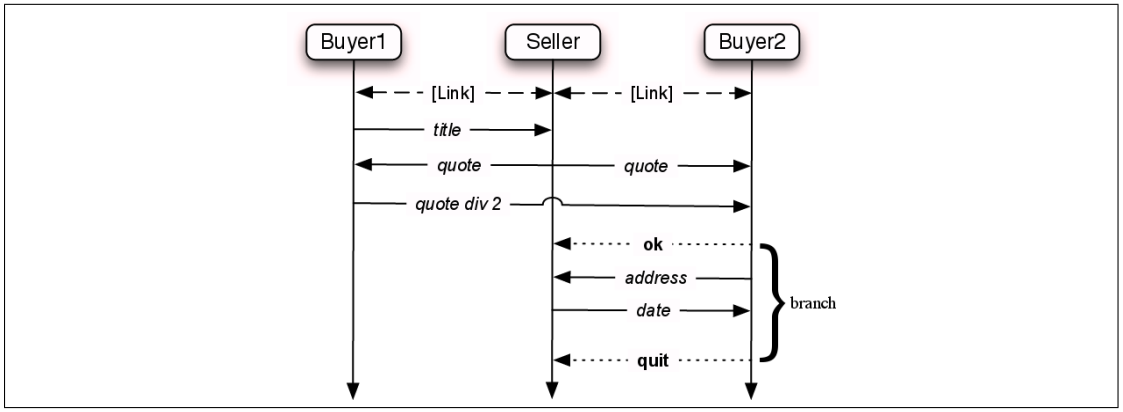


Figure 2.7: Two-Buyer Protocol [38]

The two-buyer protocol is a simplification of a financial protocol. In the two-buyer protocol, there are three roles: two buyers, and a seller. The aim of the protocol is to buy an expensive item, by undertaking the following steps:

- Buyer 1 requests the price of an item from the seller
- The seller sends the price of the item to Buyer 1 and Buyer 2
- Buyer 1 sends Buyer 2 the amount that Buyer 2 should pay
- Buyer 2 can choose to:
  - Accept the offer, at which point it sends a delivery address to the seller and receives a delivery date
  - Reject the offer, and await another offer from Buyer 1
  - End the protocol

In Scribble, this protocol would be written as follows:

```
global protocol TwoBuyer(role Buyer1, role Buyer2, role Seller) {

    title(String) from Buyer1 to Seller;
    price(Currency) from Seller to Buyer1, Buyer2;
    rec loop {
        share(Currency) from Buyer1 to Buyer2;
        choice at Buyer2 {
            accept() from Buyer2 to Buyer1;
            deliveryAddress(String) from Buyer2 to Seller;
            deliveryDate(Date) from Seller to Buyer2;
        } or {
            reject() from Buyer2 to Buyer1;
            continue loop;
        } or {
            quit() from Buyer2 to Buyer1, Seller;
        }
    }
}
```

We begin by defining the protocol header, including the protocol name and the names of all of the roles which take part in the protocol. Roles are assumed to all be associated with endpoints upon initiation of the protocol.

An interaction takes the form `MessageLabel(PayloadTypes) from Sender to Receivers`. A message may only have one message name and one sender, but it may have multiple payload types and multiple receivers. That is, in keeping with the semantics of multiparty session types, Scribble supports multicast. One notable absence, however, is that of channel delegation, since Scribble does not incorporate the notion of a channel.

2.2.3 Connections with Linear Logic

Linear logic is a substructural logic which forbids contraction (where additional propositions may be added to the hypotheses or conclusion of a sequent), and weakening (where two equal members on the same side of a sequent may be replaced by a single member), and has been used within type systems on account of its treatment of *resources*. This lends itself well to the treatment of channels within session-typed process calculi: in his original exposition of linear logic, Girard [33] speculated about the use of linear logic to reason about concurrency. Abramsky [1] and Bellin and Scott [7] both provide insights into how linear logic propositions may be interpreted as processes in process calculi.

Caires and Pfenning [16] provide a session typing system for the π-calculus that corresponds with dual-intuitionistic linear logic , showing that the connectives within linear logic are sufficiently expressive to capture communication primitives present within session typing disciplines for the π-calculus. *Communication* corresponds directly to cut elimination.

2.2.4 Linear Functional Languages

Session-typed languages must necessarily incorporate some form of linearity tracking in order to provide guarantees of session fidelity. Linearity tracking can either be achieved either using a linear type system, or techniques such as parameterised monads [6], as demonstrated by Pucella and Tov [57] and Sackman and Eisenbach [58].

Gay and Vasconcelos [32] describe an asynchronous session-typed functional language based on the linear λ calculus. The language incorporates a fixpoint operator for recursion, supports subtyping, and uses session types to prove an upper bound on the size of the buffers used for asynchronous communication. Channels are treated as linear, with a receive operation returning a pair of the received value and a new copy of the channel.

The work of Caires and Pfenning [16] and Gay and Vasconcelos [32] are connected by Wadler [62]. Wadler defines a process calculus *CP* with session types based on classical linear logic (note that this is different to the presentation of Caires and Pfenning [16], who use an intuitionistic presentation), and a linear functional language GV based on that of Gay and Vasconcelos [32], and defines a translation from GV to CP.

Lindley and Morris [46] describe an extended language, also called GV, and give the language a separate small-step operational semantics based on the linear λ-calculus. The work, building on a previous paper [45], addresses the unidirectional nature of the translation by providing semantics-preserving translations both from GV to CP *and* CP to GV.

2.2.5 Connections with Communicating Finite-State Automata

A separate line of work shows how it is possible to generate a monitor based on a communicating finite state machines (CFSMs) [15], and use the CFSM for lightweight runtime monitoring.

Although errors are caught only when they occur, as opposed to statically at compile time, runtime monitoring of session types is useful in a variety of circumstances. In particular, runtime monitoring of session types is useful when working with a dynamically-checked type system, when only parts of the system have been written in statically-typed languages, or in order to dynamically enforce assertions on the data [11]. Runtime monitoring is the approach taken in this thesis, as Erlang has a (largely) dynamically-checked type system[3].

Formally, a CFSM is described by a 5-tuple $M = (Q, C, q_0, \mathbf{A}, \delta)$. This formal definition is taken from the work of Deniélou and Yoshida [29].

Definition 1 (CFSM)**.** Let $\mathcal{P}$ be a set of process identifiers.

- Q defines a finite set of states
- $C = \{pq \in \mathcal{P}^2 \mid p \neq q\}$ defines a set of channels
- $q_0 \in Q$ is the initial state
- $\mathbf{A}$ is a finite alphabet of messages
- $\delta \subseteq Q \times (C \times !, ? \times \mathbf{A}) \times Q$ is a finite set of *transitions*.

Informally, a CFSM is a finite state machine where transitions are predicated on communication actions with other CFSMs over a finite set of channels. An example CFSM system is shown in Figure 2.8: CFSM A sends a message `Tick` to CFSM B, and both move from state 1 to state 2. Afterwards, CFSM B sends a message `Tock` to CFSM A, and both return to state 1.

Operations such as checking for deadlock-freedom are largely intractable for general CFSMs [15]. Deniélou and Yoshida [29] introduce a multiparty session calculus with fork and joining operations and a projection algorithm to local types, and an algorithm to translate local types to CFSMs. The fork and join operations are added to the calculus in place of the standard branching operations to guard against state explosion.

The resulting class of CFSM, known as *Multiparty Session Automata*, enjoy the properties of deadlock-freedom, communication safety, progress, and liveness *by construction*. The algorithm for constructing CFSMs from the multiparty session calculus described in the paper

[3] Without extensions, the basic type system of Erlang is dynamically-checked. However, tools such as Dialyzer (`http://www.erlang.org/doc/man/dialyzer.html`) use techniques such as success typing [44] to add some static type-checking functionality.

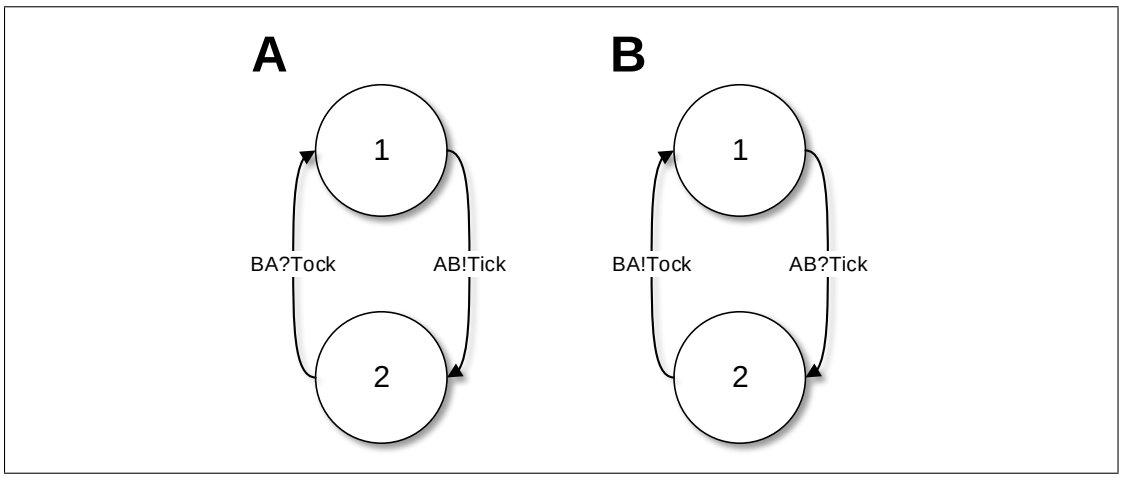


Figure 2.8: Example CFSM System

is polynomial when parallel composition is not included, and exponential in the number of parallel states when parallel composition is added.

2.2.6 Theory of Runtime Monitoring of Session Types

Work by Chen et al. [21] and Bocchi et al. [12] provide the theoretical basis for monitoring conformance to session types.

Bocchi et al. [12] define a *monitored session calculus* with monitors as a first-class construct in the process calculus itself. The process calculus is based upon multiparty session types with assertions introduced for design-by-contract development using session types [11].

The formal monitoring framework itself is based around the concept of a *network*. In this context, a network is a set of concrete endpoints, or *principals*, along with a *global transport*. A global transport consists of a global message queue and routing table, mapping abstract roles to their concrete implementations.

The formalism consists of three separate components: a reduction-based semantics for dynamic networks without monitoring, a labelled transition relation over specifications, and a semantics of monitored networks. The key to the monitoring formalism is that monitors are first class entities within the formalism, and that reductions in the semantics of monitored networks are *predicated on the labels emitted by the specification LTS*.

The semantics of monitored networks are *rejection-based*: should a principal attempt to send a message which does not match the specification, the message is not delivered.

As a result of this mechanism, the paper proves that monitored networks enjoy safety and transparency properties. Safety properties ensure that the network behaves in accordance with

the global specification, and transparency properties ensure that a monitored network behaves exactly the same as an equivalent unmonitored network which conforms to the specification. Session fidelity proves that safety and transparency hold under reduction.

2.3 The Actor Model

The actor model was initially devised by Hewitt et al. [34] and later expanded upon by Agha [3] in the context of modelling communication and concurrency in distributed systems.

> **Definition 2** (Actor [3])**.** *Actors* are computational agents which map each incoming communication to a 3-tuple consisting of
>
> 1. A finite set of communications sent to other actors;
> 2. A new behaviour (which will govern the response to the next communication processed); and,
> 3. A finite set of new actors created.

In essence, an actor is an entity which, when processing a message, can perform three actions: create a finite set of new actors, send a finite set of messages to other actors, and change how it will react to further messages. Communication in the actor model is necessarily *asynchronous*. In order to implement asynchronous communication, each actor has a unique, unforgeable *mail address*, and a message queue known as a *mailbox*.

As previously described, the actor model forms the conceptual basis for the Erlang programming language, where actors are implemented as lightweight processes. The *mail address* of an Erlang actor is a unique process ID.

2.4 Erlang

Erlang [5] is a programming language for concurrent and distributed systems, based on the actor model. Erlang's programming model involves many lightweight processes which may communicate only through explicit message passing. Processes are not implemented as native threads, but instead scheduled by the Erlang runtime system.

The design decisions to make lightweight processes central to the Erlang programming model, along with only allowing inter-process communication through explicit message passing, has several ramifications: in particular, a failure in one process is *isolated* from another, inspiring

a method of development where Erlang applications may fail and be restarted should they encounter a fault from which it is not possible to recover.

Listing 2.1: Erlang Communication Example

```
receiver() ->
  receive
    {hello, Pid} ->
      Pid ! hi,
      receiver();
    {_X, Pid} ->
      Pid ! greetings,
      receiver()
  end.

main() ->
  ReceiverPid = spawn(?MODULE, receiver, []),
  ReceiverPid ! {hello, self()},
  receive
    X -> io:format("Received: ~p~n", [X])
  end.
```

Listing 2.1 shows a simple Erlang application, demonstrating the three core communication primitives: `spawn` spawns a new actor, the `Pid ! Msg` primitive sends a message `Msg` an actor with a PID `Pid`, and the `receive` primitive searches the mailbox for messages which match the given patterns.

In this case, when the `main` function is invoked, the application spawns a new actor which executes the `receiver` function, and then sends a message `hello` to the receiver, providing the local PID. The receiver then receives the message from the mailbox, and sends the message `hi` back to the sender. Finally, the original actor receives the message from the receiver, and prints its contents—in this case, `hi`—to the console.

2.4.1 Erlang Design Philosophy

In addition to the actor-based approach to language design, Erlang is renowned for its approach to developing reliable, distributed, fault-tolerant systems.

Armstrong's PhD thesis [4] provides an excellent exposition of the requirements of distributed systems which must be highly reliable, even in the presence of errors in the software. In particular, such systems must be *fault-tolerant*, meaning that faults should be *expected*, and the software should be able to recover from, and compensate for, such faults. Additionally, such systems must be *concurrent*, meaning that they should be able to handle many simultaneous connections.

Armstrong describes Erlang as a *concurrency-oriented programming language* (COPL): a language with concurrency as a central concern in a language. A central idea in concurrency-oriented languages is the notion of a *process*. A process in a concurrency-oriented language, however, is different to the the concept of an operating system process. Operating system processes are generally heavyweight, themselves containing an address space, sets of resources, and owning child threads. Lightweight processes are instead scheduled by the runtime system of the language, allowing orders of magnitude more processes to be spawned.

In addition, a further concern is that of *failure isolation*. The idea behind failure isolation is that a failure in one process should not affect the operation of another process. Introducing the concept of isolation immediately introduces a number of additional constraints, the most notable of which is that communication using shared memory must be forbidden, as processes should be assumed to be totally independent.

Armed with lightweight isolated processes, asynchronous messages, distribution, and detection of when a process fails, Armstrong motivates the central ideology of Erlang: 'let it fail'.

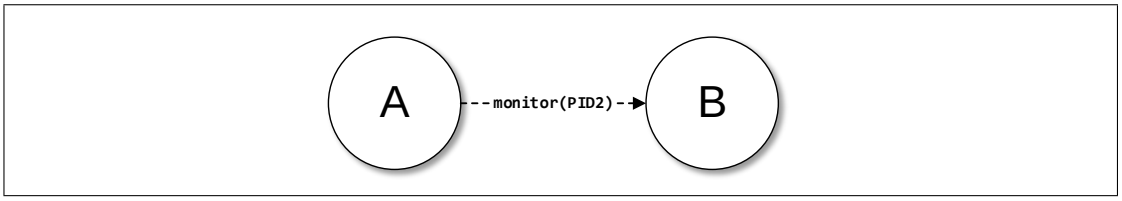
Let it Fail

Before delving too deeply into the 'let it fail' philosophy, it is necessary to note that not *all* exceptions have to be treated as fatal. As an example, attempting to open a nonexistent file throws an exception. In some scenarios, this is indeed fatal – for example, if the file is a vital configuration file needed to set up the systems. Consider, however, the case of a file server which takes a file name from the user, and returns the contents of a file. In such a case, it would be better to catch the exception, and report the error to the user.

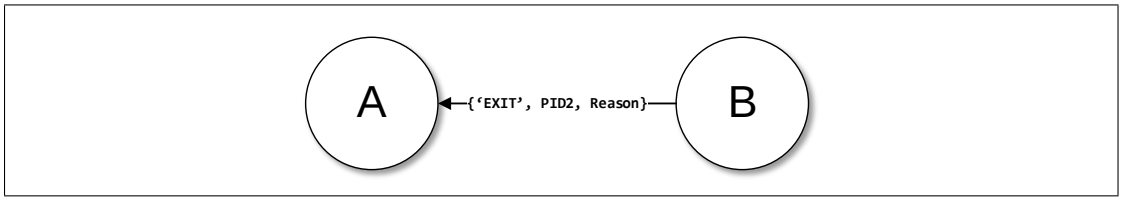
In the remainder of this section, we concentrate on the case where an error is uncorrectable – that is, there is no procedure for correcting the error in the component.

Failure detection is an important feature of a concurrency-oriented programming language. Erlang provides two native methods of failure detection: *links*, which are bidirectional, and *monitors*, which are unidirectional.

Consider the case where we have two processes A and B, with PIDs `PID1` and `PID2` respectively. Suppose A wishes to be notified when B terminates. By calling `monitor(PID2)` from A, B becomes monitored.



Now suppose B terminates. Since A has monitored B, A gets a notification of the form `{'EXIT', PID2, Reason}` from the VM, notifying it that B has terminated.



A natural question that may arise at this point is "what about if B was on a remote node, and the connection was severed, and therefore could not send a notification that it has terminated?". At this point, it is instructive to realise the distinction between processes running on the *same* VM, and those running on a *different* VM: that is, in a distributed setting.

Erlang treats the scenarios differently: should a process terminate on the same VM, the VM can reliably insert the `'EXIT'` notification into the mailbox of the monitoring process. Should a process terminate on a different VM, but with the VM still available, a message can be reliably sent between the VMs, and dispatched to the monitoring process. The final case requires somewhat more thought – what if the process is on a VM, and the VM becomes unreachable? By establishing a distributed monitor, a TCP connection is set up between the two VMs. Should the TCP connection be closed, the closed connection can be detected by the monitoring VM, which may then insert a message into the queue of the monitoring process.

Reliable failure detection enables the characteristic failure handling mechanism of Erlang: that of *supervision hierarchies*. In this model, processes are arranged into trees of *workers* and *supervisors*. Workers are processes which perform tasks, whereas *supervisors* are processes which monitor workers. Should a worker fail, a its supervisor is notified, and can take corrective action.

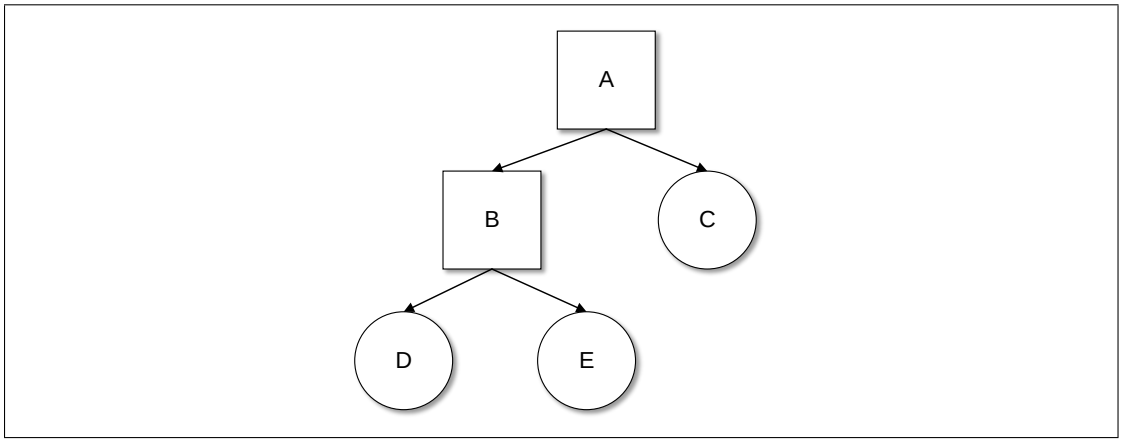


Figure 2.9: An Erlang supervision hierarchy

Consider the diagram in Figure 2.9. This basic supervision tree consists of a *root supervisor A* which supervises a supervisor *B* and a worker *C*, and workers *D* and *E* which are supervised by *B*. Should a process fail, its supervisor is notified. The supervisor can then take compensatory action, which may involve restarting the child process.

If a process is part of a supervision hierarchy, it is therefore an acceptable course of action to crash whenever an uncorrectable error occurs, as the supervisor may then restart the process, preserving the functionality of the application.

2.4.2 OTP Behaviours

A supervisor is an instance of an *OTP Behaviour*: a set of application templates which abstract out common patterns of developing Erlang applications. OTP behaviours require developers to implement callback functions, which contain application-specific logic, but abstract out lower-level behaviour. In the case of a supervisor behaviour, for example, a user simply needs to implement a single callback, `init`, which returns information about the child processes to be supervised, and upon failure, which processes should be restarted.

Another well-used OTP behaviour is a generic server, or `gen_server`. At its essence, `gen_server` is a stateful event loop which receives a message from the mailbox, executes a callback in user code, and performs an action and updates the state as a result. The OTP `gen_server` behaviour is a mature, full-featured system, allowing both asynchronous and synchronous messages.

Listing 2.2 shows the essence of a server behaviour. This simple server behaviour abstracts away from the raw communication primitives, and allows some state to be retained and updated as a result of processing messages. Abstracting the functionality in this way results in a higher level of code re-use and abstraction.

Listing 2.2: Simple Generic Server Behaviour

```
-module(simple_gen_server).
-export([behaviour_info/1]).
-export([server_startup/2, event_loop/1]).
-export([start/2, send/2]).

-record(simple_gen_server_state, {user_state, module}).

behaviour_info(callbacks) -> [{init, 1}, {handle_msg, 2}];
behaviour_info(_Other) -> undefined.

server_startup(Module, Args) ->
  StartupRes = Module:init(Args),
  case StartupRes of
    {ok, UserState} ->
      State = #simple_gen_server_state{user_state=UserState,
```

```
                                        module=Module},
      event_loop(State);
    {stop, Reason} ->
      exit(Reason)
  end.

event_loop(GenServerState) ->
  Module = GenServerState#simple_gen_server_state.module,
  UserState = GenServerState#simple_gen_server_state.user_state,
  receive
    Msg ->
      Res = Module:handle_msg(Msg, UserState),
      case Res of
        {ok, NewUserState} ->
          NewState = GenServerState#simple_gen_server_state{user_state=NewUserState},
          event_loop(NewState);
        {stop, Reason} ->
          exit(Reason)
      end
  end.

start(Module, Args) ->
  spawn(simple_gen_server, server_startup, [Module, Args]).

send(Pid, Msg) ->
  Pid ! Msg,
  ok.
```

In order to use the behaviour, a user simply needs to implement the two callbacks expected by the behaviour:

- `init/1`: Called to initialise the server. Returns either `{ok, State}` if initialised successfully, where `State` is the state of the server, or `{stop, Reason}` if not.
- `handle_msg/2`: Called when a message has been received. Returns either `{ok, NewState}` if initialised successfully, replacing the old state with `NewState`, or `{stop, Reason}` if the server should terminate.

As a small example, consider a simple integer counter. We wish to either increment the value, or retrieve the current value. This may be implemented as shown in Listing 2.3.

Listing 2.3: Simple Counter Server

```
-module(server_impl).
-export([init/1, handle_msg/2]).
-export([increment_count/1, get_count/2]).

%%% Callbacks %%%
init(_Args) -> {ok, 0}.

handle_msg(increment_count, Count) ->
```

```
  {ok, Count + 1};
handle_msg({get_count, Pid}, Count) ->
  simple_gen_server:send(Pid, Count),
  {ok, Count};
handle_msg(Other, _) ->
  error_logger:error_msg("Unhandled message: ~p~n", [Other]),
  {stop, undefined_msg}.

%%% API %%%
increment_count(Pid) ->
  simple_gen_server:send(Pid, increment_count).

get_count(Pid, ReturnPid) ->
  simple_gen_server:send(Pid, {get_count, ReturnPid}).
```

Here we can see that the `init` function initialises the state with 0, and the `handle_msg` function handles two different types of messages. In particular, the `increment_count` message increments the current counter (i.e. the server state), and the `get_count` message results in the current counter value being returned to the caller. It is regarded as good practice to abstract the messages sent to the server as API calls.

Chapter 3

Related Work

In this chapter, we review work relevant to exception handling within session types, and discuss ways in which behavioural types including session types have been integrated with the actor model.

3.1 Exceptions and Non-Standard Control Flow

There has been some work on non-standard flow within session typing systems, some of which can be useful in modelling exceptions.

We discuss four main pieces of work: Structured Interactional Exceptions [18], a method by which processes can move to a session-typed exception handler in a co-ordinated manner; Global Escape in Multiparty Sessions [17], a similar idea implemented in a multiparty session calculus; Affine Sessions [50], where session types are based on affine logic and hence can be aborted at any time; and Practical Interruptible Conversations [40], where control flow can be interrupted by an incoming message.

3.1.1 Structured Interactional Exceptions and Global Escape

The first main work on handling non-standard control flow in session types was the notion of *structured interactional exceptions* [18]. In this system, replicated channels or access points are extended to a tuple of a default process, and an exception handler. A `throw` process is introduced, which denotes a process that is throwing an exception. The operational semantics of the system allow nested exceptions, ensuring that communicating peers at a given *exception level* all transfer into the correct exception handling process.

More technically, the semantics of structured interactional exceptions rely crucially on the notion of a *meta-reduction*: a set of reduction rules to handle exceptions by erasing the remaining actions in the default process, and propagating exception messages to communicating peers in the same exception level. Further exceptions may not be thrown in an exception handler.

The system does not support channel delegation, and the *throw* process cannot be parameterised by exception types such as those in Java. Referring to the system as purely an exception handling system could be seen as a misnomer: while exceptions *do* require the sort of structured, co-ordinated transition to an exception handler as described in the paper, the technique seems to be more general, allowing the structured escape between multiple processes communicating with a shared service.

Capecchi et al. [17] extend structured interactional exceptions to the multiparty setting: the calculus described allows asynchronous exceptions to be propagated amongst a subset of the participants in the interaction. Interactional exceptions are not available in any implementation.

3.1.2 Affine Sessions

As discussed in Section 2.2.3, session types rely crucially on the notion of *linearity* – that session channels should be used exactly once. Incorporating linearity ensures that there are no race conditions, and that previous references to a linear channel cannot be used again.

Affine logic, on the other hand, still forbids contraction but allows weakening. Mostrous and Vasconcelos [50] describe a session calculus based on affine logic with *explicit weakening* in order to model computations which may be cancelled at any point during the session, hence exhibiting *at most* the behaviour prescribed by their session type. Explicit weakening is modelled using a cancellation operator $a\lightning$, read as "cancel a".

Interestingly, in addition to allowing sessions which may be aborted at any time to be modelled, the cancellation operator can be used to model exceptions. In order to do this, a `do..catch` construct is added to the language, which attempts to execute a process, and if it is aborted, instead executes the instruction handler. The calculus is stratified into *prefixes* ρ, denoting communication actions, and the remainder of the processes. Importantly, `do..catch` blocks cannot be placed around processes executing in parallel, making them address a different problem to interactional exceptions.

One elegant result of the proposed system is that the type of a process which throws and handles an exception is exactly the same a a process without any exception handling.

The semantics of session cancellation require default actions to be performed in order to resolve nondeterminism, for example when cancelling a branching operation: this may not be desired

behaviour. The typing rules are presented in a style not amenable to an algorithmic type checker implementation: the presentation requires a *non-deterministic context splitting operation* to partition a context into two disjoint halves. Without an algorithmic interpretation of the typing rules, it is difficult to implement the system in practice.

3.1.3 Practical Interruptible Conversations

The Scribble protocol description language incorporates a mechanism for non-standard control flows, where blocks can be *interrupted* by incoming messages, causing the current *interruptible* scope to be exited. As an example, consider the following protocol, which repeatedly sends a message X from role A to role B.

```
global protocol Example(role A, role B) {
    rec loop {
        X() from A to B;
        continue loop;
    }
}
```

Now, suppose that after a certain amount of time, role C wishes to advance the protocol. It then sends a message Y, which *interrupts* the protocol. At this point, A can then send a message Z to B instead. This is accomplished by using an `interruptible..with` construct: the `interruptible` block specifies a protocol which can be interrupted, and the `with` block specifies a set of messages that can interrupt the block.

```
global protocol Example(role A, role B, role C) {
    interruptible {
        rec loop {
            X() from A to B;
            continue loop;
        }
    } with {
        Y() by C;
    }
    Z() from A to B;
}
```

Interruptible conversations are implemented in the Scribble validation and projection routines. Runtime support for interruptible conversations is included in the SPY Session Python framework [53]. Interruptible conversations are designed for dynamically monitored systems as opposed to statically-typed systems. Writing exception-handling code with `interruptible` blocks is one possible use of the construct, but this is not its purpose.

We take inspiration from some of these mechanisms, but design our failure handling mechanism to handle both application-level exceptions, and failures caused due to the termination of participants, by modularly partitioning protocols into possibly-failing subprotocols [28]. Further details can be found in Chapter 6.

3.1.4 Verification of Erlang Applications

Existing work has been undertaken on the verification of distributed Erlang applications, including their communication behaviour, but take different approaches.

McErlang [30] is a model checker for Erlang. Model checking enumerates the state space of an application, and exhaustively explores this to ensure that properties hold throughout any possible execution of the program. In McErlang, properties can be specified as Linear Temporal Logic (LTL) formulae, which is a rich and expressive temporal logic. Using LTL propositions, detailed and complex propositions can be checked, which are not limited to communication patterns.

This increase in expressivity comes at a cost. Session types are specialised to checking communication protocols, and consequently it is easier to express protocols using session types than LTL formulae. Secondly, and more importantly, model checking is an expensive operation, and state explosion is a concern. One method employed by users of McErlang, as detailed by Castro et al. [20], is to restrict model checking of the application to certain smaller *scenarios*. This alleviates the issue of state explosion, but can be time-consuming. Session types can only express communication behaviour, but if messages are what needs to be verified, then session types are arguably a better option as protocols can be more concisely expressed and efficiently verified.

Nyström [56] uses static analysis to extract process supervision structures from source code, using the static analysis to identify possible errors within the structures. In order to do so, the author formalises the operational semantics for a fragment of Erlang. The purpose of the work focuses on finding problems with supervision structures as opposed to verifying communication patterns.

Colombo et al. [24] describe a runtime monitoring system for Erlang, based on the Larva [23] monitoring system for Java. ELarva makes use of Erlang's inbuilt tracing system, intercepting traces and validating the traces using monitoring processes. In order to specify a monitor, a user makes use of a lightweight DSL based on the handling of Erlang events, which is subsequently compiled into an FSM. Each monitor is spawned as a process, and the ELarva runtime delegates events to the appropriate monitor. As with McErlang, ELarva is not specialised for checking well-formedness and conformance to protocols, and thus can verify more properties.

The drawback, however, is that it is more difficult to express communication protocols, as the protocols must be manually translated into FSMs. Additionally, without session types, we cannot rely on the theoretical monitoring guarantees such as those given by Bocchi et al. [12], and we lose the guarantees provided by projecting a global type into endpoint types. Furthermore, monitoring is asynchronous and based on dynamically-generated traces, so there is no way of intercepting and not delivering a message which does not conform to the session type.

3.2 Session Types for Actor Systems

Mostrous and Vasconcelos [49] describe a term language, operational semantics, and type discipline for a minimal session-typed fragment of Erlang. The technique relies on the idea of *correlation sets*, where unique identifiers for sending and receiving messages are incorporated in session messages. The type system guarantees that sessions are completed, and that all sent messages can be received. The typing scheme described in the paper is presented using a non-deterministic context splitting operation, which is not amenable to an algorithmic implementation. Additionally, the system only supports binary sessions, not considering the multiparty setting.

Crafa [26] introduces an actor-based process calculus *AC*, along with a behavioural type system which statically guarantees deadlock freedom, and that all messages will eventually be processed. The actor-based process calculus contains a construct `react`, which waits until a message matching a specification arrives in the actor mailbox, a construct for spawning actors, and a construct for sending messages to actors. Consequently, the calculus remains close to the formal definition of an actor.

3.2.1 Multiparty Session Actors

The conceptual framework upon which we base this work is that of *multiparty session actors* [52]. In this work, Neykova and Yoshida describe the design and implementation of a system allowing actor programs using the Cell[1] actor framework for the Celery[2] distributed task processing system to be monitored using multiparty session types. Session types are written in the Scribble protocol description language, with the generated monitors used to monitor incoming and outgoing session messages.

A key insight of the work is that actors should not be restricted to fulfilling a single role in a single protocol, but instead should be able to fulfil multiple roles in multiple different proto-

[1] https://github.com/celery/cell

[2] http://www.celeryproject.org/

cols. Consequently, actors may simultaneously partake in multiple protocols, with interleaved execution between the two roles driven by the reception and handling of messages. Actors therefore become 'containers' for message handlers, and the monitors governing incoming and outgoing messages.

Consider the simple 'Hello, World!' Scribble protocol described in Listing 3.1, where role *GreetingGiver* sends a message *hello*, and role *GreetingReceiver* responds with *world*.

Listing 3.1: "Hello world" Scribble protocol

```
global protocol HelloWorld(role GreetingGiver, role GreetingReceiver) {
  hello() from GreetingGiver to GreetingReceiver;
  world() from GreetingReceiver to GreetingGiver;
}
```

Python session actors use *decorators* to register actors for protocols and roles. Python classes are annotated with `@protocol` decorators to register the actor as able to partake in a protocol, and message handlers are annotated with `@role` decorators. As a minimal example, consider the `Greeter` class in Listing 3.2, which can fulfil both 'GreetingGiver' and 'GreetingReceiver' roles.

Listing 3.2: Python Session Actor implementation of Hello World protocol

```
greeting_giver = "GreetingGiver"
greeting_receiver = "GreetingReceiver"
HelloWorld = "HelloWorld"
c = 'c'
c1 = 'c1'

@protocol(c, HelloWorld, greeting_giver, greeting_receiver)
@protocol(c1, HelloWorld, greeting_receiver, greeting_giver)
class Greeter(SessionActor):
  @role(c1, greeting_sender)
  def hello(self):
    print "Hello"
    c1.greeting_giver.send.world()

  @role(c, greeting_giver):
  def world(self):
    print "World"

  @role(c, greeting_giver)
  def join(self):
    c.greeting_receiver.send.hello()
```

The `@protocol` decorator takes the form `@protocol(key, protocol_name, role, other_roles)`, where `key` is a key representing the pairing between a protocol name and a role. A `@role` decorator

takes the form `@role(key, sender_role)`, where `key` is the key specifying a protocol-role mapping defined in a `@protocol` decorator, and `sender_role` is the name of the role sending the message.

It is instructive to distil the mappings provided by the annotations. A `@protocol` annotation serves the dual purpose of registering an actor to partake in a protocol, and providing a mapping $Key \mapsto (Protocol, Role)$ to be used within message handlers. A `@role` annotation maps a role within a protocol to a message handler.

Actors in the session actor framework may only partake in a single session instance: that is, an actor may only fulfil the `GreetingGiver` role in a single session. This restriction is too strong for server applications, which are the focus of this thesis: server applications must have the ability to interact concurrently with multiple client instances, which may be at different points within the protocol. Consequently, in the present work, we generalise multiparty session actors to allow actors to partake in multiple session *instances*.

The multiparty session actor framework includes a method by which actors with the ability to fulfil a role may be invited to fulfil the role, without a user providing a concrete endpoint identifier. The actor-role invitation workflow is further discussed and contrasted to the implementation in `monitored-session-erlang` in Section 4.6.1.

Erlang is a substantially different setting to Python. As a functional programming language, Erlang disallows mutable state, and due to the language's foundation on the actor model, must communicate purely through message passing. Consequently, various design decisions must be made which differ substantially from the Python setting. Finally, as Erlang is itself based on the actor model, we do not use AMQP and the associated abstractions provided by the AMQP framework.

Chapter 4

Monitored Session Erlang: Design and Implementation

Integrating multiparty session types and Erlang/OTP applications is challenging. How do we encapsulate the notion of a session? Can actors partake in multiple sessions at once? How do we map actors to the sessions they may take part in? How is the monitoring performed?

In this chapter, we detail the design and implementation of the `monitored-session-erlang` system. Inspired by the system implemented by Neykova and Yoshida [52], we show how Erlang actors, based on the `gen_server` behaviour, may partake in sessions. We show how monitors are generated from Scribble local projections, how actor instances are invited to fulfil roles, and how users may respond to session lifecycle events, such as when a session is started or ended.

We also show that monitoring is *orthogonal* to the supervision hierarchies used within the application: a user does not need to change the supervision hierarchy of an application in order to use `monitored-session-erlang`.

4.1 Monitored Session Erlang by Example

We illustrate the framework by example, implementing the Two-Buyer protocol described in Section 2.2.2 as an Erlang session actor. Recall that the two-buyer protocol is a simplification of a financial protocol, where two participants arrange to buy an item, arranging the cost to be shared.

Listing 4.1 shows the global protocol for the two buyer protocol, and Listings 4.2, 4.2, and 4.4 show the local projections at Buyer 1, Buyer 2, and the seller respectively.

Listing 4.1: Global protocol for the two-buyer protocol

```
global protocol TwoBuyers(role A, role B, role S) {
    title(String) from A to S;
    quote(Integer) from S to A, B;
    rec loop {
        share(Integer) from A to B;
        choice at B {
            accept(String) from B to A, S;
            date(String) from S to B;
        } or {
            retry() from B to A, S;
            continue loop;
        } or {
            quit() from B to A, S;
        }
    }
}
```

Listing 4.2: Local projection of the two-buyer protocol at buyer 1

```
local protocol TwoBuyers at A(role A,role B,role S) {
    title(String) to S;
    quote(Integer) from S;
    rec loop {
        share(Integer) to B;
        choice at B {
            accept(String) from B;
        } or {
            retry() from B;
            continue loop;
        } or {
            quit() from B;
        }
    }
}
```

Listing 4.3: Local projection of the two-buyer protocol at buyer 2

```
local protocol TwoBuyers at B(role A,role B,role S) {
    quote(Integer) from S;
    rec loop {
        share(Integer) from A;
        choice at B {
            accept(String) to A,S;
            date(String) from S;
        } or {
            retry() to A,S;
            continue loop;
        } or {
            quit() to A,S;
        }
    }
}
```

Listing 4.4: Local projection of the two-buyer protocol at the seller

```
local protocol TwoBuyers at S(role A,role B,role S) {
    title(String) from A;
    quote(Integer) to A,B;
    rec loop {
        choice at B {
            accept(String) from B;
            date(String) to B;
        } or {
            retry() from B;
            continue loop;
        } or {
            quit() from B;
        }
    }
}
```

We require three session actors: `buyer1`, `buyer2`, and `seller`.

In `monitored-session-erlang`, an actor can fulfil a number of roles in a number of protocols. The first stage of writing a `monitored-session-erlang` program, therefore, is to create a configuration file, in this case `two_buyer_conf`, registering each actor type for the roles they may play in the protocol.

```
-module(two_buyer_conf).
-export([config/0]).

config() ->
  [{buyer1, [{"TwoBuyers", ["A"]}]},
   {buyer2, [{"TwoBuyers", ["B"]}]},
   {seller, [{"TwoBuyers", ["S"]}]}].
```

The configuration file registers the actor `buyer1` to play role `A` (Buyer 1) in the `TwoBuyers` protocol, and does similar registrations for `buyer2` and `seller`.

The next stage is to implement the logic for each actor. We begin with `buyer1`, and shall describe each section of the actor in turn.

Each session actor implements the `ssa_gen_server` behaviour. The `ssactor_init` function is called when the actor is started, and provides two arguments: a list of arguments specified by the user, and the PID of the monitor. When `buyer1` is started, it initiates a the `TwoBuyers` protocol, using the `conversation:start_conversation` API function, taking role A. The return value of `ssactor_init` is the state of the actor: in this case, no state is required, so the `no_state` atom is returned.

```
-module(buyer1).
-behaviour(ssa_gen_server).
```

```
-compile(export_all).

ssactor_init(_Args, Monitor) ->
  % Start the conversation
  io:format("Starting conversation in buyer1.~n", []),
  conversation:start_conversation(Monitor, "TwoBuyers", "A"),
  no_state.
```

The next four callbacks, `ssactor_join`, `ssactor_conversation_established`, and `ssactor_conversation_ended`, handle lifecycle events of the session.

The `ssactor_join` callback is called whenever an actor is invited to join a session; in this implementation, the actor always accepts.

The `ssactor_conversation_established` function is called when all participants have accepted invitations to join the session, and communication may begin. The callback provides 5 arguments: a protocol name, role name, session ID, conversation key, and the state of the session. Once `ssactor_conversation_established` callback has been invoked, it is possible to begin sending session messages using the `conversation:send` call. In this case, `buyer1`, acting as role A, sends a message of type `title` with a payload of *Learn You Some Erlang*.

The `ssactor_conversation_error` function is called when it was not possible to establish a session, and finally the `ssactor_conversation_ended` is called whenever an established session ends. In the implementation, neither are important in this case: we simply log their occurrence.

```
ssactor_join(_, _, _, State) -> {accept, State}.

ssactor_conversation_established("TwoBuyers", "A", _CID, ConvKey, State) ->
  conversation:send(ConvKey, ["S"], "title", [], ["Learn You Some Erlang"]),
  {ok, State}.

ssactor_conversation_error(_PN, _RN, Error, State) ->
  actor_logger:err(buyer1, "Could not establish conversation: ~p~n", [Error]),
  {ok, State}.

ssactor_conversation_ended(CID, _Reason, State) ->
  actor_logger:info(buyer1, "Conversation ~p ended.~n", [CID]),
  {ok, State}.
```

The `ssactor_handle_message` callback provides 8 arguments: the protocol name, role name, session ID, role of the sender, message name, payload, state, and a `ConvKey`. A `ConvKey` can be treated as an opaque value which allows messages to be checked against the correct monitor; further details may be found in Section 4.4.1. Much like with the standard `handle_cast` and `handle_call` callbacks in the `gen_server` behaviour, it is possible to pattern match on the arguments in order to determine which handler to invoke. In this instance, the main determining factor in which handler to invoke is the message name.

As per the protocol, when the buyer receives a quote from the seller, it sends the share which it expects the second buyer to pay to buyer 2: in this setting, the buyer splits the price evenly. In order to send to buyer 2, the user calls `conversation:send`, providing the `ConvKey` bound in the function header. In terms of sending messages, this is the last message the buyer needs to send: while the handlers for other messages must be present, they simply log that the messages were received.

```
ssactor_handle_message("TwoBuyers", "A", _, SenderRole, "quote", [QuoteInt], State,
    ConvKey) ->
  actor_logger:info(buyer1, "Received quote of ~p from ~s", [QuoteInt, SenderRole]),
  conversation:send(ConvKey, ["B"], "share", [], [QuoteInt div 2]),
  {ok, State};
ssactor_handle_message("TwoBuyers", "A", _, SenderRole, "accept", [Address], State,
    _ConvKey) ->
  actor_logger:info(buyer1, "~s accepted quote; received address (~p)", [SenderRole,
      Address]),
  {ok, State};
ssactor_handle_message("TwoBuyers", "A", _, SenderRole, "retry", _, State, _ConvKey) ->
  actor_logger:info(buyer1, "~s wants to retry", [SenderRole]),
  {ok, State};
ssactor_handle_message("TwoBuyers", "A", _, SenderRole, "quit", _, State, _ConvKey) ->
  actor_logger:info(buyer1, "~s wants to quit", [SenderRole]),
  {ok, State};
ssactor_handle_message("TwoBuyers", "A", _CID, _SenderRole, Op, Payload, State, _ConvKey)
    ->
  actor_logger:err(buyer1, "Unhandled message: (~s,  ~w)", [Op, Payload]),
  {ok, State}.
```

The remaining functions are not used in this simple example as they refer to more advanced features of `monitored-session-erlang` described in Chapters 5 and 6. Additionally, we do not make use of any of the standard `gen_server` callbacks as we only use session messages.

```
ssactor_call(_, _, _, _, _, _, _, _) -> {stop, unexpected_session_call, State}.
ssactor_become(_, _, _, ConvKey, _) -> {stop, unexpected_become, State}.
ssactor_subsession_complete(_, _, State, _) -> {stop, unexpected_become, State}.
ssactor_subsession_failed(_, _, State, _) -> {ok, State}.
ssactor_subsession_setup_failed(_, _, State, _) -> {ok, State}.

handle_call(_, _, State) -> {stop, unexpected_call, State}.
handle_cast(_, State) -> {stop, unexpected_cast, State}.
handle_info(_, State) -> {stop, unexpected_info, State}.
code_change(_, State, _) -> {ok, State}.
terminate(_, _) -> ok.
```

Finally, the message handler for the second buyer is as follows:

```
ssactor_handle_message("TwoBuyers", "B", _CID, SenderRole, "quote", [QuoteInt], _State,
    _ConvKey) ->
```

```
  actor_logger:info(buyer2, "Received quote of ~p from ~s", [QuoteInt, SenderRole]),
  {ok, no_state};
ssactor_handle_message("TwoBuyers", "B", _CID, SenderRole, "share", [Share], _State,
    ConvKey) ->
  actor_logger:info(buyer2, "Received share quote (~p) from ~s", [Share, SenderRole]),
  if Share >= ?PRICE_THRESHOLD ->
       % Nah, we aint paying that
       actor_logger:info(buyer2, "Rejected share quote (threshold ~p)", [?PRICE_THRESHOLD
           ]),
       conversation:send(ConvKey, ["A", "S"], "quit", [], []);
     Share < ?PRICE_THRESHOLD ->
       % We can afford it: accept, send address to buyer2 and server,
       % and retrieve the delivery date from the server
       actor_logger:info(buyer2, "Accepted share quote (threshold ~p)", [?PRICE_THRESHOLD
           ]),
       conversation:send(ConvKey, ["A", "S"], "accept",
                         [], ["Informatics Forum"])
  end,
  {ok, no_state};
ssactor_handle_message("TwoBuyers", "B", _CID, SenderRole, "date", [DeliveryDate], _State
    , _ConvKey) ->
  actor_logger:info(buyer2, "Received delivery date of ~s from ~s", [DeliveryDate,
      SenderRole]),
  conversation:end_conversation(ConvKey, normal),
  {ok, no_state};
ssactor_handle_message("TwoBuyers", "B", _CID, _SenderRole, Op, Payload, _State, ConvKey)
     ->
  actor_logger:err(buyer2, "Unhandled message: (~s, ~w)", [Op, Payload]),
  {ok, no_state}.
```

Upon receiving the share which buyer 1 expects buyer 2 to pay, the second buyer checks whether the price is below a given threshold. If so, the buyer sends an accept message, specifying the delivery address. If not, then it sends a `quit` message. Finally, upon receiving a delivery date, the protocol has ended, and so is terminated using the `conversation:end_conversation` function. The implementation of the `seller` actor is similar.

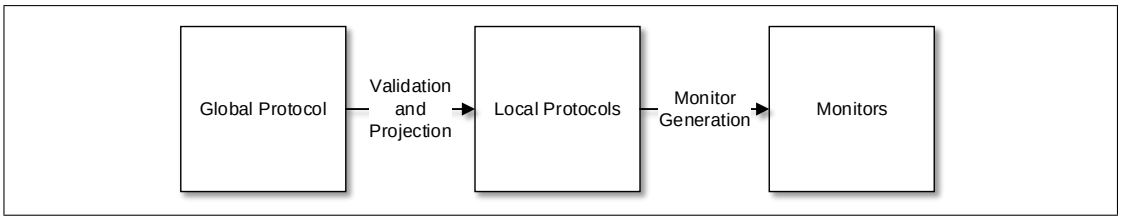
4.2 System Overview

The aim of the library is to make use of the theory of multiparty session types to ensure that protocols are safe, and to provide guarantees about the behaviour of the application should messages conform to the session type.

The Scribble protocol description language [39] is a high-level domain-specific language (DSL) used for the specification of protocols, using the theory of multiparty session types. The `scribble-java` project[1] is a set of tools for the Java programming language. The Scribble

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

toolchain provides a parser, a tool for validating the well-formedness of protocols, a tool for projecting global types to local types, a tool for validating traces against a session type, and a Java monitoring implementation.



Scribble is used as the language to describe protocols as it is a well-maintained framework implementing the theory of multiparty session types, and the `scribble-java` toolchain can be thought of as a trusted base on which to write protocols. The disadvantage of this approach is that users are required to also install the `scribble-java` tools, although this should not be an issue in practice.

As a high-level overview, the system works as follows, and as shown in Figure 4.1:

- The `monitored-session-erlang` implementation is provided with *local* projections of protocols, which are parsed and transformed into CFSM-based monitors using the algorithm described by Deniélou and Yoshida [29].
- Erlang session actors are Erlang actors which can fulfil multiple roles in multiple protocols. A configuration file defines which roles, in which protocols, actors may fulfil.
- A session *initiator* begins a session, which is registered for the role. At this point, eligible actors are invited to fulfil the roles. When all roles are fulfilled, the participants of the session are notified that the session has been initiated successfully. Conversely, if it is not possible to fulfil all of the roles (for example, if all actors eligible to fulfil a role decline the invitation, or there are no actors registered to fulfil the role), then all actors registered in the session are notified of the failure.
- Session messages are sent using the 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 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 participants have terminated and the session can no longer safely proceed: this is particularly important due to the multicasting capabilities of multiparty session types, as we wish to provide atomic multicast capabilities.

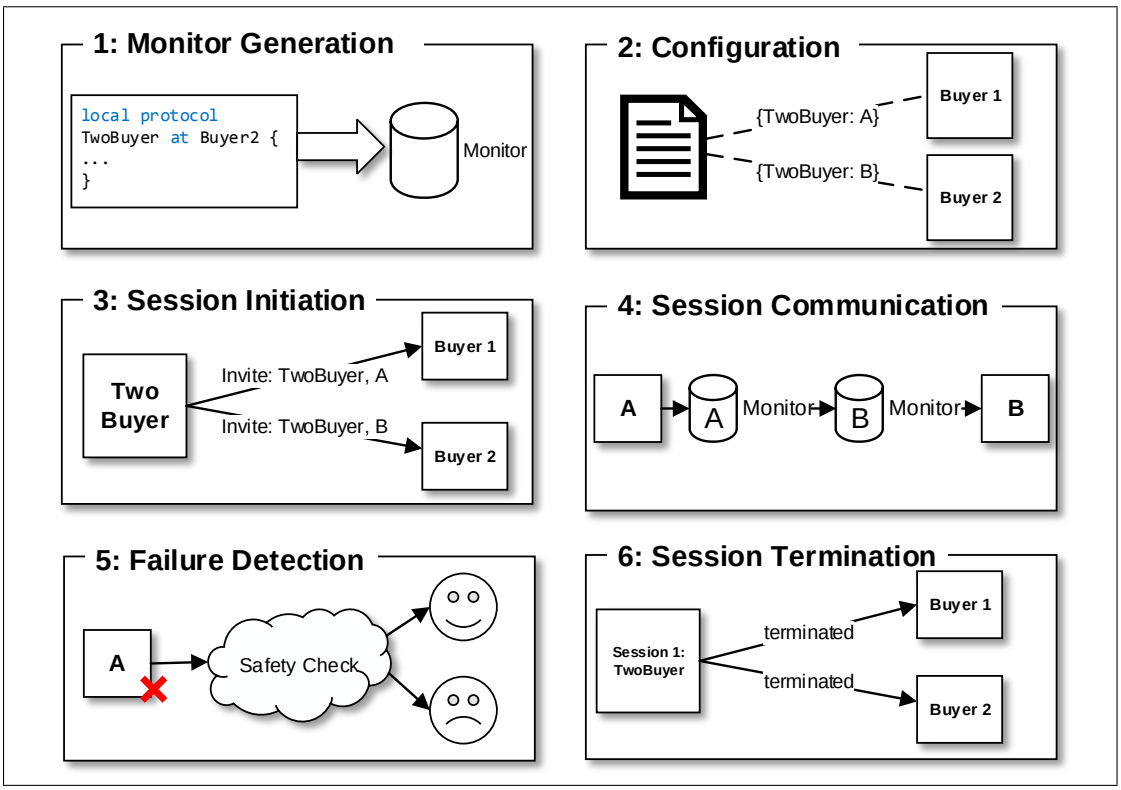


Figure 4.1: Overview of `monitored-session-erlang` system

- 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.

Importantly, we view the session system as *largely, but not completely, orthogonal to supervision tree structures*. Erlang developers may write and structure their applications as they would normally, using the well-defined and tested Erlang development methodologies. The reason the two are not *completely* orthogonal is that exceptions, for example an exception raised when a monitor rejects an outgoing message, may (if uncaught) result in a process being terminated.

The approach of throwing an exception in the actor is consistent with Armstrong's methodology for developing applications in the presence of possible software errors. Once the error is detected, we know that the actor has deviated from the desired communication protocol, and therefore has deviated from its specification. Consequently, the error should be logged, and the actor should terminate and be restarted by its supervisor.

4.3 System Structure

Figure 4.2 shows the supervision hierarchy for the `monitored-session-erlang` runtime. Two worker processes, `protocol_registry` and `actor_registry`, are used for session initiation, and

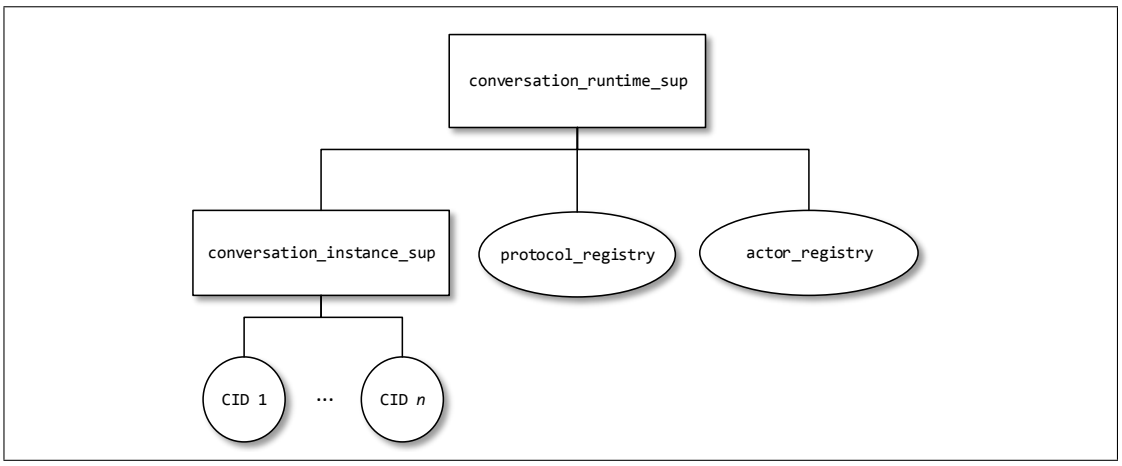


Figure 4.2: Supervision tree of `monitored-session-erlang` system

record the process IDs of active session actor instances. Both processes are restarted should they fail. The `conversation_instance_sup` process is a supervisor for `conversation_instance` processes, which are spawned upon the creation of a session, and handle session-specific actions. It is worth noting that if a `conversationInstance` process terminates, it is *not* restarted by `conversation_instance_sup`: the purpose of the supervisor is to add structure to the runtime processes.

4.4 Erlang Session Actors

The core concept in the system is the idea of an Erlang session actor. In keeping with the work of Neykova and Yoshida [52], we maintain the idea that actors can fulfil roles in multiple protocols, with the session actor behaviour 'demultiplexing' messages received from the mailbox as they are processed.

At its essence, an Erlang session actor consists of:

- A process ID P
- A mailbox M
- Actor State `State`
- A lookup table of monitors $\mathcal{M}$: (`RoleName` $\times$ `SessionID`) $\mapsto$ `Monitor`
- A handler function
  $\mathcal{H}$: (`MessageName` $\times$ `Payload` $\times$ `RoleName` $\times$ `SessionID` $\times$ `State`) $\mapsto$ (`SentMessages` $\times$ `State`)

As always, Erlang actors are uniquely identified by a process ID, and have a *mailbox*, which is an ordered queue of incoming messages. In addition, Erlang session actors provide two

additional pieces of functionality: a monitor lookup table, which maps roles and session IDs to monitors, and a handler function, which specifies the function to run when handling a message. A monitor may be uniquely identified by a pair of a role name and a session ID.

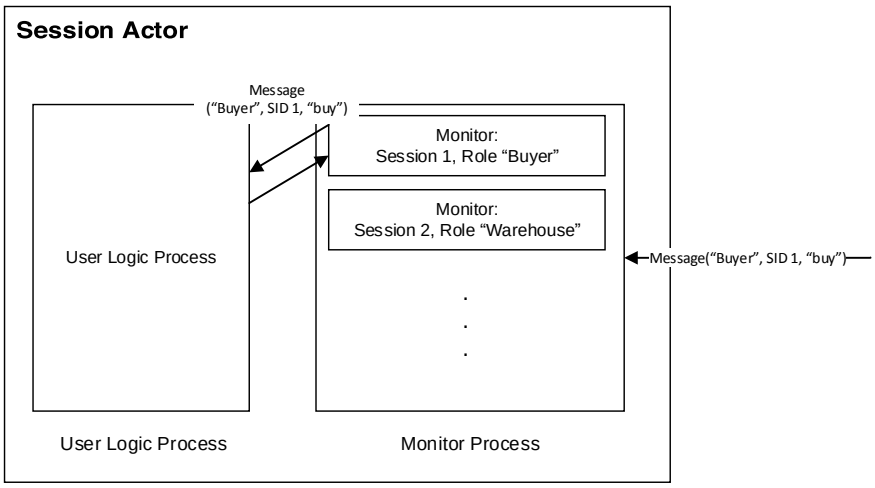


Figure 4.3: Erlang Session Actors

Figure 4.3 shows the design of Erlang session actors. Of particular importance is the concept that the monitoring process is *external* to the session actor process implementing the user application code, and that all communication to and from the user session actor process is fully mediated by the monitor process.

4.4.1 External Monitoring Process

The design decision to have a separate monitoring process instead of having a single process handling both monitors and application logic was made for several reasons. Firstly, and most importantly, by keeping the monitor as a separate process, we allow messages to be sent at any point during the handler, while eliminating the need for linearity tracking. Consider the following example:

```
ssactor_handle_message("PerformExpensiveComputation", "A", _, _, "quote", [Quote], State,
    Monitor) ->
  Monitor1 = conversation:send("B", "ComputationStarted", [], Monitor),
  Result = do_expensive_computation(),
  Monitor2 = conversation:send("B", "ComputationFinished", [Result], Monitor1),
  {ok, Monitor2, State}.
```

In this example, an actor plays role *A* in a protocol *PerformExpensiveComputation*. As the protocol name suggests, the code sends a message to another actor *B* to signify that the computation has started, and a message *ComputationFinished* along with the result at the end of the

computation. In this example, the `conversation:send` function sends a message, and returns a new monitor when finished.

The problem here is that there is nothing stopping us using `Monitor` or `Monitor1` multiple times, which would allow messages to be sent multiple times—ergo violating session fidelity. In a functional language without mutable state, some form linearity tracking would be required.

The other method of sending messages involves not sending the messages inline, but only at the end of the computation.

```
ssactor_handle_message("PerformExpensiveComputation", "A", _, _, "quote", [Quote], State)
    ->
  Result = do_expensive_computation(),
  {ok, [{"ComputationStarted", []}, {"ComputationFinished", [Result]}], State}.
```

The issue with this approach is that the framework becomes too restrictive on when messages may be sent. In this example, the `ComputationStarted` message becomes meaningless, as it is not delivered until after the computation has finished executing.

The problem can be alleviated by having an external monitoring process. Instead of providing a concrete monitor to the user, or requiring all messages to be sent at the end of a message handler, we instead provide users with a *conversation key*, or `ConvKey`:

> **Definition 3** (Conversation Key)**.** A conversation key is a 3-tuple (M, R, S), where M is the process ID of the monitor, R is the name of the role that the participant is playing in the session, and S is the process ID of the `conversation_instance` process for the session.

To the user, a `ConvKey` is intended to be an opaque, abstract value. Passing the value along with any send operations allows the appropriate monitor to be found, in order to check the outgoing message and update the monitor state. Since the monitor state will be updated as a result of the message being sent, there is no requirement for linearity tracking.

The final iteration of the above example is as follows:

```
ssactor_handle_message("PerformExpensiveComputation", "A", _, _, "quote", [Quote], State,
    ConvKey) ->
  conversation:send("B", "ComputationStarted", [], ConvKey),
  Result = do_expensive_computation(),
  conversation:send("B", "ComputationFinished", [Result], ConvKey),
  {ok, State}.
```

It is worth noting here that the update in the state of the monitor is hidden from the user, and that messages are still free to be sent at any point in the session.

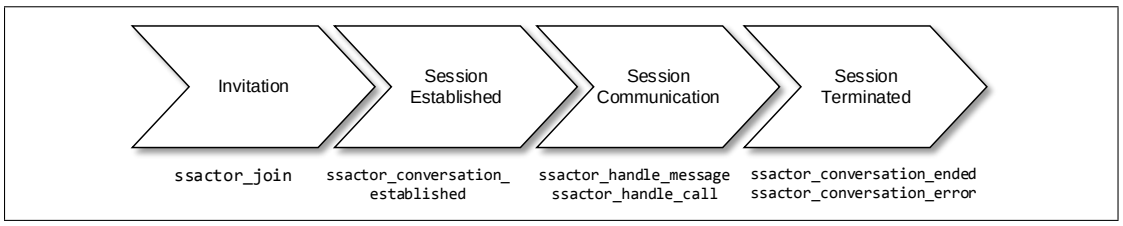


Figure 4.4: Session lifecycle, and associated callbacks

A further reason for adding a separate monitoring process instead of performing monitoring in the same process is that of concurrency: as actors are single-threaded, the use of a single monitoring process would mean that accepting or rejecting incoming messages could only happen when the actor process is not running application logic. This would hamper the efficiency of synchronous error reporting.

Finally, it is worth noting that a failure in either the monitor process or the user logic process means that the other process (the monitor process in the case of a user logic process failure, or the user logic process in the case of a monitor process failure) cannot function. Consequently, the processes are bidirectionally linked, and a failure in one will result in the termination of the other.

4.4.2 The `ssa_gen_server` Behaviour

The core target for the `monitored-session-erlang` library is Erlang/OTP server applications, hence we target a higher level of abstraction than the standard Erlang send and receive primitives. Instead, we build work on top of the Erlang/OTP generic server, or `gen_server` behaviour. The idea behind abstracting out common functionality in Erlang behaviours, in particular an analysis of a skeleton `gen_server`, is described further in Section 2.4.2.

The `ssa_gen_server` behaviour is a generic server behaviour which provides functionality for session communication. In particular, in addition to the standard `gen_server` callbacks such as `handle_call` and `handle_cast`, we require several further callback which are called at various points within the session lifecycle (shown in Figure 4.4): for example, upon setup, message receipt, and teardown of the session. More detailed information on the callbacks may be found in Appendix A.

Unlike standard `gen_server` implementations, the PID of the `ssa_gen_server` process is not publicly known. Recall that Erlang session actors are in fact a *pair* of processes: a monitoring process (`actor_monitor`), and the process which implements the user logic (`ssa_gen_server`). Spawning a session actor spawns an `actor_monitor` which subsequently spawns the `ssa_gen_server` instance, and it is the PID of the `actor_monitor` instance that is returned. This allows session

messages to be checked by the appropriate monitor prior to reaching user code.

4.5 Messaging Semantics

Communication in systems with multiparty session types is asynchronous: upon sending a message, a process does not need to wait for the message to be received before proceeding with the remainder of the session. This is achieved by associating queues with each process: when a message is sent to a process, it is appended to the queue, whereas receiving a message reads a value from the queue. In order for such a strategy to be safe, however, it is assumed that asynchronous messages are not re-ordered in transit: that is, that messages are received in the order in which they are sent. Such an assumption is made possible in practice through transport-layer protocols such as TCP.

Thankfully, the semantics of Erlang message-passing have been well-studied, mostly in the context of the McErlang [30] model checking system. Erlang semantics can be stratified into three main layers: functional, which describe the semantics of the value language; single-node, which govern communications within a VM; and distributed, which describe communications between Erlang nodes.

In particular, the distributed semantics are described by Claessen and Svensson [22] and later refined and expanded upon by Svensson and Fredlund [59]. Single-node semantics place stronger guarantees on message ordering, since message delivery is instantaneous: when a process A sends a message m to process B on the same VM, the VM can place m in the mailbox of B as part of the send operation.

In a setting with multiple nodes, however, communication must take place, and the guarantee of instantaneous delivery cannot be made. Importantly, however, messages sent from A to B remain ordered, but ordering cannot be guaranteed if messages take different routes. Thankfully, however, multiparty session types are based on exactly this assumption, ruling out protocols which could cause race conditions through the use of causality analysis.

An issue with the standard `gen_server` OTP behaviour, upon which we base our session actor implementation, is that messages may be reordered when two nodes connect for the first time. This is alleviated through the use of the `gen_server2` behaviour[2], a drop-in replacement which fixes the reordering issue.

[2] https://github.com/rabbitmq/rabbitmq-server/blob/master/src/gen_server2.erl

4.6 Sessions

4.6.1 Session Initiation

In order to start a session, it is necessary to find actors to fulfil all roles in the protocol. This is the *actor-role instantiation problem*, detailed by Neykova and Yoshida [52].

In the work on multiparty session actors by Neykova and Yoshida [52], the problem is addressed by an actor-role invitation workflow, based on AMQP exchanges. Briefly, AMQP (Advanced Message Queueing Protocol) is a protocol for message queues, enabling ordered delivery of messages. AMQP introduces the idea of an *exchange*, which can distribute messages to other entities. There are multiple types of exchanges: direct exchanges, which send a message directly to a subscriber with a given key; broadcast exchanges, which deliver a message to all subscribers; and round-robin exchanges, which maintain a circular queue of messages and deliver a message to a subscriber and advance their point in the queue.

The actor discovery workflow in the Python implementation of multiparty session actors involves creating AMQP round-robin exchanges per type of actor, and AMQP broadcast exchanges per type of protocol. Upon session initiation, a new exchange is created with a unique session ID. Next, the protocol exchange sends an invitation to actors which may fulfil the roles. Finally, should an actor accept the invitation, the actor binds itself to the protocol instance exchange to receive all messages addressed to a certain role. When sending a message, the message is sent to the protocol instance exchange, which delivers to the actor registered to receive the message.

Erlang provides native actor functionality, so we do not use AMQP. Our implementation of the actor-role invitation makes substantial simplifications over that presented by Neykova and Yoshida [52]. In particular, we require two registries: `protocol_registry`, which associates protocol names with (*Role, Monitor*) pairs, and an `actor_registry`, which associates active session actors with the roles they may fulfil. More concretely, the `actor_registry` is a map *Protocol Name* $\mapsto$ (*Role* $\mapsto$ *Actor PID*).

When a session actor is spawned, it is registered by the `actor_registry` process, which allows it to be invited to roles. When a session actor process terminates, it is deregistered.

The procedure for initiating a session (shown in Figure 4.5) 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.

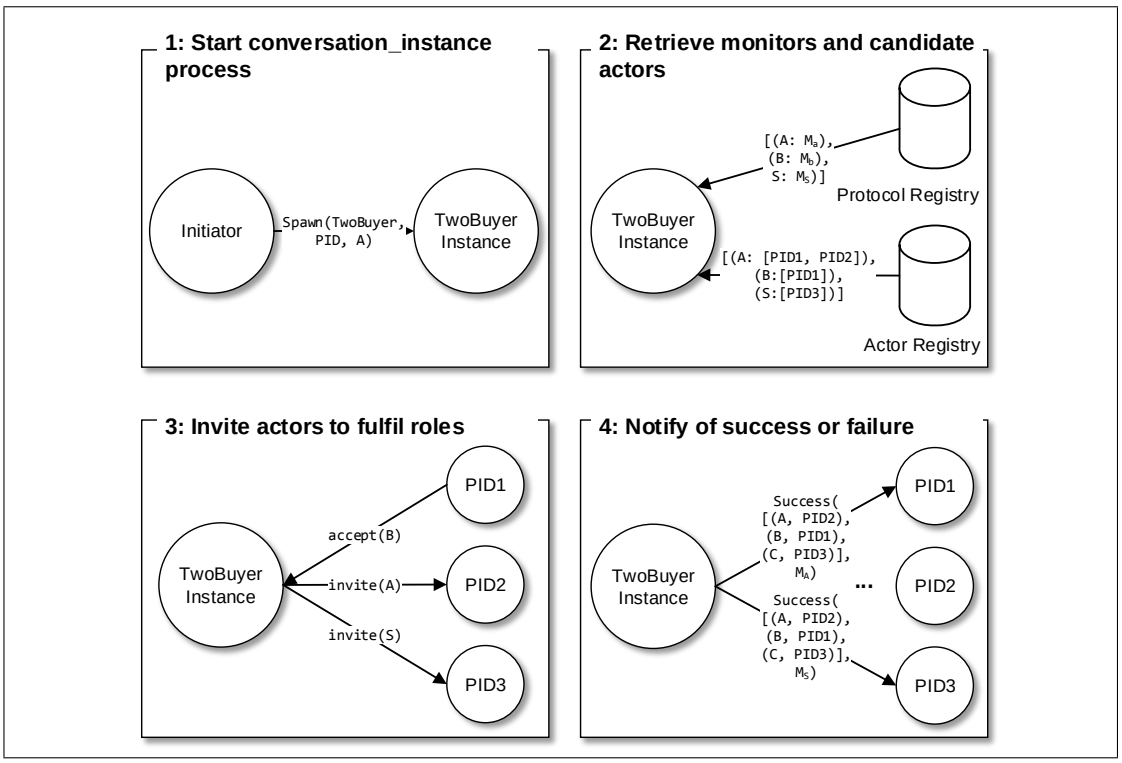


Figure 4.5: Actor-Role Invitation Workflow for `monitored-session-erlang`

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. For each role in turn, `conversation_instance` process invites eligible actors to fulfil the role.

   - If all roles have been fulfilled, then the setup is complete, each actor is notified, and the `ssactor_conversation_established` callback is called in each participant.

   - If it is not possible to fulfil a role, for example because no session actor which can fulfil the role is active, or 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 new protocol are notified, resulting in the `ssactor_conversation_error` callback being invoked.

4.6.2 Role Registration

As previously stated, session actors are entities which may fulfil multiple roles in multiple instances of multiple protocols. This is in contrast to the approach of Neykova and Yoshida [52], where each actor could not participate in an arbitrary number of session instances. The

decision to change this restriction was primarily due to the application domain of Erlang server applications, where actors often need to serve requests from an arbitrary number of clients.

Actors are associated with roles through the use of a configuration file. This takes the following form:

```
SessionConfig = [ActorSpec]
ActorSpec = {ActorName, [ProtocolSpec]}
ProtocolSpec = {ProtocolName, [RoleName]}

ActorName = atom()
ProtocolName = string()
RoleName = string()
```

In essence, a session configuration comprises a list of tuples, where each tuple maps an actor type to a protocol name and a list of roles which it may fulfil in the protocol.

As an example, consider the Two Buyer protocol, with three actor types: `buyer1`, `buyer2`, and `seller`. These three actor types should register for the "TwoBuyer" protocol, for the "Buyer1", "Buyer2", and "Seller" roles respectively. A suitable configuration file would therefore be:

```
config() ->
  [{buyer1, [{"TwoBuyers", ["Buyer1"]}]},
   {buyer2, [{"TwoBuyers", ["Buyer2"]}]},
   {seller, [{"TwoBuyers", ["Seller"]}]}].
```

The configuration file can then be used to populate both the protocol-role mapping in each `actor_type` process, and the role-actor type mapping in each `protocol_type process`, hence allowing the actor-role discovery process to take place.

4.7 Monitoring

Monitoring forms the core of the monitored session actors system. The monitoring subsystem consists of two phases: *generation*, where a CFSM-based monitor is constructed from the local projection of a multiparty session type, and the *runtime*, where the state of a monitor is maintained, and incoming and outgoing messages are checked against the monitor.

4.7.1 Generation

The first step of monitor generation is parsing the local projection of the session type: this is achieved using the Erlang `leex`[3] and `yecc`[4] tools, resulting in an abstract syntax tree representing the protocol.

The algorithm implemented by `monitored-session-erlang` largely follows the algorithm presented by Deniélou and Yoshida [29].

In essence, transitions between states are predicated on send and receive operations. As an example of monitor generation, consider the local projection of the two buyer protocol at Buyer 2:

```
local protocol TwoBuyers at B(role Buyer1, role Buyer2, role Seller) {
    quote(Integer) from Seller;
    share(Integer) from Buyer1;
    rec loop {
      choice at Buyer2 {
          accept(String) to Buyer1, Seller;
          date(String) from Seller;
      } or {
          retry() to Buyer1, Seller;
          continue loop;
      } or {
          quit() to Buyer1, Seller;
      }
    }
}
```

The monitor generated from the projection is shown in Figure 4.6.

Naïvely implemented, the size of a generated CFSM is exponential in the size of the global type used to generate the projections, due to the handling of parallel composition, since all interleavings must be accounted for. Without parallel composition, however, the algorithm is polynomial.

An optimisation described by Hu et al. [40] uses the concept of *nested finite state machines* in order to reduce the complexity of monitor generation to polynomial, even in the presence of parallel composition.

As an example, consider the following protocol:

[3] http://erlang.org/doc/man/yecc.html
[4] http://erlang.org/doc/man/leex.html

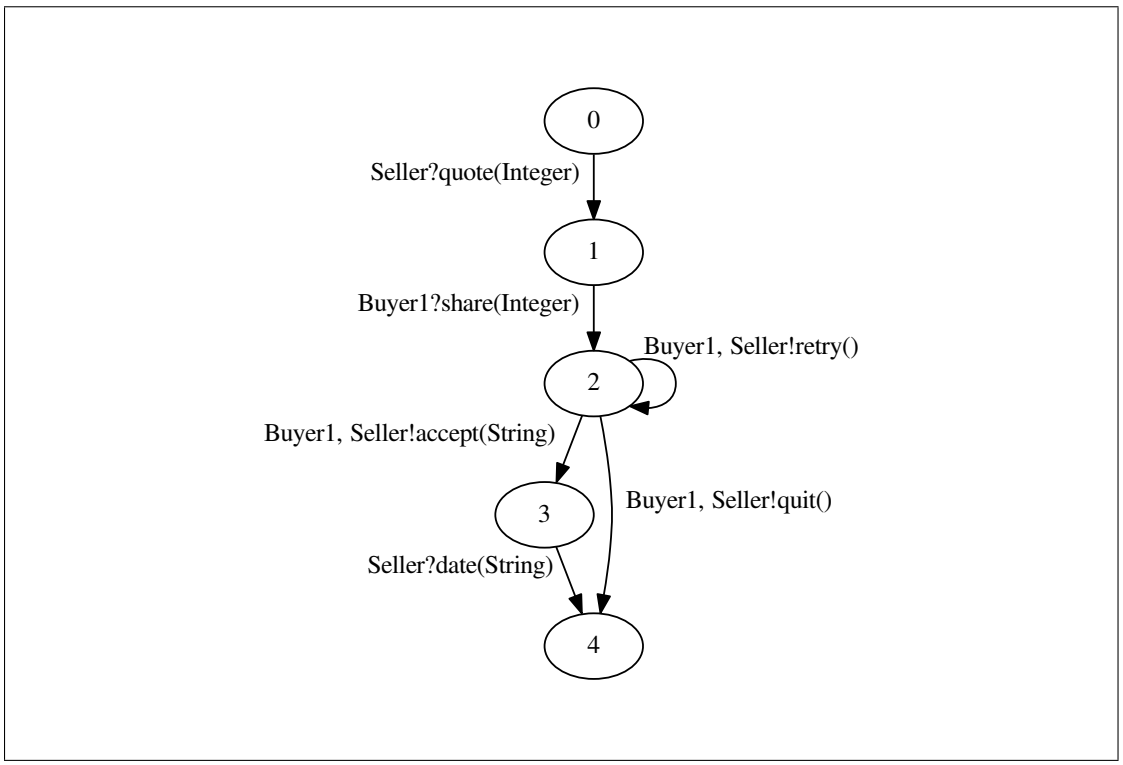


Figure 4.6: Monitor for Buyer2 projection of the Two Buyer Protocol

```
global protocol InterleavingExample(role Role1, role Role2) {
  par {
    A() from Role1 to Role2;
    B() from Role2 to Role1;
  } and {
    C() from Role2 to Role1;
    D() from Role1 to Role2;
  }
}
```

The protocol would permit the traces $\{ABCD, ACBD, ACDB, CABD, CADB, CDAB\}$. Naïvely implemented, the monitor would be as shown in Figure 4.7.

With the nested FSM optimisation, however, we have a much more manageable monitor, as shown in Figure 4.8.

The nested FSM optimisation means that instead of calculating the interleaving product of all possible message combinations, we instead calculate a separate FSM for each branch of a parallel composition block. Due to the message uniqueness requirement, it is possible to ensure that at most one transition matches from all parallel blocks. The nested FSM has two sub-FSMs, each of which is the monitor for a branch of the `par` block. Once both nested FSMs have reached state 3, the outer FSM can proceed to state 2.

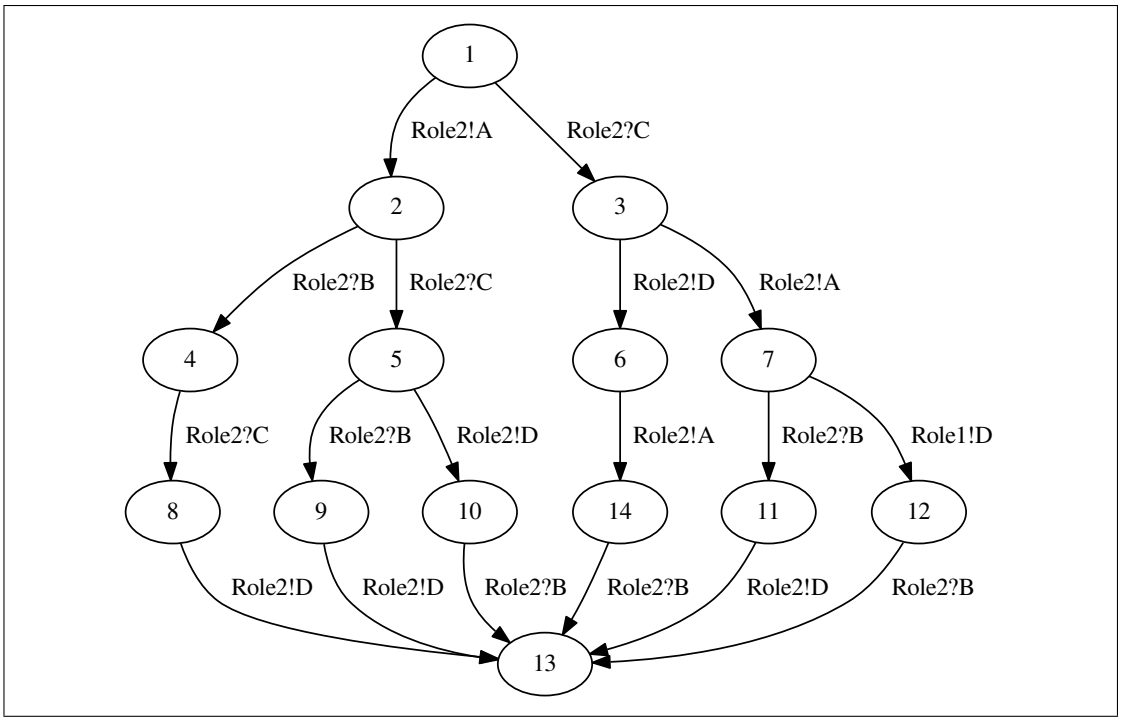


Figure 4.7: Monitor without nested FSM optimisation

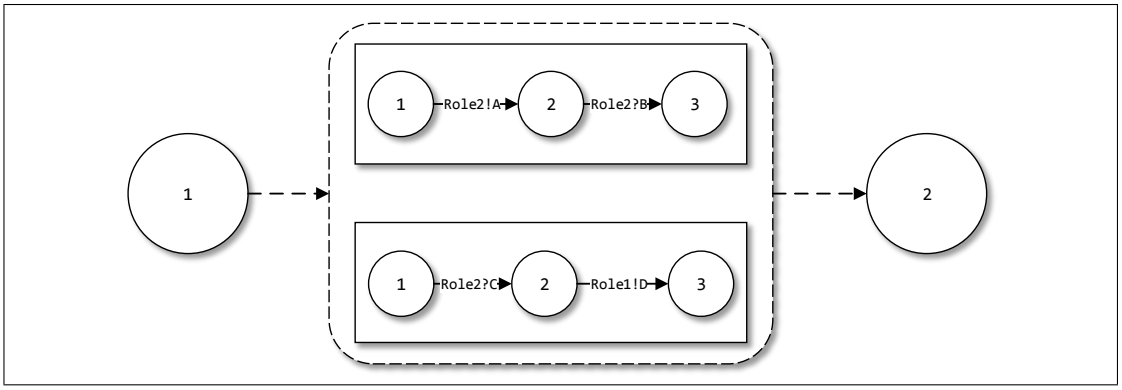


Figure 4.8: Monitor without nested FSM optimisation

Using nested FSMs to model parallel composition as opposed to computing all possible interleavings results in a slight loss of expressivity—messages in branches which are composed in parallel must be distinct—but the loss of expressivity seems a worthy tradeoff in return for tractable monitor sizes and generation times.

In the implementation, we define three types of transition:

Send – `{send, ToID, Recipients, MessageName, PayloadTypes}`
Specifies a transition that may be taken to the state with ID `ToID`, by sending a message with name `MessageName` and types `PayloadTypes`, to recipient roles `Recipients`.

Receive – `{recv, ToID, Sender, MessageName, PayloadTypes}`
Specifies a transition that may be taken to the state with ID `ToID`, by receiving a message with name `MessageName` and types `PayloadTypes`, from the sender role `Sender`.

Par – `{par, ToID, NestedFSMIDs}`
Specifies a transition that may be taken to state with ID `ToID` when all nested finite state machines with IDs `NestedFSMIDs` are in a terminal state.

A notable difference to standard CFSMs is that 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 6.2 provide mechanisms to enforce atomic multicast. `Par` transitions may be taken if all nested finite state machines with IDs in `NestedFSMIDs` are in a terminal state, thus providing a 'join' abstraction.

4.7.2 Runtime

Once a monitor has been generated, it may be used to check incoming and outgoing messages against the local specification for a type. At its outermost level, the runtime representation of a monitor is a hashtable `monitor_instances`, mapping monitor IDs to monitor instances. Monitor 0 always maps to the *root* monitor: in protocols without parallel composition, monitor 0 will be the only entry. In monitors with parallel scopes, there will be one monitor entry per parallel branch. The `monitors` field contains monitor specifications. The `reachability_dicts` field is used for push-based failure detection, described in Section 6.2.1.

```
-record(outer_monitor_instance, {protocol_name,
                                 role_name,
                                 monitors,
                                 monitor_instances,
                                 reachability_dicts
                                }).
```

A monitor instance consists of four main components: a unique FSM ID within the monitor, the current state of the monitor, hashtables mapping state numbers to state descriptions, and state numbers to transitions.

```
-record(monitor_instance, {fsm_id,
                           current_state = 0,
                           states,
                           transitions
                          }).
```

Checking a message involves checking whether any transitions can be made from the current state. In the case of send and receive transitions, the message will need to be checked against the condition of the transition. If the message passes the check, then the monitor state is advanced, and the message can be sent. If not, then an exception is raised. Failure handling is discussed in more detail in Chapter 6.

4.8 Session API

In this section, we detail how users may send messages, and the integration of the inter-role co-operative scheduling mechanism introduced by Neykova and Yoshida [52].

4.8.1 Send

A user may send a monitored session message by calling the `conversation:send(ConvKey, Recipients, MessageName, Values)` function.

In order to send a message, four pieces of information are required:

1. A `ConvKey`, describing the role the actor is playing in the session.
2. A list of recipients
3. A message name
4. A list of values

Recall that the monitoring process is *external* to the logic process. In order to send a session message, the `conversation:send/4` function calls the `actor_monitor` process associated with the logic process. Also recall that a `ConvKey` contains the role name and session ID, which uniquely identify a monitor.

The actor monitor process begins by retrieving the appropriate monitor using the role name and session ID. In order to verify that sending a message is permitted by a monitor, the algorithm

begins by checking whether the root monitor—that is, the monitor with ID 0—has a matching send transition from the current state. In order to do so, the algorithm retrieves the list of transitions from the current state: in the case of send and receive transitions, the interaction type (in this case, send) is checked against the transition type, and the `check_send` and `check_receive` functions return true if the message and interaction type match the transition. In the case of a `par` transition, the nested FSMs are checked recursively.

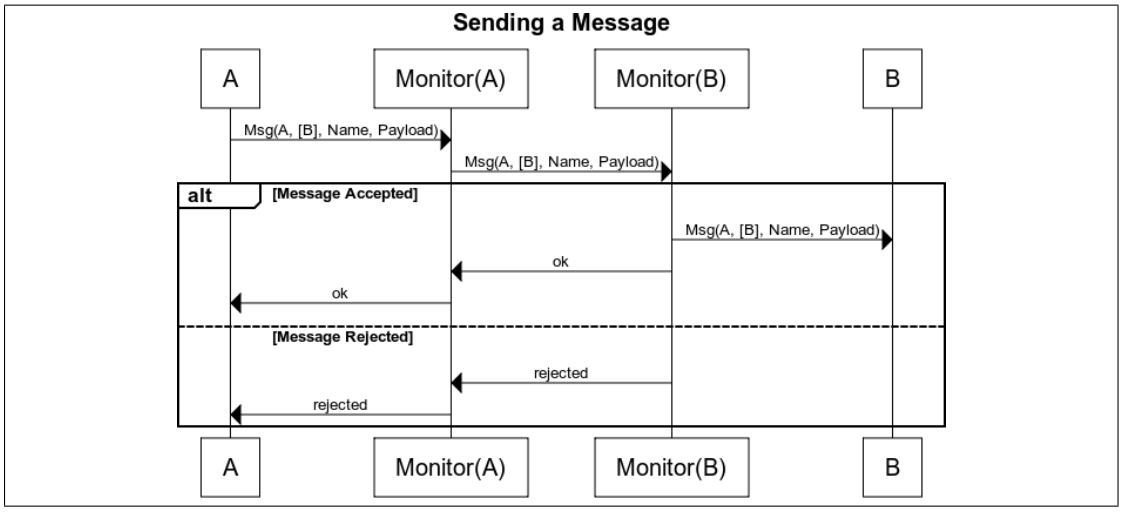


Figure 4.9: Internal messages sent when delivering a message

Figure 4.9 shows the process of delivering a message. Suppose an an actor `A` wishes to send a message with name `Name` and payload `Payload` to an actor `B`. Firstly, `A` makes a blocking synchronous call to the monitor process to ascertain whether the message can be sent.

If so, then the monitor process for `A` resolves the PID for the monitor process of `B` using the internal routing table, and makes a synchronous call to the monitor to check whether it will accept the message. If so, the monitor for `B` will deliver the message to `B` and return `ok` to the monitor for `A`, which will in turn return `ok` to the process.

If either monitor fails, then `rejected` is returned to the actor, which throws an exception. *The default behaviour, should a monitor reject a message, is to throw an exception. Such behaviour is consistent with the Erlang design ideology of letting a process fail if its behaviour deviates from a specification, and logging the error in order to allow the failure to be investigated at a later stage*. In such a case, the failure is detected and handled, as described in Chapter 6. In the unlikely event that a user anticipates a monitor failure, the exception can be caught and handled.

4.8.2 Become

In the session actor model, actors may fulfil multiple roles in multiple protocols. Importantly, however, only one role is 'active' at any one time, due to the single-threaded nature of the actor model. Importantly, it should be possible to switch between roles: a message received in one session should be able to trigger the sending of a message in a different session, for example.

Neykova and Yoshida [52] introduce a construct, `become`, which allows actors to switch between roles. In the original Python implementation of multiparty session actors, as discussed in Section 3.2.1, actors could only partake in a single instance of a session at a time. Consequently, switching to another role was an unambiguous process.

In the Erlang setting, we allow multiple session instances, as is necessary for server applications. Unfortunately, however, a consequence of this is that the co-operative role scheduling becomes ambiguous: which instance should be switched to?

Our solution is to allow session instances to be *registered*, using a unique key. This is achieved by calling the `register_conversation` function:

```
register_conversation(RegKey, ConvKey={ProtocolName, RoleName, ConvID, MonitorPID}) ->
  actor_monitor:register_become(MonitorPID, RegKey, ProtocolName, RoleName, ConvID).
```

In order to switch to a registered session, a user uses the `conversation:become/5` API function, specifying the key (`RegKey`), along with the role name the actor should play, and an operation and arguments.

```
become(ConvKey={_, _, _, MonitorPID}, RegKey, RoleName, Operation, Arguments) ->
  actor_monitor:become(MonitorPID, RegAtom, RoleName, Operation, Arguments).
```

Should `RegKey` be a valid key, with the actor registered to play `RoleName` in the associated session, a message is sent to the user process which invokes the `ssactor_become` callback.

```
ssactor_become(ProtocolName, RoleName, Operation, Arguments, ConvKey, State) ->
  ...
  {ok, State}.
```

In this callback, the given `ConvKey` allows the actor to communicate as the role `RoleName` in the conversation registered to `RegName`. The `Operation` and `Arguments` allow data to be shared between the two sessions.

The whole process is as follows:

1. Upon session initiation, the user calls `conversation:register_conversation/2` to register the session with a key.

2. The user calls the `actor_monitor` process, which associates the key with the session ID.
3. When a user wishes to transition to another role, they call `conversation:become` function, which sends a message to the `actor_monitor` process.
4. If the key is registered, and the actor is registered to play the role `RoleName` in the session, then a message is sent to the logic process implementing the `ssa_gen_server` behaviour, which then invokes the `ssactor_become` callback.
5. The `ssactor_become` callback is invoked with the `ConvKey` allowing the actor to play the given role in the specified session.

In this chapter, we have described how communication in Erlang applications can be monitored using multiparty session types. A session is initiated, with actors invited to fulfil roles within the session. Actors are able to partake in an arbitrary number of session instances, and are registered to take part in sessions via a configuration file. Much like with standard actors implementing the `gen_server` behaviour, lifecycle events in a session (for example, session initiation) invoke callback functions, which allow users to respond to the events. Monitoring is undertaken by an external process: no session message should be delivered without firstly being checked by a monitor.

Some questions remain, however: are there communication patterns which are not immediately expressible using multiparty session types, and how do we detect and handle failures? We explore these questions further in Chapters 5 and 6.

Chapter 5

Encoding Erlang Communication Patterns

In this section, we detail communication patterns that are common in Erlang applications, but difficult to encode using standard multiparty session types. We discuss two main patterns: passing process IDs in order to dynamically introduce processes into a session, and synchronous calls as used by the `gen_server` behaviour. We discuss two methods by which these two patterns may be encoded: subsessions for introducing participants dynamically (as originally observed by Neykova and Yoshida [51]), and additional constructs within the Scribble language for safely encoding blocking synchronous calls.

5.1 Passing Process IDs

In Erlang applications, it is often the case that in order to satisfy a request, a process must send the ID of another process. This is particularly the case when processes act as *registries* for other processes, mapping keys to process IDs.

Consider the case of a chat server, where a user wishes to join a chat room. Each chat room is modelled as a process. In order to join the chat room, the user sends a `find_room` message to the registry, which returns the process ID of the room. The user can then use the process ID to join the room. This interaction pattern is shown in Figure 5.1.

As a first attempt, this interaction pattern could be encoded as a Scribble protocol as in Listing 5.1.

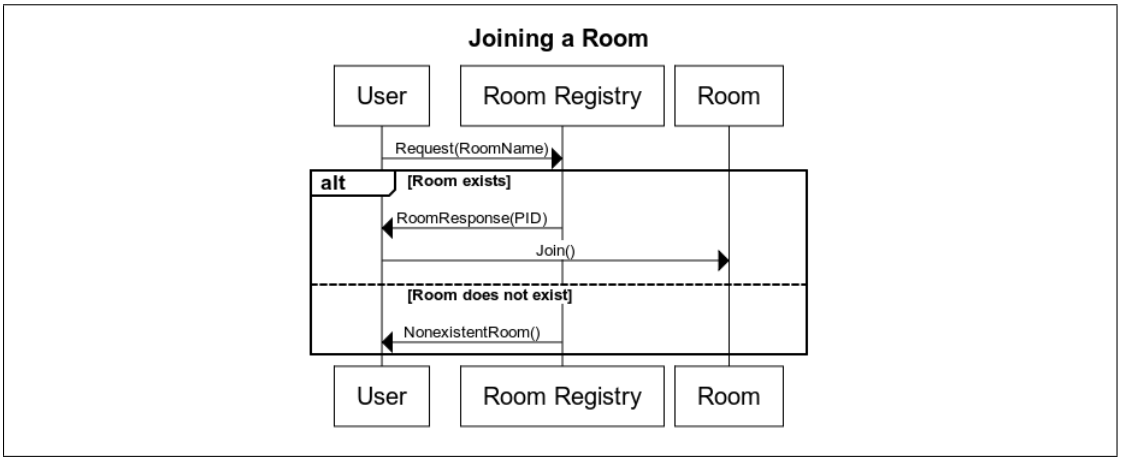


Figure 5.1: Diagram showing chat room joining scenario

Listing 5.1: First attempt at the room joining interaction

```
global protocol JoinRoom(role User, role RoomRegistry, role Room) {
  Request(RoomName) from User to RoomRegistry;
  choice at RoomRegistry {
    RoomResponse(PID) from RoomRegistry to User;
    Join() from User to Room;
    ...
  } or {
    NonexistentRoom() from RoomRegistry to User, Room;
  }
}
```

There are two issues with this presentation. The first issue is that it is assumed that roles are available throughout the entirety of an interaction: that is, roles are associated with endpoints upon session initiation. This is not the case here: we do not know which endpoint will fulfil the `Room` role until the request has been made, and the `RoomRegistry` resolves the room name to a process ID.

The other problem with this presentation lies with the Scribble requirement that the set of roles involved in a choice block must be the same in all branches. As the `Join` message is sent in to the room should the resolution be successful, it is therefore necessary to send a `NonexistentRoom` message (or another message) to the room in the second block, which is clearly nonsensical.

A second approach would be *parameterised* multiparty session types. Parameterised multiparty session types provide a degree of type-dependence, allowing participants to have session types parameterised by indices. The parameterised session framework is particularly useful in the domain of parallel programming and high-performance computing [55]. Such an abstraction would be useful should there be a known, static set of rooms populated at compile time, with a function translating room names into indices. However, in the (likely) case where the number

of rooms is not statically known when the protocol is written, for example when users may create or remove rooms, the parameterised framework becomes too rigid.

5.1.1 On the Merits of Subsessions

One way of encoding such a pattern is through the use of *subsessions* [28]. Subsessions are an abstraction to allow *nested* protocols in session types, allowing increased modularity but also, importantly, for the introduction of participants not involved in the original session. In the subsession model, an initiator process establishes a new session, and can invite participants of the original session by sending *internal* invitations, but also *external* invitations to participants uninvolved in the original session.

Subsession Implementation

The syntax of Scribble allows multiple protocols to be defined in a file.

In order to allow subsessions to be initiated we introduce the `initiates` Scribble construct:

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

The `initiates` construct states that a `Role` initiates a protocol `ProtocolName` with role instantiations `Roles`. Should the subsession complete successfully, then the protocol will proceed with the interactions described in `SuccessBlock`. We defer discussion of the `handle` clauses until Section 6.3.1.

Returning to the chat room example, it is now possible to write the room joining pattern as:

```
global protocol JoinRoom(role User, role RoomRegistry) {
  Request(RoomName) from User to RoomRegistry;
  choice at RoomRegistry {
    RoomResponse(PID) from RoomRegistry to User;
    User initiates RoomActions(User, new Room) {}
  } or {
    NonexistentRoom() from RoomRegistry to User;
  }
}

global protocol InRoom(role User, role Room) {
  Join() from User to Room;
  ...
}
```

We split the protocol into two: the `JoinRoom` protocol involves the `User` and `RoomRegistry` roles and, importantly, *not* the `Room` role. Instead, the `InRoom` protocol is invoked by `User`: `User` is internally invited, whereas `Room` is dynamically introduced using the `new` keyword.

The use of subsessions for dynamic role introduction was discussed briefly by Neykova and Yoshida [51], in the context of an actor performing a Fibonacci protocol. Here, we extend the construct to include the initiator role (to allow the construct to be projected and the creation of the subsession to be monitored) and apply the technique to the passing of process IDs.

Further details of the subsession API, its implementation, and its applicability to failure handling are given in Section 6.3.1.

5.2 Synchronous Calls

A second pattern used heavily in Erlang `gen_server` behaviours is that of synchronous calls. Synchronous calls allow a call to be made to another actor *during the execution of a message handler*; the call can be used to obtain a value which can then be used in the remainder of the handler, or it can be used for synchronisation, for example.

An obvious solution to such an issue, and indeed the primary solution used in the literature, is to represent a synchronous call as two separate synchronous messages. In order to incorporate this, however, handlers previously making use of synchronous calls must be split whenever the call is made, saving the state, and adding a separate handler to be invoked when the return message is received.

To illustrate this, consider the scenario where we have three state cells: `StateCell1`, `StateCell2`, `StateCell3`, and a client, `Client`. Each state cell supports two operations: `get`, which returns the value of the state cell, and `put`, which updates the value of the state cell.

Consider the simple scenario, shown in Figure 5.2, where a client gets the values of two state cells, adds the values, and puts the result into a third state cell.

In a standard `gen_server` program, such an application could be encoded as three synchronous calls, as follows:

```
state_cells(StateCell1PID, StateCell2PID, StateCell3PID) ->
  Res1 = state_cell:get(StateCell1PID),
  Res2 = state_cell:get(StateCell2PID),
  ok = StateCell3:put(Res1 + Res2, StateCell3PID).
```

The `state_cell:get/1` function would be defined as:

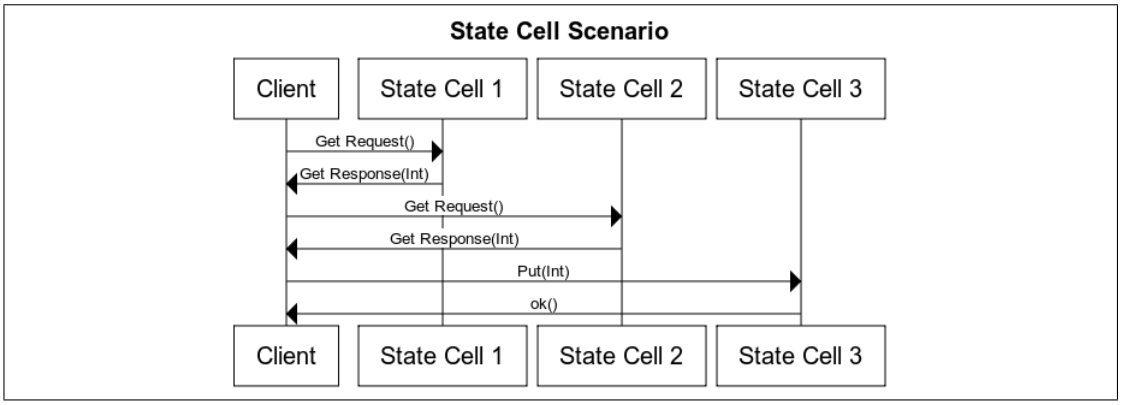


Figure 5.2: State cell scenario

```
get(PID) ->
  gen_server:call(PID, get).
```

In order to translate this pattern, we could write a Scribble protocol using asynchronous messages, as shown in Listing 5.2.

Listing 5.2: Scribble protocol for the state cell scenario encoded as asynchronous messages

```
global protocol SequencedStateCells(role Client, role StateCell1, role StateCell2, role
    StateCellRes) {

    get_request() from Client to StateCell1;
    get_response(Integer) from StateCell1 to Client;

    get_request() from Client to StateCell2;
    get_response(Integer) from StateCell2 to Client;

    put_request(Integer) from Client to StateCellRes;
    put_response(Atom) from StateCellRes to Client;
}
```

The implementation, however, is more verbose than that of the Erlang version. The implementation using asynchronous messages is shown in Listing 5.3.

Listing 5.3: Erlang implementation of state cell scenario using asynchronous messages

```
ssactor_conversation_established(_PN, _RN, _CID, ConvKey, _State) ->
  Res1 = async_state_cell:get_request(ConvKey, "StateCell1"),
  {ok, 0}.

ssactor_handle_message(_, _, _, "StateCell1", "get_response", [Res1], _, ConvKey) ->
  async_state_cell:get_request(ConvKey, "StateCell2"),
  {ok, Res1};
ssactor_handle_message(_, _, _, "StateCell2", "get_response", [Res2], State, ConvKey) ->
  ToPut = State + Res2,
```

```
  async_state_cell:put_request(ConvKey, "StateCellRes", ToPut),
  {ok, ToPut};
ssactor_handle_message(_, _, _, "StateCellRes", "put_response", _, State, _ConvKey) ->
  {ok, State}.
```

The `async_state_cell:get_request/1` and `async_state_cell:put_request/1` functions would be defined as:

```
put_request(ConvKey, StateCellName, NewValue) ->
  conversation:send(ConvKey, [StateCellName], "put_request", [], [NewValue]).

get_request(ConvKey, StateCellName) ->
  conversation:send(ConvKey, [StateCellName], "get_request", [], []).
```

The `async_client:put_response/1` and `async_client:get_response/2` functions are defined similarly.

The implementation, transformed to use asynchronous instead of synchronous messages, is less readable, and more cumbersome to write. In particular, the state must be saved in between each request, and the handler must be split whenever the call response should be received.

5.2.1 Encoding Synchronous Calls

Having a notion of synchronous calls is therefore desirable, wherein an actor may make a call, block until a response is received, and use the result of the call in the remainder of the message handler.

The introduction of blocking calls does however introduce additional constraints. Firstly, a response must always be sent to the actor, otherwise it will block indefinitely (or, in the case of Erlang, until the default timeout has expired). Additionally, the actor which is blocking while waiting for a response should not be involved in any further interactions until a response is received.

Consequently, we wish to disallow possibly-deadlocking interactions such as in Listing 5.4, where in servicing a synchronous call, the state cell requests synchronous confirmation from the client. Naturally, the `confirmation` request cannot be received and processed before the `put` request, meaning that the interaction deadlocks.

Listing 5.4: Problematic call example

```
global protocol SequencedStateCells(role Client, role StateCell) {
    put_request(Integer) from Client to StateCell;
    confirmation_request() from StateCell to Client;
    confirmation_response() from Client to StateCell;
```

```
    put_response(Atom) from StateCell to Client;
}
```

5.2.2 Scribble Modifications

We introduce a construct, `call`, which signifies a synchronous call.

```
call MessageName(Payloads) returning Payload from SenderRole to ReceiverRole {
    Interactions }
```

The `call` construct represents a synchronous call with name `MessageName` and payload types `Payloads` from `SenderRole` to `ReceiverRole`, returning a value of type `Payload`. In order to service the synchronous call, the receiver role may perform additional interactions `Interactions`.

Importantly, the call construct is subject to the following constraints:

1. A participant cannot call itself.
2. No interaction in the `Interactions` block may involve `SenderRole`.
3. Should a `call` block be contained in a `par` block, no other `par` block may involve `SenderRole` or `ReceiverRole`.
4. The `Interactions` block should contain no `rec` blocks or `continue` statements.

In terms of Scribble projections, the `call` construct is projected as:

- At the caller, `send_call_request MessageName(Payloads) to ReceiverRole`, the projection of `Interactions`, and `receive_call_response MessageName(Payload) from ReceiverRole`.
- At the receiver, `receive_call_request MessageName(Payloads) from SenderRole`, the projection of `Interactions`, and `send_call_response MessageName(Payloads) to SenderRole`

Informally, the projection of a `call` block involves two types of message: a call request, and a call response. At the caller, the projection of `Interactions` will be empty, as the role is not permitted to be involved in any of the internal interactions.

Figure 5.3 shows the syntax of global types with synchronous calls, building on the syntax of global types described by Bettini et al. [9], which is the most widely-used formation of global multiparty session types in the literature.

The type $p \to q : \langle\langle G_{interactions} \rangle\rangle_{S_{req}, S_{ret}}.G$ denotes that participant p makes a synchronous call of type S_{req} to participant q, with the synchronous call executing interactions $G_{interactions}$ before returning a value of type S_{ret} and continuing as G. The remainder of the types are standard.

G	$::=$	$p \to \Pi : \langle s \rangle . G'$	Send message
	$\mid$	$p \to \Pi : \langle \{ l_i : G_i \}_{i \in I}$	Send branch selection
	$\mid$	$\mu \mathbf{t} . G$	Recursion
	$\mid$	$\mathbf{t}$	Recursion variable
	$\mid$	**end**	End
	$\mid$	$p \to q : \langle \langle G_{interactions} \rangle \rangle_{S_{req}, S_{ret}} . G$	Synchronous call
S	$::=$	bool |int |...	Ground types

Figure 5.3: Syntax of global types with synchronous calls

As an example, consider the following scenario, where a client makes a request to a state cell, which subsequently makes a request to a persistent store:

```
global protocol PersistentStateCell(role Client, role StateCell, role Database) {
  call get() returning String from Client to StateCell {
    call select() returning Atom from StateCell to Database;
  }
}
```

The global type would be:

$$\texttt{Client} \to \texttt{StateCell} \langle \langle \texttt{StateCell} \to \texttt{Database} \langle \langle \rangle \rangle_{(),int} \rangle \rangle_{(),int} . \mathbf{end}$$

T	$::=$	$!\langle \Pi, S \rangle ; T$	Send
	$\mid$	$?(\mathrm{p}, S) ; T$	Receive
	$\mid$	$\oplus \langle \Pi, \{ l_i : T_i \}_{i \in I} \rangle$	Selection
	$\mid$	$\&(\mathrm{p}, \{ l_i : T_i \}_{i \in I})$	Branching
	$\mid$	$\mu \mathbf{t} . T$	Recursion
	$\mid$	$\mathbf{t}$	Recursion variable
	$\mid$	$!?\langle p, S \rangle ; T$	Send call request
	$\mid$	$??(p, S) ; T$	Receive call request
	$\mid$	$!!\langle p, S \rangle ; T$	Send call response
	$\mid$	$?!(p, S) ; T$	Receive call response
	$\mid$	end	End

Figure 5.4: Syntax of local types with synchronous calls

Figure 5.4 shows the syntax of local types. The majority of types are standard, but with the addition of types for sending and receiving call requests and responses. Note that calls are between only two participants, as opposed to standard multicasting functionality.

The projection function of type G onto a local type for role p, $G \upharpoonright^{env} r$, is inductively defined with respect to a *call environment* which takes the shape $\langle$ *caller role, callee role, return type, continuation* $\rangle$. The projection function is shown in Figure 5.5.

The projection function is partial: a global protocol without a well-formed projection is deemed to be ill-formed. The main departure from most standard presentations of the projection function is the inclusion of the call environment: by making a synchronous call, information about the call is added to the call environment, including the caller, callee, return type, and remainder of the protocol. The projection of a communication action is undefined if it is contained within the current set of callers, and the projection of a recursion variable is undefined if the current call environment is non-empty.

The projection of end depends on the current call environment. Should a call environment be empty, the projection of the protocol has completed, hence the projection is end. Should the call environment not be empty, however, then the projection of end denotes the sending or receiving of a call response, with the remainder of the protocol (as saved in the call environment) projected with the previous call environment.

Proposition 1 (Non-interference)**.** *Given a well-formed global type* $G = p \to q : \langle \langle G_{interactions} \rangle \rangle_{S_{req}, S_{ret}}.G'$, *the projection* $G \upharpoonright^{env} p = !?\langle q, S_{req} \rangle.?!(q, S_{ret}).(G' \upharpoonright^{env} p)$.

Proof. Applying the projection rule $G \upharpoonright^{env} p$ gives us $!?\langle q, S_{req} \rangle; G_{interactions} \upharpoonright^{\langle p,q,S_{ret},G' \rangle \cdot env} p$. The next step is to show that $G_{interactions} \upharpoonright^{\langle p,q,S_{ret},G' \rangle \cdot env} p = ?!(q, S_{ret}).(G' \upharpoonright^{env} p)$.

We firstly show that for an arbitrary global type H, $H \upharpoonright^{env} p = \mathbf{end} \upharpoonright^{env} p$ if $p \in roles(env)$ and $env \neq \emptyset$.

To do so is a straightforward induction: in the SEND and SELECT cases, only the third rule applies, as $p \in roles(env)$ by the assumption, ruling out the first two projection rules, with the required result following from the induction hypothesis. Cases REC and RECVAR are impossible as the environment is nonempty.

$$roles(env) = \{r_{caller} | \langle r_{caller}, r_{callee}, S_{ret}, G_{cont} \rangle \in env\}$$

SEND

$$p \to \Pi : \langle S \rangle . G' \upharpoonright^{env} r = \begin{cases} !\langle \Pi, S \rangle ; (G' \upharpoonright^{env} r) & \text{if } r = p \text{ and } r \notin roles(env) \\ ?(p, S); (G' \upharpoonright^{env} r) & \text{if } r \in \Pi \text{ and } r \notin roles(env) \\ G' \upharpoonright^{env} r & p \neq r \text{ and } r \notin \Pi \end{cases}$$

SELECT

$$p \to \Pi : \{l_i : G_i\}_{i \in I} \upharpoonright^{env} r = \begin{cases} \oplus \langle \Pi, \{l_i : G_i \upharpoonright^{env} r\}_{i \in I} \rangle & \text{if } r = p \text{ and } r \notin roles(env) \\ \&(p, \{l_i : G_i \upharpoonright^{env} r\}_{i \in I}) & \text{if } r \in \Pi \text{ and } r \notin roles(env) \\ & \text{if } r \neq p, r \notin \Pi, \text{ and} \\ G_1 \upharpoonright^{env} r & G_i \upharpoonright^{env} r = G_j \upharpoonright^{env} r \\ & \text{for all } i,j \in I \end{cases}$$

CALL

$$p \to q : \langle \langle G_{interactions} \rangle \rangle_{S_{req}, S_{ret}} . G \upharpoonright^{env} r =$$

$$\begin{cases} !?\langle q, S_{req} \rangle ; (G_{interactions} \upharpoonright^{\langle p,q,S_{ret},G \rangle \cdot env} r) & \text{if } r = p \text{ and } r \notin roles(env) \\ ??(p, S_{req}); (G_{interactions} \upharpoonright^{\langle p,q,S_{ret},G \rangle \cdot env} r) & \text{if } r = q \text{ and } r \notin roles(env) \\ G_{interactions} \upharpoonright^{\langle p,q,S_{ret},G \rangle \cdot env} r & \text{if } p \neq r \text{ and } q \neq r \end{cases}$$

END

$$\mathsf{end} \upharpoonright^{\emptyset} r = \mathsf{end} \qquad \mathsf{end} \upharpoonright^{\langle p,q,S_{ret},G \rangle \cdot env} r = \begin{cases} ?!(q, S_{ret}); G \upharpoonright^{env} r & \text{if } r = p \\ !!(p, S_{ret}); G \upharpoonright^{env} r & \text{if } r = q \\ G \upharpoonright^{env} r & otherwise \end{cases}$$

REC

$$(\mu \mathbf{t}.G) \upharpoonright^{\emptyset} r = (\mu \mathbf{t}.(G \upharpoonright^{\emptyset} r))$$

RECVAR

$$\mathbf{t} \upharpoonright^{\emptyset} r = \mathbf{t}$$

Figure 5.5: Projection function $G \upharpoonright^{env} r$

The more interesting case is CALL:

$$
\begin{aligned}
& p' \to q' \langle \langle H_{int} \rangle \rangle_{S_{req}, S_{ret}} . H \upharpoonright^{env} p \\
= & \text{ (Projection rule 3: projection cases 1 and 2 are not applicable as } p \in env) \\
& H_{int} \upharpoonright^{\langle p', q', S_{ret}, H \rangle \cdot env} p \\
= & \text{ (IH)} \\
& \mathbf{end} \upharpoonright^{\langle p', q', S_{ret}, H \rangle \cdot env} p \\
= & \text{ (Projection rule 3 for end, as caller roles in environment are distinct)} \\
& H \upharpoonright^{env} p \\
= & \text{ (IH)} \\
& \mathbf{end} \upharpoonright^{env} p
\end{aligned}
$$

Finally, we apply this result to $G_{interactions} \upharpoonright^{\langle p, q, S_{ret}, G' \rangle \cdot env} p$, giving $\mathbf{end} \upharpoonright^{\langle p, q, S_{ret}, G' \rangle \cdot env} p$. By the projection rule for **end** we get $?!(q, S_{ret}); G' \upharpoonright^{env} p$ as required.

□

Corollary 1. *Participant p is not involved in any interactions while it is waiting for the response of the synchronous call.*

5.2.3 Implementation

In order to implement the call functionality, much like with `gen_server`, we require an additional callback `ssactor_handle_call` to be implemented. The `ssactor_handle_call` function has takes the same arguments as `ssactor_handle_message`, but includes the ability to reply to the call, by returning a tuple `{reply, Reply, NewState}`.

The call request and call response types are translated into monitor transitions in exactly the same way as sending and receiving messages. More specifically, we add four new transition types: `send_call_request`, `send_call_response`, `recv_call_request`, and `recv_call_response`, which are added in the same ways as standard send and receive transitions.

Additionally, we extend the session API to include a `call` function, identical to the existing `send` function. The `call` function initially calls the monitor, which checks whether the call is allowed. If so, then a synchronous call is made to the remote actor, which checks whether the incoming call is allowed. If so, the the handler function is executed, which can send a reply message. The reply message is sent to the caller.

Returning to our motivating example, it is now possible to write the protocol for the state cell scenario as shown in Listing 5.5.

Listing 5.5: Scribble protocol for synchronous state cells example

```
global protocol SynchronousSequencedStateCells(role Client, role StateCell1,
    role StateCell2, role StateCellRes) {
    call get() returning String from Client to StateCell1;
    call get() returning String from Client to StateCell2;
    call put(Integer) returning Atom from Client to StateCellRes;
}
```

The local protocols for the client and the first state cell are shown in Listings 5.6 and 5.7 respectively.

Listing 5.6: Local projection of synchronous state cells protocol at Client

```
local protocol SynchronousSequencedStateCells at Client(role Client,role StateCell1,role
    StateCell2,role StateCellRes) {
    send_call_request get() to StateCell1;
    receive_call_response get(String) from StateCell1;
    send_call_request get() to StateCell2;
    receive_call_response get(String) from StateCell2;
    send_call_request put(Integer) to StateCellRes;
    receive_call_response put(Atom) from StateCellRes;
}
```

Listing 5.7: Local projection of synchronous state cells protocol at State Cell 1

```
local protocol SynchronousSequencedStateCells at StateCell1(role Client,role StateCell1,
    role StateCell2,role StateCellRes) {
    receive_call_request get() from Client;
    send_call_response get(String) to Client;
}
```

Finally, the implementation of the client is shown in Listing 5.8, with a style much closer to that of Erlang, without the need to save intermediate states and separate the handler.

Listing 5.8: Implementation of the client for the synchronous state cells example

```
ssactor_conversation_established(_PN, _RN, _CID, ConvKey, State) ->
  Res1 = sequenced_state_cell:get(ConvKey, "StateCell1"),
  Res2 = sequenced_state_cell:get(ConvKey, "StateCell2"),
  sequenced_state_cell:put(ConvKey, "StateCellRes", Res1 + Res2),
  {ok, State}.
```

Chapter 6

Failure Detection and Handling

6.1 Overview and Motivation

A common asumption 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 real-world distributed systems, or indeed standalone Erlang applications. As discussed in Chapter 3, an important design pattern in Erlang applications is to arrange processes in *supervision hierarchies*, allowing processes to fail when they encounter an unrecoverable fault, and letting them be restarted by their supervisors.

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 enable error handling and recovery.

6.2 Failure Detection

Once a session has been established, a *process group* is formed, consisting of all participants in the session. Should a process in the group 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.

6.2.1 Push-Based

As described in Section 2.4.1, Erlang provides the ability to *monitor* processes. Consider the case where a process A is monitoring a second process B. If B terminates or becomes unreachable, a message is placed into the mailbox of A, notifying it of the fact that process B has terminated. If a process is set to *trap exits*, then the message is delivered as normal. If not, then A will also terminate.

Push-based failure detection involves detecting when any participant in the session has failed, and ascertaining whether or not the participant is involved in the remainder of the session. The main concepts behind push-based failure detection are shown in Figure 6.1.

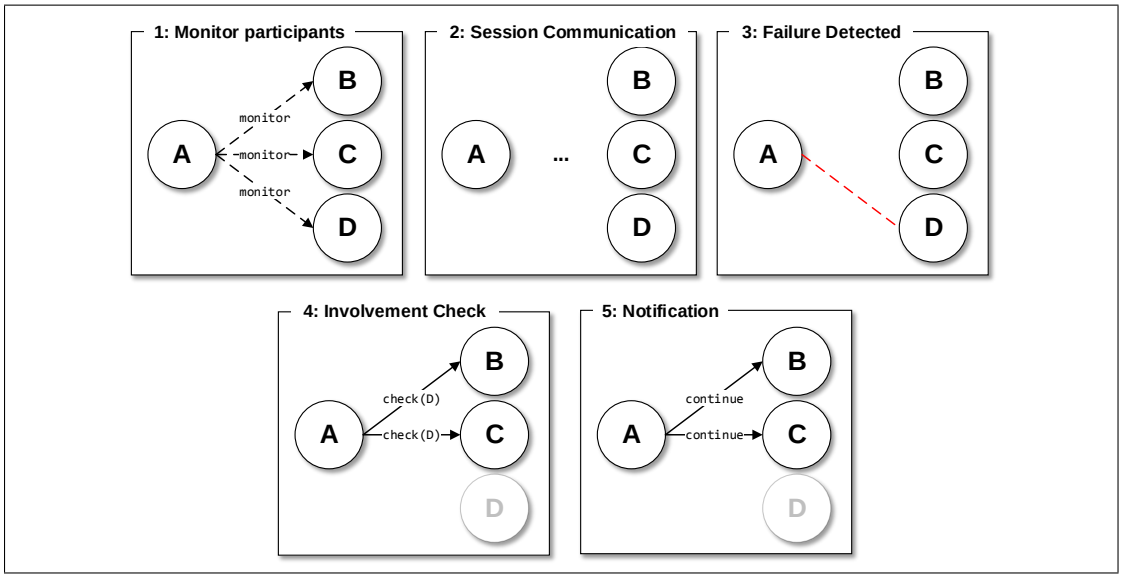


Figure 6.1: Push-Based Failure Detection

Recall that a process, `conversation_instance`, is spawned in order to co-ordinate session lifecycle actions such as invitations and termination. Upon successful initiation of a session, and before notifying participants that the session has been successfully established, the `conversation_instance` monitors all processes in the session and sets itself to trap exits.

Once each process has been monitored by the `conversation_instance` process, the session can begin, and functions as normal. Once a process terminates, the `conversation_instance` process is notified, and begins a safety check to ascertain whether the role played by the terminated process is involved in the remainder of the session.

A role *r* is *involved* in a session if there exists a transition reachable from the current state, where *r* is the sender or receiver in a communication.

In order to determine whether a role is involved in the remainder of a session, the `conversation_instance` process sends a message `check_role_reachable` to all other participant monitor processes. Upon

receiving the message, a monitor process performs reachability analysis on the monitor for the current session, returning the result to the `conversation_instance` process.

Should all responses be `false`, meaning that the role is not involved in the remainder of the session, then the session can safely continue. If any responses are `true`, then it is not possible to safely continue the session. If the session is a subsession, then the subsession will terminate, notifying the parent session as described in Section 6.3. If the session is not a subsession, then the session will end, notifying all participants.

Reachability Analysis

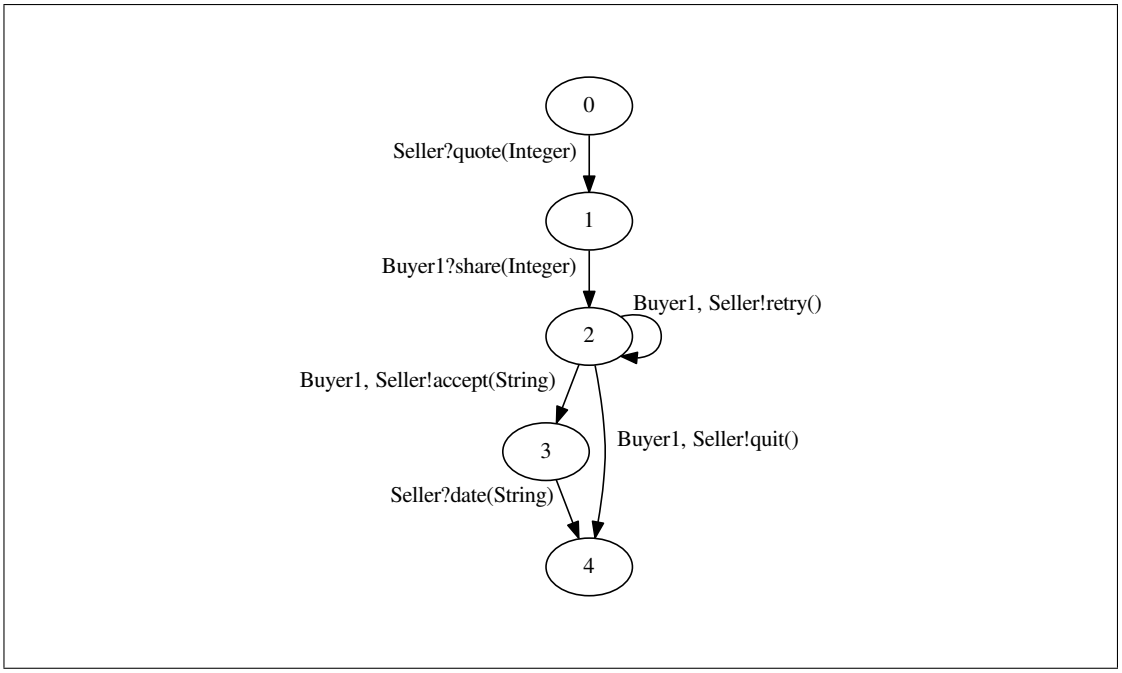


Figure 6.2: Two Buyer Protocol: Monitor for Buyer 2

Recall the monitor for the projection of the two buyer protocol at Buyer 2, shown in Figure 6.2. Different roles will be reachable at different points in the session, and the set of roles involved in paths reachable from each state can be determined by a reachability analysis algorithm.

As an example, the reachability sets for each state are shown in Table 6.1.

The reachability analysis algorithm is shown in Listing 6.1.

State	Reachable Roles
0	[Buyer1, Seller]
1	[Buyer1, Seller]
2	[Buyer1, Seller]
3	[Seller]
4	[]

Table 6.1: Reachability table for two-buyer protocol at Buyer2

Listing 6.1: Reachability Analysis Algorithm

```
generate_reachability_dict(FSM):
  reachable_from_inner(0, FSM, {}, {}, {}, {})

reachable_from_inner(NodeID, FSM, CurrentPathRoles, CurrentPathFSMIDs,
                     ReachableDict, VisitedSet):
  if NodeID in VisitedSet:
    return (CurrentPathRoles, CurrentPathFSMIDs, ReachableDict)

  NewVisitedSet = VisitedSet ∪ {NodeID}

  WorkingRoleSet = {}
  WorkingIDSet = {}
  WorkingDict = ReachableDict

  for each Transition in transitions(NodeID):
    TransitionRoles = roles_in_transition(Transition)
    TransitionFSMs = fsms_in_transition(Transition)
    OutgoingID = outgoing_id(Transition)

    NewCurrentPathRoles = TransitionRoles ∪ CurrentPathRoles
    NewCurrentPathFSMIDs = TransitionFSMIDs ∪ CurrentPathFSMIDs

    (SubpathRoles, SubpathFSMIDs, NewReachableDict) =
      reachable_from_inner(ToID, FSM, NewCurrentPathRoles, NewCurrentPathFSMIDs,
                           WorkingDict, NewVisitedSet)

    WorkingRoleSet = WorkingRoleSet ∪ SubpathRoles
    WorkingIDSet = WorkingIDSet ∪ SubpathFSMIDs
    WorkingDict = NewReachableDict

  NewReachableDict = store(NodeID, (WorkingRoleSet, WorkingIDSet), WorkingDict)
  return (WorkingRoleSet, WorkingIDSet, ReachableDict)
```

The algorithm only needs to be run once, upon monitor generation, and memoises results during the course of execution to avoid unnecessary recomputation. In addition to storing the roles reachable on each subpath, the algorithm also records the IDs of any nested FSMs used to

implement parallel scopes: this is necessary when detecting which roles are reachable in the nested FSMs.

The steps of the algorithm are as follows:

1. Check if the node has already been visited: if so, then return the current paths, FSM IDs, and reachability table.
2. If not, then add the current node to the visited set, initialise three variable `WorkingRoleSet`, `WorkingIDSet`, and `WorkingDict`, and for each outgoing transition:
   (a) Initialise the variables `NewCurrentPathRoles` and `NewCurrentPathFSMIDs` as the union of the roles and nested FSM IDs referenced by the transition respectively
   (b) Recursively calculate the roles and FSM IDs in the subpath
   (c) Update `WorkingRoleSet`, `WorkingIDSet`, and `WorkingDict` with the results of the recursive call
3. Store the results in `WorkingDict`, and return a 3-tuple of `(WorkingRoleSet, WorkingIDSet, WorkingDict)`

Upon termination, the algorithm will return a complete reachability table for the given FSM.

The reachability table takes the form of a map `StateID` $\mapsto$ `(RolesInvolved, FSMsInvolved)`. Checking whether a role is involved at the current point in the session is therefore achieved by a lookup of the current state ID. As the reachability table also contains reachable nested FSM IDs, the process can be applied recursively.

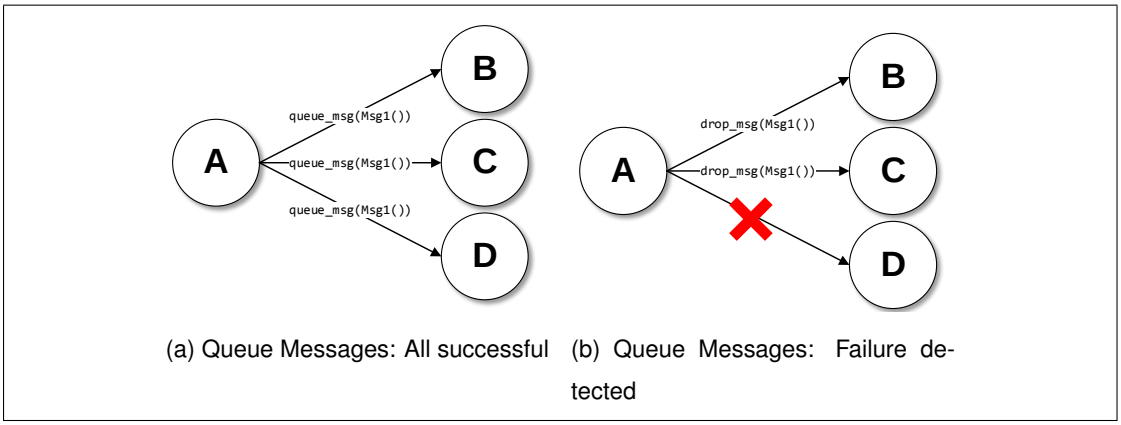
6.2.2 Pull-Based

In contrast to push-based failure detection, which involves processes being notified upon the termination of another process, pull-based failure detection involves ensuring another process is active prior to sending a message.

Multiparty session types allow messages to be sent to multiple participants. Consequently, it is desirable to ensure that messages are only delivered if all processes receiving the message are active. In order to do this, pull-based failure detection makes use of reliable send messages, using a two-phase commit to ensure that firstly, all recipient processes are available, and secondly, 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 6.3a). The result of this call will be one of three things: 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.



(a) Queue Messages: All successful (b) Queue Messages: Failure detected

Figure 6.3: Pull-based failure detection: queue

When a message is sent, it is assigned a unique identifier. Should a message be accepted, it is stored in a table taking the form of a map `MessageID` $\mapsto$ `(ProtocolName, RoleName, ConvID, Message)`. Should all messages be delivered successfully, a second, asynchronous message will be sent to *commit* the message, sending the messages to the actor processes to be handled.

On the other hand, if the a queue message fails for any participant (Figure 6.3b), 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 as an exception. If the failure is due to a process being unreachable, however, it is not possible for the session to continue, and a failure handler is invoked.

A problem with the two-phase commit approach is if a process should terminate after queueing a message but before committing it. A possible solution to this would be to implement a full atomic multicast protocol, which would require all participants sent the message to each other, before passing the message onto the actor for handling. While affording slightly more safety, this method would require more messages to be sent.

6.2.3 Discussion

Push- and pull-based approaches each have advantages and disadvantages. 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 approaches fall short when an actor terminates while processing a message; consider the following protocol:

```
global protocol PullExample(role A, role B, role C) {
  X() from A to B, C;
  Y() from B to A, C;
}
```

Consider the case where the message `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.

Push-based failure detection falls short should a message handler involving the failed role be executed while the safety check is in progress. In the `PullExample` protocol, for example, consider the case where `A` terminates while `B` processes message `A`. Without pull-based detection, there no guarantee that the failure will be detected prior to `Y` being sent to A and C, resulting in a non-atomic send operation where only C receives the message.

Consequently, the 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).

6.3 Failure Handling

Once a failure has been detected, how can it be handled?

In Chapter 3, we discussed various, primarily theoretical, approaches to handling exceptions and failures in session-typed processes. Our setting is largely different: we are working in an environment where not only may exceptions be raised in the case of an error in application logic, but we must be able to take into account the case where a process terminates. In this section, we describe an approach based on subsessions [28], which can handle both application-level exceptions, and failures due to processes terminating during the session.

6.3.1 Subsessions for Failure Handling

In their paper on subsessions, Demangeon and Honda [28] speculate that subsessions could be useful for exception handling. In the setting of Erlang applications, we cannot assume that the processes that fulfil roles persist through the lifetime of a session. Consequently, the modularity

of the subsession abstraction allows us to minimise the amount of time that a participant is involved with a session, and, in the case of failure, allows us to repeat smaller computations.

One approach to failure handling is that of 'role repopulation': upon detection of an actor terminating, and therefore not being able to fulfil the role in the remainder of the session, inviting a separate actor to fulfil the role and continue the protocol. This is difficult for two main reasons: we must have a separate copy of the monitor state independent of the process, which would require either centralised monitors or costly and difficult synchronisation. The other reason is that we would need a method of ensuring that the actor participating in the session begins the computation at the correct point in the protocol.

Separating blocks of computation into subsessions retains the possibility of inviting other actors to repopulate roles where the original participant has terminated. By explicitly demarcating the computation blocks, clean-up operations can be executed in order to compensate for failures, and the protocol can be retried from the beginning without needing to maintain monitor state.

6.3.2 Scribble Constructs for Failure Handling Subsessions

Our approach to failure handling involves making subsessions first-class entities within session types. Consequently, in order to fulfil a session type, an implementation must spawn a subsession when specified by the session type.

Recall the `initiates` construct from Section 5.1.1.

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

The `initiates` construct specifies that a given subsession of protocol `ProtocolName` should be executed. The protocol then proceeds based on the result of the subsession: should the subsession execute successfully, the protocol will proceed as `SuccessBlock`. When raising an exception within an implementation, a user must specify the name of the failure. Should an exception with name `FailureName` be raised, then the protocol will proceed as the `FailureBlock` associated with the name of the failure.

The `initiates` syntax is reminiscent of the `try..in..unless` construct advocated by Benton and Kennedy [8] for exception handling in ML-like programming languages.

Only the subsession initiator is aware that the subsession is to be initiated. For participants involved in the success and failure blocks, an `initiates` block is semantically identical to an external choice by the subsession initiator role: although the choice at the initiator is driven by the result of the subsession, this is transparent to the participant. Consequently, the projection of an `initiates` block for a non-initiator participant is a local choice.

In order to guarantee that protocols are safe—that is, that monitor transitions remain deterministic, and all participants are aware of a choice that has been made—we impose the same restrictions as a global choice block. In the original presentation of multiparty session types, projections were subject to a somewhat restrictive property meaning that it was only possible to define a projection on a choice in which all projections of a choice had the same session type. Consequently, all branches behave the same regardless of the chosen option, in turn avoiding the problem that all participants are notified of the chosen branch. Later work [19] relaxes this constraint subject to *mergeability* conditions. Informally, mergeability conditions enable the projection of branches with different labels to be different. In Scribble, this is realised by requiring the following well-formedness conditions on choices, taken from the Scribble language specification [60]:

> "In a global-choice, of the form `choice at A` $block_1$ `or ... or` $block_n$, the following conditions should be satisfied.
>
> 1. There should be strictly more than one block: $n > 1$.
> 2. For $0 < i \leq n$, in each $block_i$, A should send the first message. All the other participants appearing in the block should appear first as receivers, before possibly be senders.
> 3. Any participant B, different from A, receiving a message in $block_i$, should also be receiving a message in all other blocks. The messages received by B in the other blocks should however be distinct if B's following actions are different.
> 4. The messages that A sends should be different in each block."

By enforcing these restrictions on the success and `handle` branches of an `initiates` block, we ensure that the protocol remains well-formed.

6.3.3 Session API Additions

In order to implement failure-handling subsessions in user code, we provide three additional API functions. The additional functions are shown in Table 6.2.

The three functions `start_subsession`, `subsession_success`, and `subsession_failed` govern the lifecycle events of subsessions. Should it be allowed by the monitor, the `subsession_success` function starts a new subsession by inviting the necessary participants. Inspired by the work of Demangeon and Honda [28], there are two types of invitation: internal, referring to participants active within the current session, and external, which refers to participants outside of the current session. Internal invitations are specified by providing a role name which is present in the current session, whereas an external invitation may either be a role name, or a pair mapping a role name to a process ID, should a particular participant need to be invited.

The `subsession_success` and `subsession_failed` functions are called by a participant within the subsession to signify that either the subsession has succeeded, returning a value, or that the subsession has failed, notifying the initiator of the type of failure that has occurred.

We also require the implementation of three new callback functions:

`ssactor_subsession_complete(SubsessionName, Result, State, ConvKey)`
: Called in the initiator when the subsession for protocol `SubsessionName` completes successfully, returning `Result`.

`ssactor_subsession_failed(SubsessionName, FailureName, State, ConvKey)`
: Called in the initator when the subsession for protocol `SubsessionName` terminates abnormally, for reason `FailureName`.

`ssactor_subsession_setup_failed(SubsessionName, Reason, State, ConvKey)`
: Called in the initiator when it was not possible to start the subsession, for example, if it was not possible to find an actor to fulfil a role.

When implemented, all callbacks should return `{ok, NewState}`, where `NewState` is the updated actor state.

Should a subsession terminate due to a participant becoming unavailable, as detected by the failure handling mechanisms, the subsession will exit with the reason `ParticipantOffline`. This can be detected in a `handle` block, and if possible, the subsession can be retried.

6.3.4 Case Study: BPMN Travel Booking Scenario

The Business Model Process and Notation (BPMN)[1] modelling language is a modelling language for describing business processes. Describing business processes naturally shares several similarities with describing protocols: BPMN provides several constructs such as parallel composition and choice which map directly onto those of multiparty session types.

In this section, we describe and implement a travel booking scenario from the BPMN 2.0 examples guide [14], which includes error handling. The scenario proceeds as follows:

1. The customer sends a request to travel to a destination between two dates.
2. The travel agent forwards the request to a flight booking service and a hotel booking service.
3. The flight booking service and hotel booking service return flights and hotels matching the booking criteria to the travel agent.

[1] http://www.omg.org/spec/BPMN/2.0/

Function Name	Arguments	Description
`start_subsession(ConvKey, ProtocolName, InternalInvitations, ExternalInvitations)`	`ConvKey`: The `ConvKey` for the initiator role and session. `ProtocolName`: The name of the protocol to spawn as a subsession. `InternalInvitations`: A list of roles which should be invited from the current session. `ExternalInvitations`: A list of roles which should be invited from outside of the current session.	Starts a new subsession.
`subsession_failed(ConvKey, FailureName)`	`ConvKey`: The `ConvKey` for a role within the subsession. `FailureName`: The name of the failure that occurred.	Terminates the current subsession, notifying the parent session of the failure.
`subsession_complete(ConvKey, Result)`	`ConvKey`: The `ConvKey` for a role within the subsession. `Result`: A result value to be returned to the initiator subsession.	Terminates the current subsession, notifying the parent session of the success, and returning a value.

Table 6.2: Subsession API Functions

4. The travel agent sends the alternatives to the customer.

5. The customer can either select the desired flights and hotels, and proceed with the booking, or cancel the request. If no response is received after 24 hours, then the request is cancelled.

6. If the customer chooses to continue, the travel agent requests credit card information from the customer. If no response is received after 24 hours, then the request is cancelled.

7. The travel agent sends requests to the flight booking service and hotel booking service to book the requests, which respond with a confirmation if the booking is successful. Should either booking be unsuccessful, *both* the flight and hotel bookings should be cancelled, and the process retried up to a set number of times.

8. Once the booking is made, the travel agent should attempt to charge the credit card. Should this be successful, then the customer should be notified. Should it fail, the booking should be cancelled, the customer should be asked for another credit card.

BPMN processes consist of a series of *Activities* and *Tasks*. An activity is a generic name for a unit of work to be performed, and a task is a core, atomic unit of work in a process. An Activity can consist of multiple sub-tasks, and is indicated by a rounded rectangle. A task is a rounded rectangle containing text.

Messages are denoted by envelopes: a black envelope denotes sending a message, and a white envelope denotes receiving a message; the sending or receiving of messages can also be associated with a task.

Diamond-shaped elements are known as gateways, and handle control flow (for example, branching and joining). A diamond with a + symbol indicates that the branches are executed in parallel.

Finally, and more interestingly, BPMN includes functionality for error handling. There exist two constructs: *compensation handlers* (denoted by a rewind symbol) which can be used to undo the effects of an action, and *error handlers*, which perform error handling. Error handlers are Activity blocks with a dotted line, and begin with a lightning symbol.

In order to implement BPMN-style error handling, we can spawn the activity which can throw an error as a subsession, with the communication actions performed by the error handler in a `handle` block.

Figure 6.4 shows the travel booking process as a BPMN diagram.

The translation to a Scribble protocol, without any error handling, is shown in Listing 6.2.

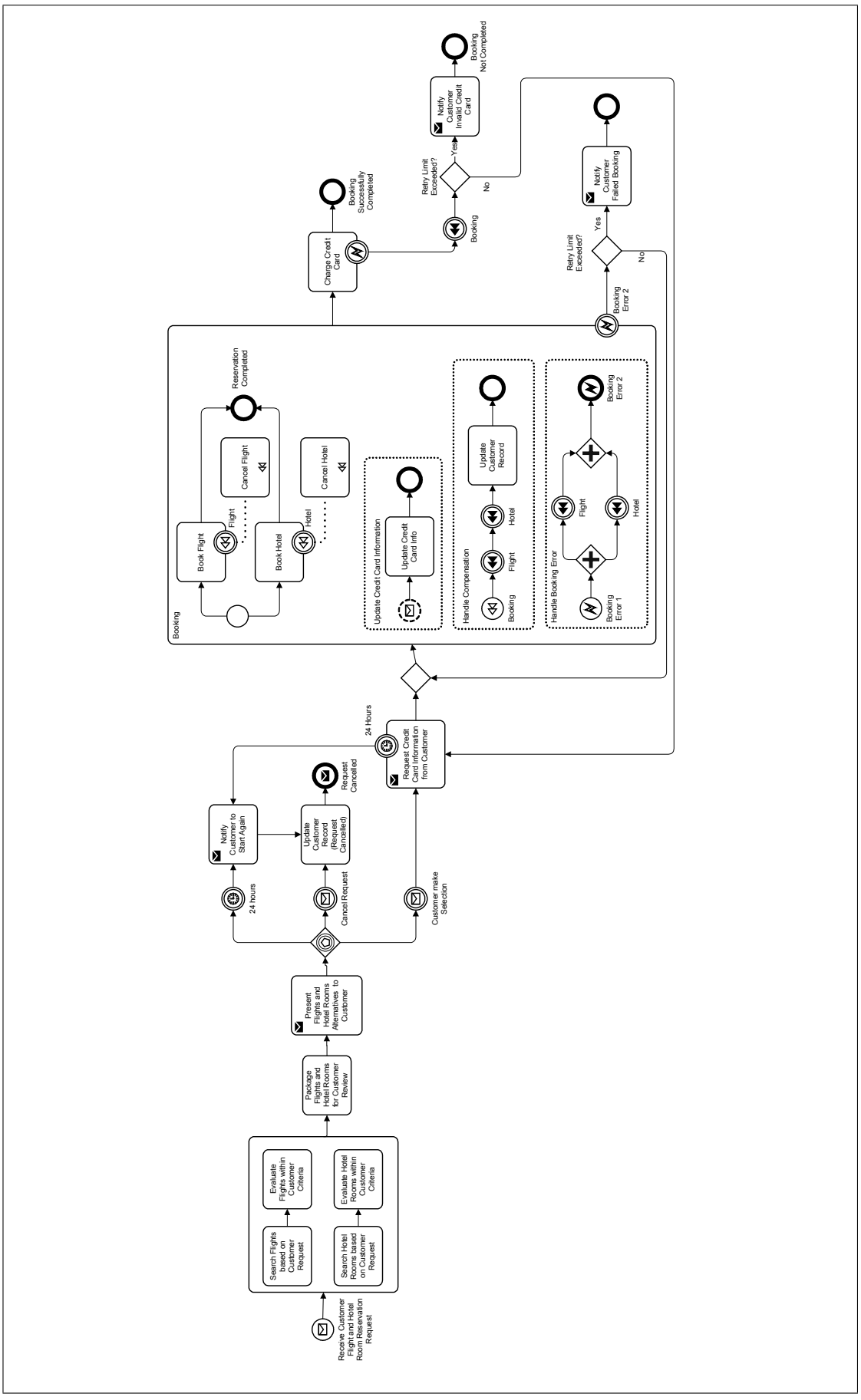


Figure 6.4: Travel Booking Scenario BPMN Diagram [14]

Listing 6.2: Scribble Protocol for Travel Booking Scenario

```
global protocol BookTravel(role TravelAgent, role Customer,
    role FlightBookingService, role HotelBookingService, role PaymentProcessor) {

    // 1) Receive customer request
    customerRequest(RequestInfo) from Customer to TravelAgent;

    // 2) In parallel, search for flights and hotels
    par {
        flightInfoRequest(RequestInfo) from TravelAgent to FlightBookingService;
        flightInfoResponse(OutwardFlights, ReturnFlights) from FlightBookingService to
            TravelAgent;
    } and {
        hotelInfoRequest(RequestInfo) from TravelAgent to HotelBookingService;
        hotelInfoResponse(HotelDetails) from HotelBookingService to TravelAgent;
    }

    // Send details to customer
    customerResponse(OutwardFlights, ReturnFlights, HotelDetails) from TravelAgent to
        Customer;

    // Customer can choose to proceed, or cancel the booking and start again.
    choice at Customer {
        proceedWithBooking(OutwardFlight, ReturnFlight, HotelName) from Customer to
            TravelAgent;
        // At this point, the customer has asked for the booking.
        ccInfoRequest() from TravelAgent to Customer;
           ccInfoResponse(CCNumber, ExpiryDate, CVC) from Customer to TravelAgent;
        par {
            bookFlight(Name, OutwardFlight, ReturnFlight) from TravelAgent to
                FlightBookingService;
            flightBookingConfirmation() from FlightBookingService to TravelAgent;
        } and {
            bookHotel(Name, HotelName, CheckInDate, CheckOutDate) from TravelAgent to
                HotelBookingService;
            hotelBookingConfirmation() from HotelBookingService to TravelAgent;
        }
        processPayment(CCNumber, ExpiryDate, CVC, Money) from TravelAgent to
            PaymentProcessor;
        paymentConfirmation() from PaymentProcessor to TravelAgent;
        confirmation() from TravelAgent to Customer;
    } or {
        cancelBooking() from Customer to TravelAgent, FlightBookingService,
            HotelBookingService, PaymentProcessor;
    }
}
```

Points of Failure

As specified by the scenario, there are five main failure points in the application logic:

1. Customer fails to respond to the packages returned by the flight search
2. Customer fails to respond to the request for credit card information
3. Failure when booking flight
4. Failure when booking hotel
5. Failure when charging credit card

The five points above arise due to specific logic errors. Additionally, as we are working with an Erlang system wherein developers are encouraged to let processes fail, and session actors can also partake in other sessions, we must also consider that processes can fail at any time during the session. We do not consider timing constraints such as points 1 and 2, but these could be addressed by timed session types [10, 13, 54], and handled in an identical manner.

Partitioning the Protocol

Conceptually, we can partition the protocol into three separate phases:

Request Phase
: The customer makes the initial request, the travel agent requests appropriate flights and hotels from providers, and sends the flights and hotels to the user.

Booking Phase
: The customer accepts a selection, sending credit card details to the travel agent. The travel agent proceeds to make the booking.

Payment Phase
: Having made the booking, the travel agent contacts the payment processing service, and attempts to make the payment. If payment processing fails, the booking is cancelled, and the booking stage must be repeated with new information.

Failures may arise when booking flights and hotels in the booking phase, and when charging a credit card in the payment phase. As a result, we split the outer protocols into three separate protocols: `BookTravel`, `PerformBooking`, and `PerformPayment`, and a further subsession `CancelBooking` which notifies the hotel- and flight booking services that the booking should be cancelled.

```
global protocol BookTravel(role TravelAgent, role Customer,
    role FlightBookingService, role HotelBookingService, role PaymentProcessor) {
    // 1) Receive customer request
    customerRequest(RequestInfo) from Customer to TravelAgent;
    // 2) In parallel, search for flights and hotels
    par {
```

```
        flightInfoRequest(RequestInfo) from TravelAgent to FlightBookingService;
        flightInfoResponse(OutwardFlights, ReturnFlights) from FlightBookingService to
            TravelAgent;
    } and {
        hotelInfoRequest(RequestInfo) from TravelAgent to HotelBookingService;
        hotelInfoResponse(HotelDetails) from HotelBookingService to TravelAgent;
    }
    // Send details to customer
    customerResponse(OutwardFlights, ReturnFlights, HotelDetails) from TravelAgent to
        Customer;
    // Customer can choose to proceed, or cancel the booking and start again.
    choice at Customer {
        proceedWithBooking(OutwardFlight, ReturnFlight, HotelName) from Customer to
            TravelAgent;
        rec BookingLoop {
            TravelAgent initiates PerformBooking(TravelAgent, Customer, new
                FlightBookingService, new HotelBookingService) {
                // Success!
                TravelAgent initiates PerformPayment(TravelAgent, PaymentProcessor) {
                    confirmation() from TravelAgent to Customer;
                } handle (PaymentFailure) {
                    paymentFail() from TravelAgent to Customer;
                    continue BookingLoop;
                }
            } handle (BookingFailure) {
                bookingFail() from TravelAgent to Customer;
                TravelAgent initiates CancelBookings(TravelAgent, new
                    FlightBookingService, new HotelBookingService) {
                    continue BookingLoop;
                }
            }
        }
    } or {
        cancelBooking() from Customer to TravelAgent;
    }
}
```

```
global protocol PerformBooking(role TravelAgent, role Customer, role FlightBookingService
    , role HotelBookingService) {
        // At this point, the customer has asked for the booking.
        ccInfoRequest() from TravelAgent to Customer;
        ccInfoResponse(CCNumber, ExpiryDate, CVC) from Customer to TravelAgent;
        par {
            bookFlight(Name, OutwardFlight, ReturnFlight) from TravelAgent to
                FlightBookingService;
            flightBookingConfirmation() from FlightBookingService to TravelAgent;
        } and {
            bookHotel(Name, HotelName, CheckInDate, CheckOutDate) from TravelAgent to
                HotelBookingService;
            hotelBookingConfirmation() from HotelBookingService to TravelAgent;
        }
}
```

```
global protocol PerformPayment(role TravelAgent, role PaymentProcessor) {
    processPayment(CCNumber, ExpiryDate, CVC, Money) from TravelAgent to PaymentProcessor
        ;
    paymentConfirmation() from PaymentProcessor to TravelAgent;
}
```

```
global protocol CancelBookings(role TravelAgent, role FlightBookingService, role
    HotelBookingService) {
    par {
        cancelFlightBooking() from TravelAgent to FlightBookingService;
    } and {
        cancelHotelBooking() from TravelAgent to HotelBookingService;
    }
}
```

In the `BookTravel` protocol, the travel agent receives the customer request, requests hotel and flight information in parallel, and sends the details back the customer as before. Should the customer choose to proceed, we then proceed to spawn the `PerformBooking` subsession. The `PerformBooking` subsession can fail, for example if the flights or hotels are no longer available: in this case, the relevant participant would end the session, with the reason `BookingFailure`. Note that the throwing of the exception can happen at any point in the subsession, and therefore does not appear in the type. Should the session fail, the protocol will progress to the `BookingFailure` branch, initiating the `CancelBookings` subsession to cancel the bookings, before allowing the booking to be retried.

Note, however, the additional message `bookingFail` that must be sent to the customer should the booking fail. Recall that an `initiates` block is projected as a choice block at non-initiator roles, and therefore the success and `handle` branches must adhere to the safety conditions of choice blocks, meaning that the set of roles in each must be identical.

Should the booking session succeed, the `PerformPayment` session will be initiated, which processes the payment. Should this be successful, then a confirmation will be sent to the user and the protocol will have completed. Should the payment have failed, the customer will be notified and the process will repeat.

6.4 Implementation

The implementation of the scenario is standard: we create `ssa_gen_server` implementations for the travel agent, customer, flight booking service, and hotel booking service, and register the actors for their roles in a configuration file.

The more interesting parts of the implementation are based around failure-handling subsessions. As an example, the actor implementing the `TravelAgent` role must initiate the `PerformBooking` subsession upon receiving a `proceedWithBooking` message from the customer. Should the subsession fail, the `bookingFail` message should be sent to the client, and the `CancelBookings` subsession should be initiated. Both functions are shown in Listing 6.3.

Listing 6.3: Implementation of subsession operations in TravelAgent role

```
ssactor_subsession_complete("PerformBooking", _, State, ConvKey) ->
  error_logger:info_msg("PerformBooking complete~n"),
  conversation:start_subsession(ConvKey, "PerformPayment", ["TravelAgent"],
                                ["PaymentProcessor"]),
  {ok, State};
ssactor_subsession_complete("PerformPayment", _, State, ConvKey) ->
  travel_customer_split:confirmation(ConvKey),
  {ok, State};
ssactor_subsession_complete("CancelBookings", _, State, ConvKey) ->
  conversation:start_subsession(ConvKey, "PerformBooking", ["TravelAgent", "Customer"],
  ["FlightBookingService", "HotelBookingService"]),
  {ok, State}.

ssactor_subsession_failed("PerformBooking", "BookingFailure", State, ConvKey) ->
  travel_customer_split:booking_failed(ConvKey),
  conversation:start_subsession(ConvKey, "CancelBookings", ["TravelAgent"],
                                ["FlightBookingService", "HotelBookingService"]),

  {ok, State}; ...
```

Should the `PerformBooking` subsession succeed, the `PerformPayment` subsession will be started. Should the `PerformPayment` booking be successful, a confirmation is sent. If the `PerformBooking` subsession is unsuccessful, however, then the `ssactor_subsession_failed` callback will be invoked, the `CancelBooking` subsession will be initiated, and upon its completion, the `PerformBooking` subsession will be restarted. Should there need to be a limit on the number of times that the subsession is invoked (in order to prevent infinite loops, for example), this could easily be recorded in the state.

Finally, Figure 6.5 shows the monitor for the `TravelAgent` role, omitting the nested FSMs representing the parallel scopes.

Note in particular the transitions for initiating subsessions: we require one transition to check that the subsession has been initiated, and multiple transitions dictate the next state based on the result of the subsession.

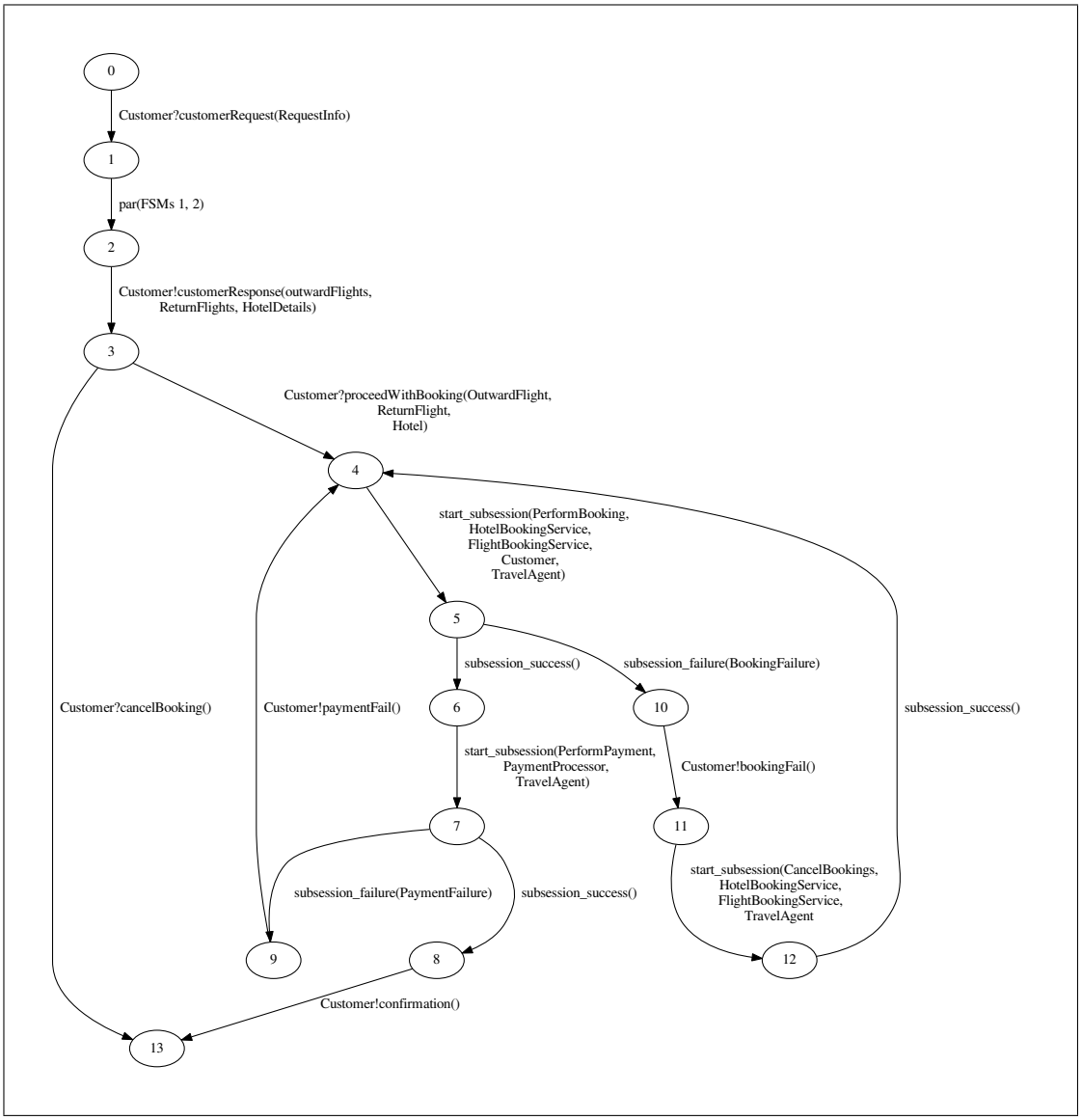


Figure 6.5: Monitor for `BookTravel` protocol, projected at `TravelAgent`

Chapter 7

Evaluation

7.1 Case Studies

7.1.1 DNS Server

The Domain Name System (DNS) provides a hierarchical service to resolve domain names to IP addresses. While the DNS protocol itself is simple—a DNS request is a simple request-response interaction, where the complexity instead lies in the packet itself—processing a DNS request in an actor-based language gives rise to interesting communication patterns.

Overview of `erlang-dns`

The `erlang-dns` project[1] is a DNS server based on Erlang/OTP design principles. Figure 7.1 shows the supervision hierarchy of the DNS server. The system is structured as a typical Erlang/OTP application: a root supervisor `ed_sup` supervises the entire system. The `ed_zone_sup` supervisor supervises the zone subsystem, in particular the `ed_zone_registry_server` which acts as a registry for individual zone processes, mapping domain names to their resolver processes; and the `ed_zone_data_sup` supervisor supervises individual instances of zone data servers. Zone data servers (shown in the diagram as '.com' and '.net') are processes which map domain names to IP addresses.

The application is entirely request-driven: it is a server in which communication amongst participants occurs purely in order to satisfy requests.

Upon system initiation, the `ed_udp_server` process opens a UDP acceptor socket, and listens for incoming requests. When a query is received, a `ed_udp_handler_server` process is spawned to

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

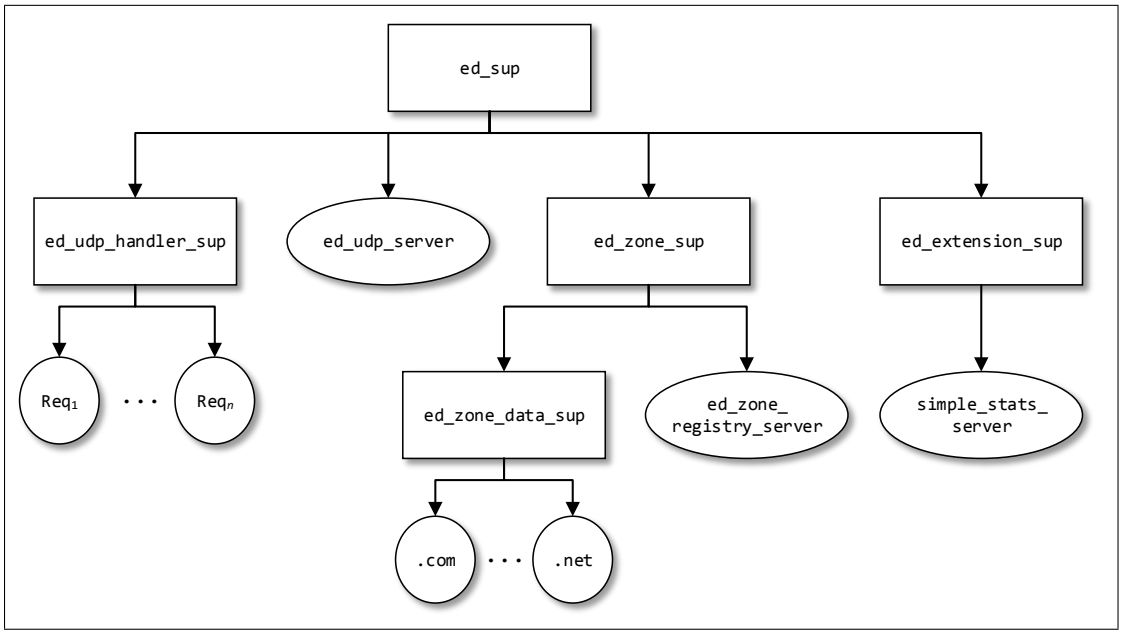


Figure 7.1: `erlang-dns` Supervision Hierarchy

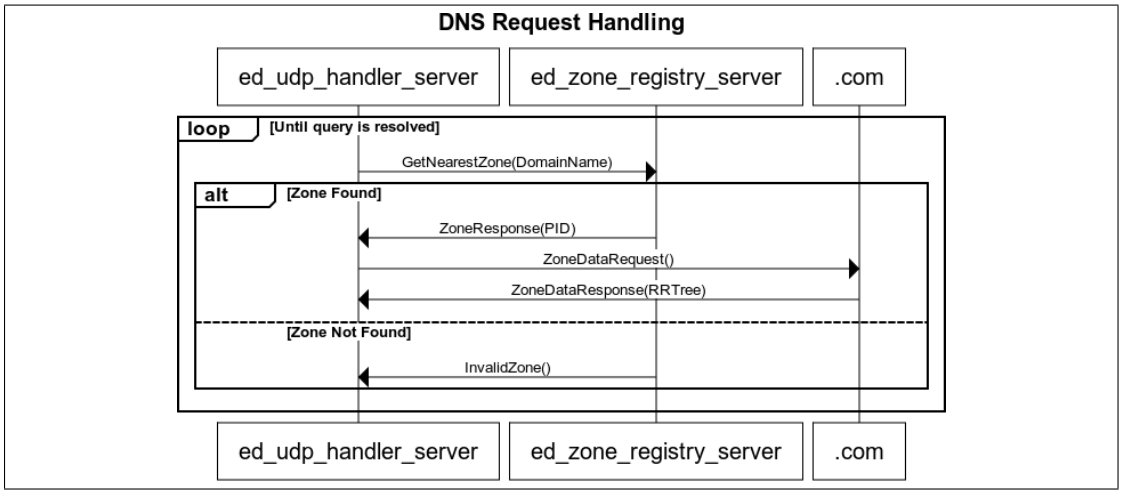


Figure 7.2: Messages sent when fulfilling a DNS lookup request

handle the request.

Figure 7.2 shows the messages sent when processing a DNS query. The UDP handler server firstly contacts the zone registry server to ascertain whether or not the zone is available. If not, then the server returns `InvalidZone()`, an 'invalid zone' DNS packet is sent to the host making the request, and the session terminates. If the zone is found, however, then the zone registry returns the PID of the zone data server. The zone data server is then contacted, returning the information about the zone. At this point, if the IP address can be resolved from the request, then it is returned to the user. Alternatively, it may be necessary to perform a recursive lookup, at which point the protocol repeats.

Encoding the Protocol

We begin the session at the point after a UDP handler server has been spawned to handle the request. We have three roles: `UDPHandlerServer` (fulfilled by the `ed_udp_handler_server` instance initiating the session), `DNSZoneRegServer` (fulfilled by the `ed_zone_reg` process), and `DNSZoneDataServer` (eventually fulfilled by the zone data server which is able to handle the request).

Note that we cannot fulfil the `DNSZoneDataServer` role at the beginning of the session, as the actor to invite depends on the result of the the request to the zone registry. Consequently, we write the main body of the protocol `HandleDNSRequest` involving `UDPHandlerServer` and `DNSZoneDataServer`, and a subprotocol `GetZoneData` involving `UDPHandlerServer` and `DNSZoneDataServer`.

Listing 7.1: Scribble protocols for DNS server

```
global protocol HandleDNSRequest(role UDPHandlerServer, role DNSZoneRegServer) {
  rec QueryResolution {
    // Request the nearest zone
    FindNearestZone(DomainName) from UDPHandlerServer to DNSZoneRegServer;

    // DNSZoneRegServer checks whether it has the data
    choice at DNSZoneRegServer {
       // If we do, then get the PID for the zone data server
       ZoneResponse(ZonePID) from DNSZoneRegServer to UDPHandlerServer;
       // Introduce the zone data server using a subsession
       UDPHandlerServer initiates GetZoneData(UDPHandlerServer, new DNSZoneDataServer) {
         // Now we've done that, we can do possible recursive lookups
         continue QueryResolution;
       }
    } or {
      InvalidZone() from DNSZoneRegServer to UDPHandlerServer;
    }
  }
}
```

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

Listing 7.1 shows the Scribble protocol for the DNS server, which concisely describes the interaction patterns. Note that if a zone server responds with a zone PID, a new subsession is spawned using the returned PID to inhabit the `DNSZoneDataServer` role.

Implementation

We begin by writing a configuration file, associating each actor with the role it plays in each protocol. Only `ed_udp_handler_server` participates in both sessions.

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

The next step is to change each of the actors, which were formerly instances of `gen_server`, to be instances of `ssa_gen_server`. Making such a change can be done gradually: once the `ssactor_init` callback has been implemented, a `ssa_gen_server` works in exactly the same way as a `gen_server` in terms of handling call and cast messages.

Vitally, *no changes are made to the supervision structure*. The supervision structure itself is *orthogonal* to the monitoring of messages: developers do not have to change the supervision hierarchies of their applications in order to use `monitored-session-erlang`.

There are two points of note where the implementation of the original system diverges from that of the original server. The first is the spawning of the subsession to dynamically introduce the new role. In order to do so, we replace the previous implementation of the `get_zone` function (Listing 7.2) with that in Listing 7.3, which starts a new subsession. We also must add a case in the `ssactor_sconversation_established` callback for `ed_udp_handler_server`, which makes and returns the result of the synchronous call the the zone data server (Listing 7.4). Finally, the call handler in `ed_zone_data_server` must be changed to be a `ssaactor_handle_call`, but this change is trivial.

Listing 7.2: Original `get_zone` function

```
get_zone(Pid) ->
  gen_server:call(Pid, get_zone).
```

Listing 7.3: Modified `get_zone` function

```
get_zone(Pid, ConvKey) ->
  conversation:start_subsession(ConvKey, "GetZoneData", ["UDPHandlerServer"],
                                [{"DNSZoneDataServer", Pid}]).
```

Listing 7.4: DNS Subsession Initiation

```
ssactor_conversation_established("GetZoneData", _RN, _CID, ConvKey, State) ->
  RRTree = conversation:call(ConvKey, "DNSZoneDataServer", "ZoneDataRequest", [], []),
  conversation:subsession_complete(ConvKey, RRTree),
  {ok, State}.
```

The change is relatively small, but still a little larger than would be ideal given that only one interaction takes place in the subsession. The subsession abstraction is modular for introducing roles, and scales well when either multiple new roles are introduced or the subprotocol is more complex. For smaller calls, however, a future area for exploration may be finding a more lightweight abstraction for the special case of using a new participant only for a single call.

One area that could be improved in the implementation is that `FindNearestZone` was implemented in the original `gen_server` implementation as a synchronous call. Currently, synchronous calls are encoded with the assumption that they return a single value, whereas the result of the `FindNearestZone` could return two, depending on whether or not the zone was found. We can of course still encode the pattern in the system, but need to instead model the synchronous call as two asynchronous messages.

Failure Model

The failure model for the DNS server is simple. DNS is often implemented using UDP, which is itself an unreliable protocol: consequently, 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 let the supervisor restart the component, and let the request time out.

7.1.2 Chat Server

In this section, we detail the implementation of a chat server using `monitored-session-erlang`. We begin by outlining the system supervision hierarchy, shown in Figure 7.3.

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

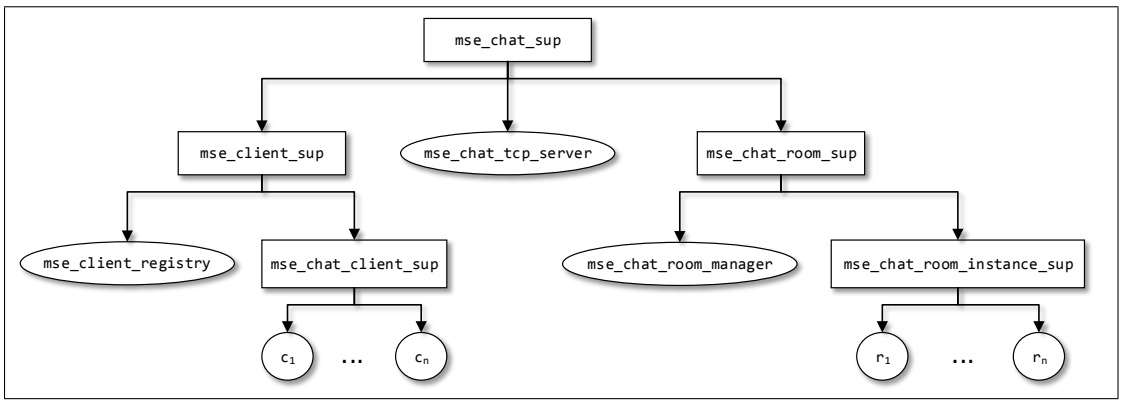


Figure 7.3: Supervision Hierarchy for Chat Server

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.

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.

We specify the protocol using two sessions: `ChatServer` (Listing 7.5), which describes the interactions prior to joining a chat room, and `ChatSession` (Listing 7.6), which describes the interactions within the session.

Listing 7.5: `ChatServer` protocol

```
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;
  }

}
```

Listing 7.6: `ChatSession` protocol

```
global protocol ChatSession(role ClientThread, role ChatRoom) {
  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;
```

```
    }
  }
}
```

Upon receiving the room PID, the client creates a new subsession for interactions within the chat room. In this section, 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. This is encoded using the `par` block, where each scope contains actions that can be performed in an interleaved fashion.

Implementation

The implementation requires some interesting communication patterns. Firstly, TCP messages are delivered to an actor and must trigger the sending of a message in the correct session. Secondly, the chat server makes essential use of the fact that multiple session instances can exist, in contrast to the work of Neykova and Yoshida [52]. Moreover, different sessions must *interact*: a message from one client to a chat room triggers a send to other clients in the chat room.

We begin, as ever, by creating a configuration file detailing the roles that each actor plays in the session:

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

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}.
```

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.

7.2 Performance

In this section, we empirically evaluate the performance of the framework.

7.2.1 Message Delivery Time

In order to measure the overheads of the `monitored-session-erlang` framework on message delivery time, we make use of the `PingPong` benchmark.

The `PingPong` benchmark is a simple, but effective, measure of the time taken to deliver messages. In this benchmark, an actor *A* sends a message, `ping`, to an actor `B`. Upon receiving a `ping` message, *B* sends a message `pong` back to *A*. This pattern is repeated for a set number of iterations.

The PingPong benchmark is a useful measure, as it allows the overheads of the monitoring framework to be compared against a baseline implementation, without the framework.

We run four different scenarios:

Vanilla Erlang

A simple `PingPong` implementation, not using the `monitored-session-erlang` framework. The scenario consists of two actors communicating sending messages using `gen_server2:cast`.

Session Erlang

An implementation of `PingPong` using the full `monitored-session-erlang` framework.

Session Erlang – No error reporting

An implementation of `PingPong` using the `monitored-session-erlang` framework, but without synchronous reporting of monitoring errors.

Session Erlang – No monitoring

An implementation of `PingPong` using the `monitored-session-erlang` framework, but without monitoring of messages.

The three different scenarios using `monitored-session-erlang` demonstrate different aspects of the system. By default, and in contrast to the original work on session actors, sending a message involves a synchronous call to the monitor, which subsequently performs a further synchronous call to monitor of the destination role. Should either of these checks fail, the message is not sent, and an exception is raised. Should the exception be caught, the program can attempt to send a different message.

Sending a message in this synchronous manner has the advantage of allowing a computation to be aborted as soon as a message is rejected by a monitor. The communication pattern for monitoring with error reporting is shown in Figure 7.4.

Alternatively, by employing a fully-asynchronous approach to monitoring, there are 2 fewer messages required, as results do not need to be returned from the destination monitor to the source monitor, and the source monitor to the source actor. A disadvantage, however, is that

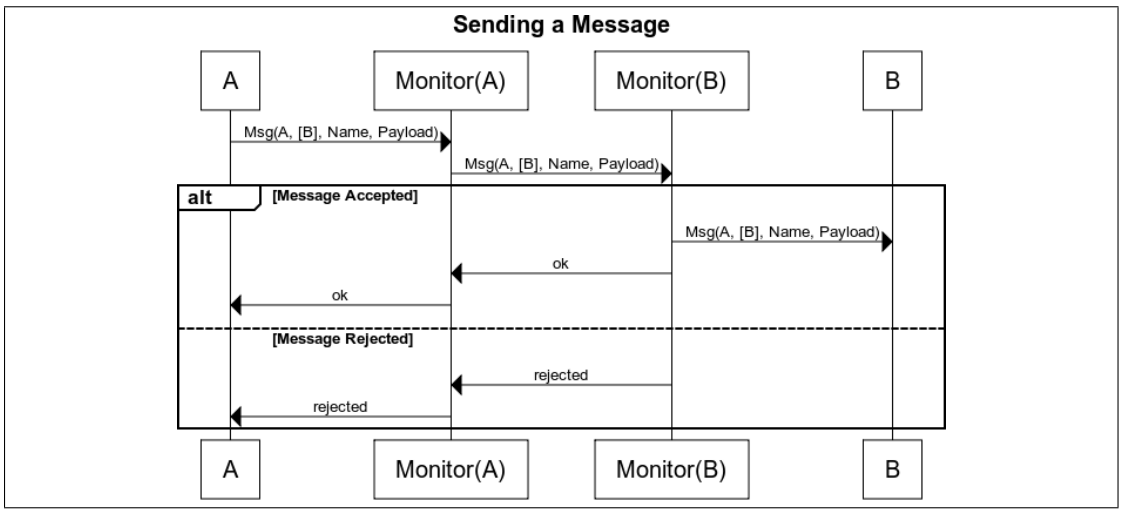


Figure 7.4: Communication pattern for sending a message with synchronous monitor error reporting

errors have to be reported as separate messages: consequently, the remainder of the handler must run before an error report can be processed, resulting in wasted computation. The communication pattern for monitoring without error reporting is shown in Figure 7.5.

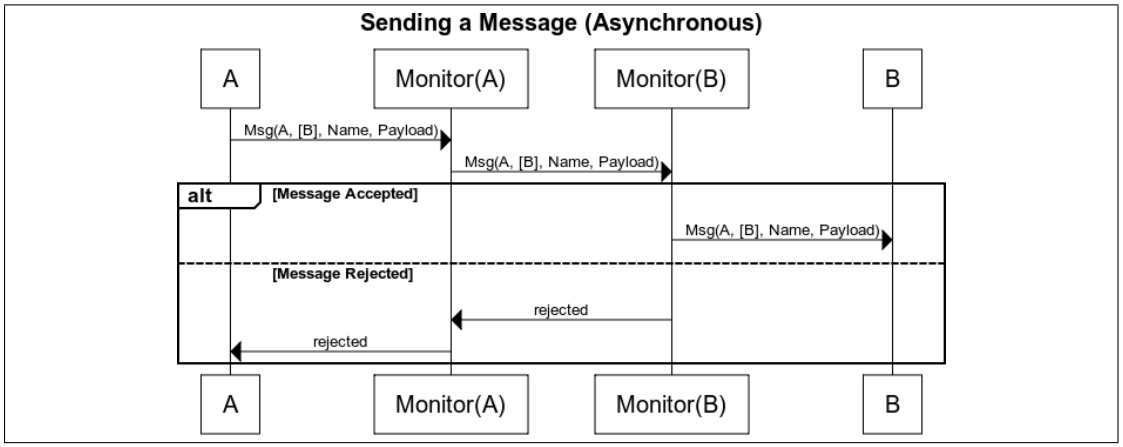


Figure 7.5: Communication pattern for sending a message without monitor error reporting

The final variation uses the `monitored-session-erlang` framework, but without monitoring: consequently, the overheads incurred in this scenario are as a result of having an external process per actor containing session state, and looking up the PID associated with destination roles. The final scenario implies no error reporting.

Experimental Setup

The PingPong benchmark was run on two cluster nodes, each with 4 16-core AMD Opteron 6376 processors, with each core running at 2300MHz, with a 2048Kb cache. Each node has

264Gb of memory. The round-trip-time latency between the nodes was measured as 0.101ms using `ping`.

Both nodes run Scientific Linux 7, using Erlang version 7.0.

The `PingPong` protocol used is shown in Listing 7.7.

Listing 7.7: Scribble protocol for PingPong benchmark

```
global protocol PingPong(role A, role B) {
  rec loop {
    ping() from A to B;
    pong() from B to A;
  }
}
```

The independent variable was the number of `ping` messages sent in the session, and the dependent variable was the time taken (in milliseconds) for the scenario to complete. Each case was repeated 100 times; the value plotted is the arithmetic mean time over the 100 repeats.

The time taken to complete a scenario is measured from when a session is successfully established (more precisely, upon the invocation of `ssactor_conversation_established` in actor A), and when the final `pong` message is received. Consequently, the time taken to set up the session is not included in these experimental results.

Experimental Results

Figure 7.6 shows the experimental results of the four basic experimental scenarios. Firstly, as would be expected, all four graphs are linear, meaning that the overhead of monitoring is constant throughout the session.

As would also be expected, monitoring incurs some overhead: the standard Erlang `gen_server2` implementation performs fastest, with a mean time per ping-pong iteration of 0.111ms. The use of the `monitored-session-erlang` framework without the monitor functionality yields an average time per ping-pong iteration of 0.160ms, an increase of 0.049ms per iteration due to the introduction of an additional monitoring process, and the resolution of role names to process IDs.

Introducing monitoring, but without synchronous error reporting, results in a mean time per ping-pong iteration of 0.188ms. Checking messages against a simple monitor with one outgoing transition therefore results in an overhead of only 0.028ms.

The full `monitored-session-erlang` system has a mean time per ping-pong iteration of 0.23ms, giving a final overhead per iteration of 0.12ms (or 0.06ms per message).

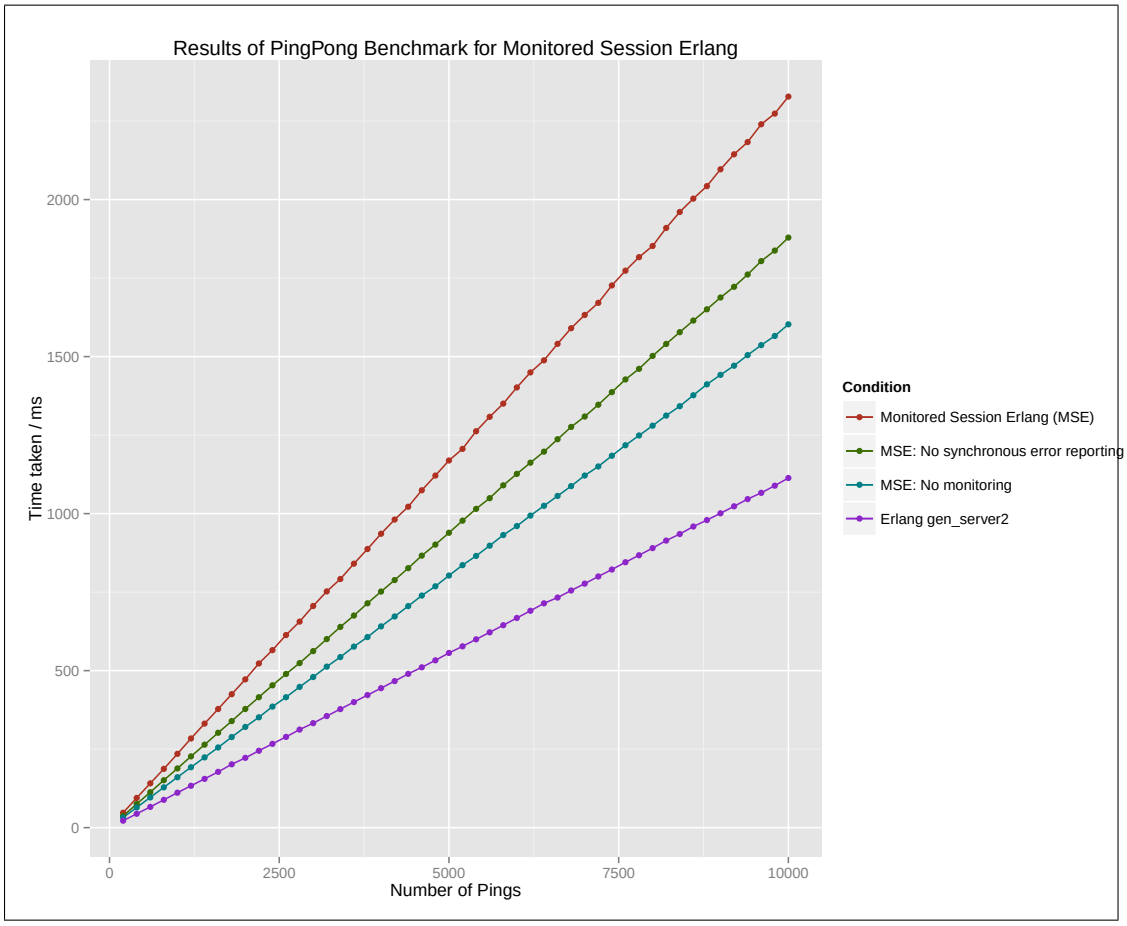


Figure 7.6: Experimental results of PingPong benchmark for four experimental scenarios of `monitored-session-erlang`

The overheads can be explained by the additional messages that need to be sent between the monitors in order to detect and report monitoring errors: while the original `gen_server2` implementation requires only two messages to be sent, the introduction of the `actor_monitor` process requires that an additional two messages are sent: one when sending, and one when receiving. Adding monitoring only adds a small overhead, but synchronously reporting monitoring errors requires two additional messages to be sent: one to return the result from the local monitor, and one to return the result from the remote monitor. Importantly, however, the messages sent to deliver the message to the remote actor and the message to notify the local actor of the successful send can be sent in parallel.

7.2.2 Monitor Size

Table 7.1 shows statistics about the number of states and transitions in monitors, as well as the time in milliseconds taken to construct the monitor, and the amount of memory to store the monitor representation. The construction time displayed in the table is the arithmetic mean of 100,000 runs on the same machine as in Section 7.2.1.

In all cases, monitor construction time is negligible, at under a millisecond. Memory usage ranges from 1.6KB to 17KB in contrast to the implementation of Hu et al. [40] in which most monitors remain under 1.5KB: while still reasonably small, our implementation is not yet optimised for space efficiency.

Scenario	Protocol Name	Role	Number of States	Number of Transitions	Construction Time /ms	Memory Usage / b
Travel Agent	BookTravel	Customer	5	7	0.156	6224
Travel Agent	BookTravel	FlightBookingService	5	3	0.071	3088
Travel Agent	BookTravel	HotelBookingService	5	3	0.070	2800
Travel Agent	BookTravel	TravelAgent	20	20	0.443	17792
Travel Agent	CancelBookings	FlightBookingService	4	2	0.050	1608
Travel Agent	CancelBookings	HotelBookingService	4	2	0.050	1592
Travel Agent	CancelBookings	TravelAgent	6	3	0.074	2888
Travel Agent	PerformBooking	Customer	3	2	0.050	2192
Travel Agent	PerformBooking	FlightBookingService	5	3	0.071	2944
Travel Agent	PerformBooking	HotelBookingService	5	3	0.072	3040
Travel Agent	PerformBooking	TravelAgent	10	7	0.160	7752
Travel Agent	PerformPayment	PaymentProcessor	3	2	0.050	2384
Travel Agent	PerformPayment	TravelAgent	3	2	0.050	2544
Chat Server	ChatServer	ClientThread	8	10	0.211	7328
Chat Server	ChatServer	RoomRegistry	6	8	0.169	6104
Chat Server	ChatSession	ChatRoom	6	4	0.096	3272
Chat Server	ChatSession	ClientThread	6	4	0.095	3096
DNS Server	GetZoneData	DNSZoneDataServer	3	2	0.063	2112
DNS Server	GetZoneData	UDPHandlerServer	3	2	0.064	2144
DNS Server	HandleDNSRequest	DNSZoneRegServer	3	3	0.073	2784
DNS Server	HandleDNSRequest	UDPHandlerServer	5	5	0.111	4232
Two Buyer	TwoBuyers	A	5	6	0.130	2976
Two Buyer	TwoBuyers	B	5	6	0.139	3072
Two Buyer	TwoBuyers	S	5	6	0.132	2976

Table 7.1: Monitor Benchmarks

Chapter 8

Conclusion

Erlang has gained a positive reputation as a language for developing highly-robust and reliable distributed software. Erlang encourages developers to design applications in such a way that faults are isolated, and processes may be restarted by *supervisors* upon failure in order to achieve a high level of availability and reliability.

In order to facilitate fault isolation, Erlang is based upon the actor model. Consequently, processes *do not co-ordinate with each other using shared state*, instead relying on message passing. In spite of the heavy use of message passing, until now there has been little work on ensuring that Erlang processes conform to communication patterns, or that the communication patterns are race- or deadlock-free.

8.1 Overview of Contributions

This thesis has investigated the use of *multiparty session types* to allow protocols to be externally specified and checked for safety, and *runtime monitoring* techniques to ensure that communications in Erlang applications conform to their session types.

In doing so, we have adapted the existing work on multiparty session actors by Neykova and Yoshida [52] to the setting of distributed Erlang/OTP applications, in the form of a framework, `monitored-session-erlang`. Designing and implementing `monitored-session-erlang` has involved investigating how the conceptual framework of multiparty session actors should be adapted to the setting of an actor-based functional language with no shared or mutable state, and has involved departures from the original work such as synchronous error reporting and allowing actors to partake in multiple instances of a protocol. We have argued that monitoring of session types should be treated as largely, but not completely orthogonal to the structure of Erlang/OTP applications using supervision trees: developers should *not* have to change the supervision

structure of their applications in order to use the framework. Monitor failures throw exceptions which may in turn cause a process to fail and be restarted by a supervisor.

Secondly, we have investigated common Erlang patterns which cannot be expressed using multiparty session types alone. Erlang processes often send process IDs, in essence dynamically introducing participants partway through a session, and have shown how subsessions [28] can be used to encapsulate this pattern. Additionally, we have investigated the use of synchronous, blocking calls in Erlang applications, and developed extensions to the Scribble language and `monitored-session-erlang` runtime to both ensure that communication with an actor awaiting the response of a synchronous call is not involved while the call is in progress, and that intermediate state does not have to be saved and 'threaded' through multiple asynchronous message handlers.

Thirdly, we have investigated the issue of failure detection and failure handling. Previous work on multiparty session types concentrates on the case where the processes playing roles in sessions are available throughout the entirety of a session: an assumption which cannot be made in the setting of Erlang/OTP applications due to the 'let it fail' design philosophy. In order to address the issue of failures within sessions, we have considered two methods of failure-detection: push-based, using Erlang's `monitor` functions to detect when a process has terminated and role reachability analysis to check whether it is safe to proceed; and pull-based, using synchronous calls and a two-phase commit protocol to ensure atomic multicasts.

In order to handle failures, we again used subsessions, introducing extensions which allow the remainder of a protocol to be predicated on the success or failure of a subsession. We detailed how, by splitting possibly-failing portions of a session into subsessions, we may repeat part of a protocol with a new instance of a participant, should a participant fail. We have also shown that the same technique may be used to handle application-level exceptions.

Finally, we evaluate the framework through a case study of adding session types to a publicly-available DNS server, and the implementation of a chat server. We have additionally performed an empirical evaluation using the PingPong benchmark, in order to investigate the overheads incurred by different features of the framework, as well as investigated the generation and storage overheads of monitors.

8.2 Critical Evaluation

In developing `monitored-session-erlang`, we have investigated how multiparty session types can be used to encode and monitor the patterns in Erlang/OTP applications. A primary concern has been to ensure that application developers can still use standard OTP design patterns such

as supervision hierarchies, and by monitoring incoming and outgoing messages, developers can be alerted when a message does not conform to the session type, with the supervision tree correcting the process.

One limitation with the implementation of synchronous calls is that they currently only allow a single value to be returned: consequently, it is not possible to encode that the result of a synchronous call determines the remainder of the session. This is limiting, and is the obvious extension for future work.

Secondly, the addition of an external monitoring process, along with synchronous error reporting, does incur some overheads on the time taken to deliver a message. The addition of the external monitoring process contributes to this, due to the additional messages required. It is worthwhile to note, however, that the overheads are incurred *only* when sending and receiving messages: no other traces are dynamically generated and monitored, unlike with other approaches to runtime verification.

Finally, finding appropriate case studies was a difficult task: open-source Erlang applications seemed to be either too small or simple to be of value, or too large to analyse and encode in the time available, for example `yaws` or `ejabberd`. In order to do larger case studies, perhaps a better approach would be to record traces of Erlang communications, and use session type synthesis techniques [43] to generate the session types.

8.3 Future Work

An interesting direction to take would be a full formal semantics for session actors. A promising starting point would be to adapt a process calculus based on the actor model, for example the Aπ calculus introduced by Agha and Thati [2], and integrate the calculus with existing formal frameworks for dynamic monitoring of session types such as the work by Bocchi et al. [12].

This thesis has aimed to investigate high-level Erlang/OTP applications, and consequently builds on top of the Erlang `gen_server` behaviour. We do not investigate the lower level communication primitives, such as sending and selective receives. Such a line of work would likely be a departure from the session actor framework, but it would be interesting to investigate how communication at a lower level could be monitored.

Another, more ambitious line of work would investigate the extent to which session types in Erlang applications could be checked statically. Work on success typing [44] has allowed data types to be specified and checked in Erlang applications, and it would be interesting to see whether such an approach could be extended to check session types.

Finally, it would be interesting to see how session types could be added to an actor-based language as a first-class construct: this would involve identifying the required abstractions, and developing an appropriate type system for actor mailboxes.

Appendix A

`ssa_gen_server` Callback Functions

`ssactor_init`

Replaces `gen_server:init`, and allows the session actor to be initialised with some user state.

`ssactor_join`

Called when an actor is invited to fulfil a role in a protocol. The actor may choose to accept or decline the invitation, and update the user state.

`ssactor_handle_message`

Handles an asynchronous session message, which has already been accepted by the monitor.

`ssactor_handle_call`

Handles a *synchronous* session message, which has already been accepted by the monitor. This is discussed further in Section 5.2

`ssactor_become`

Actors can partake in multiple protocols, and switch between them. The `ssactor_become` callback is called when an actor wishes to switch to a different protocol.

`ssactor_conversation_established`

The invitation algorithm for inviting actors to fulfil roles within a session is asynchronous. The `ssactor_conversation_established` function is called when a session is successfully established: that is, when all roles have been successfully populated by endpoints, and session communication can begin.

`ssactor_conversation_error`

Conversely, should an error occur when establishing a session (for example, if no actors accept the invitation to join), then the `ssactor_conversation_error` function will be called.

This allows any necessary cleanup to take place.

ssactor_conversation_ended

The ssactor_conversation_ended function is called when a session is terminated, either for normal reasons or in the case of an error.

Appendix B

Sample Output

In this example, we show the output from a sample run of the two buyer protocol. All three actors are arranged in a supervision tree, and should an actor terminate, it will be restarted by the supervisor.

In this circumstance, we modify the message sent by S from 'date' to 'datum'. In doing so, the monitor fails, and an exception is thrown in the calling process before any other messages are sent in the system. The system begins a safety check, finds that S is still required in the protocol, and ends the session. The session is then started again, with the same result—note that the PIDs for actors A and B remain constant, but B is different as it has been restarted by the supervisor.

```
Erlang/OTP 18 [erts-7.0] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe] [kernel-
    poll:false]

Eshell V7.0  (abort with ^G)
1> sup_two_buyer_main:main().

=INFO REPORT==== 21-Aug-2015::06:41:12 ===
Initialising conversation system.

=WARNING REPORT==== 21-Aug-2015::06:41:12 ===
WARN: Could not parse file SynchronousSequencedStateCells.scr: ignoring

=INFO REPORT==== 21-Aug-2015::06:41:12 ===
actor registry registered at:undefined
Initialised successfully
Starting seller
started supervised seller successfully
started buyer2 successfully
started buyer1 successfully
Starting conversation in buyer1.
Press to start a new session
```

```
=INFO REPORT==== 21-Aug-2015::06:41:12 ===
Starting conversation for protocol TwoBuyers.

=INFO REPORT==== 21-Aug-2015::06:41:12 ===
Started instance process

=INFO REPORT==== 21-Aug-2015::06:41:12 ===
Registered for role A in protocol TwoBuyers.
Actor sup_buyer1, actor PID <0.46.0>, monitor instance <0.45.0>.
=INFO REPORT==== 21-Aug-2015::06:41:12 ===
Registered for role B in protocol TwoBuyers.
Actor sup_buyer2, actor PID <0.44.0>, monitor instance <0.43.0>.
=INFO REPORT==== 21-Aug-2015::06:41:12 ===
Registered for role S in protocol TwoBuyers.
Actor sup_seller, actor PID <0.42.0>, monitor instance <0.41.0>.
=INFO REPORT==== 21-Aug-2015::06:41:12 ===
Res: ok

=INFO REPORT==== 21-Aug-2015::06:41:12 ===
Actor seller: Received title To Kill a Mockingbird from A

=INFO REPORT==== 21-Aug-2015::06:41:12 ===
Actor buyer2: Received quote of 40 from S

=INFO REPORT==== 21-Aug-2015::06:41:12 ===
Actor buyer1: Received quote of 40 from S

=INFO REPORT==== 21-Aug-2015::06:41:12 ===
Actor buyer2: Received share quote (20) from A

=INFO REPORT==== 21-Aug-2015::06:41:12 ===
Actor buyer2: Accepted share quote (threshold 50)

=INFO REPORT==== 21-Aug-2015::06:41:12 ===
Actor seller: B accepted quote; received address Informatics Forum
Starting seller

=INFO REPORT==== 21-Aug-2015::06:41:12 ===
Actor buyer1: B accepted quote; received address ("Informatics Forum")

=WARNING REPORT==== 21-Aug-2015::06:41:12 ===
Monitor failed when processing message {message_record,#Ref<0.0.2.164>,"S",
                                        ["B"],
                                        "datum",
                                        ["String"],
                                        ["Sometime in the future"]} (send). Error:
                                            bad_message

Actor sup_seller, actor PID <0.42.0>, monitor instance <0.41.0>.
=ERROR REPORT==== 21-Aug-2015::06:41:12 ===
Actor terminating for reason {{error,bad_message},
                              [{conversation,send,5,
                                [{file,"src/conversation/conversation.erl"},
                                 {line,25}]},
                               {sup_seller,ssactor_handle_message,8,
```

```
                                [{file,
                                  "src/SupervisedTwoBuyer/sup_seller.erl"},
                                 {line,35}]},
                                {ssa_gen_server,handle_cast,2,
                                 [{file,"src/behaviours/ssa_gen_server.erl"},
                                  {line,195}]},
                                {gen_server2,handle_msg,2,
                                 [{file,"src/util/gen_server2.erl"},
                                  {line,1034}]},
                                {proc_lib,init_p_do_apply,3,
                                 [{file,"proc_lib.erl"},{line,239}]}]}

SSACTOR: Actor sup_seller, actor PID <0.42.0>, monitor PID <0.41.0>.
=ERROR REPORT==== 21-Aug-2015::06:41:12 ===
** Generic server <0.42.0> terminating
** Last message in was {'$gen_cast',
                           {ssa_msg,"TwoBuyers","S",<0.47.0>,
                               {message_record,#Ref<0.0.2.158>,"B",
                                   ["A","S"],
                                   "accept",
                                   ["String"],
                                   ["Informatics Forum"]}}}
** When Server state == {actor_state,sup_seller,<0.41.0>,no_state}
** Reason for termination ==
** {{error,bad_message},
    [{conversation,send,5,
                  [{file,"src/conversation/conversation.erl"},{line,25}]},
     {sup_seller,ssactor_handle_message,8,
                 [{file,"src/SupervisedTwoBuyer/sup_seller.erl"},{line,35}]},
     {ssa_gen_server,handle_cast,2,
                    [{file,"src/behaviours/ssa_gen_server.erl"},{line,195}]},
     {gen_server2,handle_msg,2,
                  [{file,"src/util/gen_server2.erl"},{line,1034}]},
     {proc_lib,init_p_do_apply,3,[{file,"proc_lib.erl"},{line,239}]}]}

=INFO REPORT==== 21-Aug-2015::06:41:12 ===
Role "S" down. Beginning safety check.

=INFO REPORT==== 21-Aug-2015::06:41:12 ===
Safety check failed: "S" still needed in "B"

=INFO REPORT==== 21-Aug-2015::06:41:12 ===
Actor buyer2: Conversation <0.47.0> ended.


=INFO REPORT==== 21-Aug-2015::06:41:12 ===
Actor buyer1: Conversation <0.47.0> ended.


Press to start a new session
Starting conversation in buyer1.

=INFO REPORT==== 21-Aug-2015::06:41:19 ===
Starting conversation for protocol TwoBuyers.
```

```
=INFO REPORT==== 21-Aug-2015::06:41:19 ===
Started instance process

=INFO REPORT==== 21-Aug-2015::06:41:19 ===
Registered for role A in protocol TwoBuyers.
Actor sup_buyer1, actor PID <0.46.0>, monitor instance <0.45.0>.
=INFO REPORT==== 21-Aug-2015::06:41:19 ===
Registered for role B in protocol TwoBuyers.
Actor sup_buyer2, actor PID <0.44.0>, monitor instance <0.43.0>.
=INFO REPORT==== 21-Aug-2015::06:41:19 ===
Registered for role S in protocol TwoBuyers.
Actor sup_seller, actor PID <0.49.0>, monitor instance <0.48.0>.
=INFO REPORT==== 21-Aug-2015::06:41:19 ===
Res: ok

=INFO REPORT==== 21-Aug-2015::06:41:19 ===
Actor seller: Received title To Kill a Mockingbird from A

=INFO REPORT==== 21-Aug-2015::06:41:19 ===
Actor buyer2: Received quote of 40 from S

=INFO REPORT==== 21-Aug-2015::06:41:19 ===
Actor buyer1: Received quote of 40 from S
Starting seller

=INFO REPORT==== 21-Aug-2015::06:41:19 ===
Actor buyer2: Received share quote (20) from A

=INFO REPORT==== 21-Aug-2015::06:41:19 ===
Actor buyer2: Accepted share quote (threshold 50)

=INFO REPORT==== 21-Aug-2015::06:41:19 ===
Actor seller: B accepted quote; received address Informatics Forum

=INFO REPORT==== 21-Aug-2015::06:41:19 ===
Actor buyer1: B accepted quote; received address ("Informatics Forum")

=WARNING REPORT==== 21-Aug-2015::06:41:19 ===
Monitor failed when processing message {message_record,#Ref<0.0.2.194>,"S",
                                       ["B"],
                                       "datum",
                                       ["String"],
                                       ["Sometime in the future"]} (send). Error:
                                           bad_message

Actor sup_seller, actor PID <0.49.0>, monitor instance <0.48.0>.
=ERROR REPORT==== 21-Aug-2015::06:41:19 ===
Actor terminating for reason {{error,bad_message},
                              [{conversation,send,5,
                                [{file,"src/conversation/conversation.erl"},
                                 {line,25}]},
                               {sup_seller,ssactor_handle_message,8,
                                [{file,
                                  "src/SupervisedTwoBuyer/sup_seller.erl"},
                                 {line,35}]},
```

```
                                {ssa_gen_server,handle_cast,2,
                                 [{file,"src/behaviours/ssa_gen_server.erl"},
                                  {line,195}]},
                                {gen_server2,handle_msg,2,
                                 [{file,"src/util/gen_server2.erl"},
                                  {line,1034}]},
                                {proc_lib,init_p_do_apply,3,
                                 [{file,"proc_lib.erl"},{line,239}]}]}

SSACTOR: Actor sup_seller, actor PID <0.49.0>, monitor PID <0.48.0>.
=ERROR REPORT==== 21-Aug-2015::06:41:19 ===
** Generic server <0.49.0> terminating
** Last message in was {'$gen_cast',
                           {ssa_msg,"TwoBuyers","S",<0.50.0>,
                               {message_record,#Ref<0.0.2.190>,"B",
                                   ["A","S"],
                                   "accept",
                                   ["String"],
                                   ["Informatics Forum"]}}}
** When Server state == {actor_state,sup_seller,<0.48.0>,no_state}
** Reason for termination ==
** {{error,bad_message},
    [{conversation,send,5,
                   [{file,"src/conversation/conversation.erl"},{line,25}]},
     {sup_seller,ssactor_handle_message,8,
                 [{file,"src/SupervisedTwoBuyer/sup_seller.erl"},{line,35}]},
     {ssa_gen_server,handle_cast,2,
                     [{file,"src/behaviours/ssa_gen_server.erl"},{line,195}]},
     {gen_server2,handle_msg,2,
                  [{file,"src/util/gen_server2.erl"},{line,1034}]},
     {proc_lib,init_p_do_apply,3,[{file,"proc_lib.erl"},{line,239}]}]}

=INFO REPORT==== 21-Aug-2015::06:41:19 ===
Role "S" down. Beginning safety check.

=INFO REPORT==== 21-Aug-2015::06:41:19 ===
Safety check failed: "S" still needed in "B"

=INFO REPORT==== 21-Aug-2015::06:41:19 ===
Actor buyer2: Conversation <0.50.0> ended.

=INFO REPORT==== 21-Aug-2015::06:41:19 ===
Actor buyer1: Conversation <0.50.0> ended.
```

Bibliography

[1] Samson Abramsky. Proofs as Processes. *Theoretical Computer Science*, 135(1):5–9, 1994.

[2] Gul Agha and Prasanna Thati. An algebraic theory of actors and its application to a simple Object-Based language. In Olaf Owe, Stein Krogdahl, and Tom Lyche, editors, *From Object-Orientation to Formal Methods*, volume 2635 of *Lecture Notes in Computer Science*, chapter 4, pages 26–57. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-21366-6. doi: 10.1007/978-3-540-39993-3_4. URL http://dx.doi.org/10.1007/978-3-540-39993-3_4.

[3] Gul A. Agha. Actors: a Model of Concurrent Computation in Distributed Systems. 1985.

[4] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, The Royal Institute of Technology Stockholm, Sweden, 2003.

[5] Joe Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, 2010.

[6] Robert Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19:335–376, July 2009. ISSN 1469-7653. doi: 10.1017/s095679680900728x. URL http://dx.doi.org/10.1017/s095679680900728x.

[7] Gianluigi Bellin and Philip J. Scott. On the Pi-Calculus and Linear Logic. *Theoretical Computer Science*, 135(1):11–65, 1994.

[8] Nick Benton and Andrew Kennedy. Exceptional syntax. *Journal of Functional Programming*, 11:395–410, 2001. ISSN 1469-7653. doi: 10.1017/s0956796801004099. URL http://dx.doi.org/10.1017/s0956796801004099.

[9] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR 2008-Concurrency Theory*, pages 418–433. Springer, 2008.

[10] L. Bocchi, J. Lange, and N. Yoshida. Meeting Deadlines Together. In *CONCUR 2015—Concurrency Theory*, 2015.

[11] Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR 2010-Concurrency Theory*, pages 162–176. Springer, 2010.

[12] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. In Dirk Beyer and Michele Boreale, editors, *Formal Techniques for Distributed Systems*, volume 7892 of *Lecture Notes in Computer Science*, pages 50–65. Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-38592-6_5. URL http://dx.doi.org/10.1007/978-3-642-38592-6_5.

[13] Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed multiparty session types. In *CONCUR 2014–Concurrency Theory*, pages 419–434. Springer, 2014.

[14] Omg Bpmn. BPMN 2.0 by Example. *OMG Document Number: dtc/2010-06-02*, 2010.

[15] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM (JACM)*, 30(2):323–342, 1983.

[16] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer Berlin Heidelberg, 2010. doi: 10.1007/978-3-642-15375-4_16. URL http://dx.doi.org/10.1007/978-3-642-15375-4_16.

[17] Sara Capecchi, Elena Giachino, Nobuko Yoshida, and Others. Global escape in multiparty sessions. *Mathematical Structures in Computer Science*, 2013.

[18] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured interactional exceptions in session types. In Franck Breugel and Marsha Chechik, editors, *CONCUR 2008 - Concurrency Theory*, volume 5201 of *Lecture Notes in Computer Science*, pages 402–417. Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-85361-9_32. URL http://dx.doi.org/10.1007/978-3-540-85361-9_32.

[19] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Communication-Centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2), June 2012. doi: 10.1145/2220365.2220367. URL http://doi.acm.org/10.1145/2220365.2220367.

[20] David Castro, Victor M. Gulias, Clara Benac Earle, Lars-AAke Fredlund, and Samuel Rivas. A Case Study on Verifying a Supervisor Component using McErlang. *Electronic Notes in Theoretical Computer Science*, 271:23–40, 2011.

[21] Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko

Yoshida. Asynchronous Distributed Monitoring for Multiparty Session Enforcement. In Roberto Bruni and Vladimiro Sassone, editors, *Trustworthy Global Computing*, volume 7173 of *Lecture Notes in Computer Science*, pages 25–45. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-30065-3_2. URL http://dx.doi.org/10.1007/978-3-642-30065-3_2.

[22] Koen Claessen and Hans Svensson. A semantics for distributed erlang. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang*, ERLANG '05, pages 78–87, New York, NY, USA, 2005. ACM. ISBN 1-59593-066-3. doi: 10.1145/1088361.1088376. URL http://dx.doi.org/10.1145/1088361.1088376.

[23] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA — safer monitoring of Real-Time java programs (tool paper). In *Software Engineering and Formal Methods, 2009 Seventh IEEE International Conference on*, pages 33–37. IEEE, November 2009. ISBN 978-0-7695-3870-9. doi: 10.1109/sefm.2009.13. URL http://dx.doi.org/10.1109/sefm.2009.13.

[24] Christian Colombo, Adrian Francalanza, and Rudolph Gatt. Elarva: A monitoring tool for erlang. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification*, volume 7186 of *Lecture Notes in Computer Science*, pages 370–374. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-29860-8_29. URL http://dx.doi.org/10.1007/978-3-642-29860-8_29.

[25] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, pages 1–65, 2015.

[26] Silvia Crafa. Behavioural types for actor systems. *CoRR*, abs/1206.1687, 2012. URL http://arxiv.org/abs/1206.1687.

[27] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, pages 139–150. ACM, 2012.

[28] Romain Demangeon and Kohei Honda. Nested Protocols in Session Types. In *CONCUR 2012—Concurrency Theory*, pages 272–286. Springer, 2012.

[29] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In Helmut Seidl, editor, *Programming Languages and Systems*, volume 7211 of *Lecture Notes in Computer Science*, pages 194–213. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-28869-2_10. URL http://dx.doi.org/10.1007/978-3-642-28869-2_10.

[30] Lars-AAke Fredlund and Hans Svensson. McErlang: A Model Checker for a Distributed Functional Programming Language. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 125–136, New York, NY, USA, 2007. ACM. doi: 10.1145/1291151.1291171. URL http://doi.acm.org/10.1145/1291151.1291171.

[31] Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2):191–225, November 2005. ISSN 0001-5903. doi: 10.1007/s00236-005-0177-z. URL http://dx.doi.org/10.1007/s00236-005-0177-z.

[32] Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20:19–50, January 2010. ISSN 1469-7653. doi: 10.1017/s0956796809990268. URL http://dx.doi.org/10.1017/s0956796809990268.

[33] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, January 1987. ISSN 03043975. doi: 10.1016/0304-3975(87)90045-4. URL http://dx.doi.org/10.1016/0304-3975(87)90045-4.

[34] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. URL http://portal.acm.org/citation.cfm?id=1624804.

[35] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978. ISSN 0001-0782. doi: 10.1145/359576.359585. URL http://dx.doi.org/10.1145/359576.359585.

[36] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer Berlin Heidelberg, 1993. doi: 10.1007/3-540-57208-2_35. URL http://dx.doi.org/10.1007/3-540-57208-2_35.

[37] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems*, volume 1381 of *Lecture Notes in Computer Science*, chapter 9, pages 122–138. Springer Berlin Heidelberg, Berlin/Heidelberg, 1998. ISBN 3-540-64302-8. doi: 10.1007/bfb0053567. URL http://dx.doi.org/10.1007/bfb0053567.

[38] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 273–284, New York, NY, USA,

2008. ACM. doi: 10.1145/1328438.1328472. URL http://doi.acm.org/10.1145/1328438.1328472.

[39] 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.

[40] 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.

[41] Naoki Kobayashi. Type systems for concurrent programs. In *Formal Methods at the Crossroads. From Panacea to Foundational Support*, pages 439–453. Springer, 2003.

[42] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the Pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, September 1999. ISSN 0164-0925. doi: 10.1145/330249.330251. URL http://dx.doi.org/10.1145/330249.330251.

[43] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 221–232, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2676964. URL http://dx.doi.org/10.1145/2676726.2676964.

[44] Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, pages 167–178, New York, NY, USA, 2006. ACM. ISBN 1-59593-388-3. doi: 10.1145/1140335.1140356. URL http://dx.doi.org/10.1145/1140335.1140356.

[45] Sam Lindley and J. Garrett Morris. Sessions as Propositions. In Alastair F. Donaldson and Vasco T. Vasconcelos, editors, *Proceedings of the 7th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, volume 155 of *Electronic Proceedings in Theoretical Computer Science*, pages 9–16. Open Publishing Association, 2014. doi: 10.4204/EPTCS.155.2. URL http://dx.doi.org/10.4204/EPTCS.155.2.

[46] Sam Lindley and J. Garrett Morris. A Semantics for Propositions as Sessions. In *Programming Languages and Systems*, pages 560–584. Springer, 2015.

[47] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. ISBN 0387102353. URL http://portal.acm.org/citation.cfm?id=539036.

[48] Robin Milner. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1st edition, June 1999. ISBN 0521658691. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0521658691.

[49] Dimitris Mostrous and Vasco T. Vasconcelos. Session typing for a featherweight erlang. In Wolfgang De Meuter and Gruia-Catalin Roman, editors, *Coordination Models and Languages*, volume 6721 of *Lecture Notes in Computer Science*, pages 95–109. Springer Berlin Heidelberg, 2011. doi: 10.1007/978-3-642-21464-6_7. URL http://dx.doi.org/10.1007/978-3-642-21464-6_7.

[50] Dimitris Mostrous and Vasco T. Vasconcelos. Affine sessions. In Eva Kühn and Rosario Pugliese, editors, *Coordination Models and Languages*, Lecture Notes in Computer Science, pages 115–130. Springer Berlin Heidelberg, 2014. doi: 10.1007/978-3-662-43376-8_8. URL http://dx.doi.org/10.1007/978-3-662-43376-8_8.

[51] Rumyana Neykova and Nobuko Yoshida. Multiparty Session Actors. Extended journal version. Under submission.

[52] Rumyana Neykova and Nobuko Yoshida. Multiparty Session Actors. In Eva Kühn and Rosario Pugliese, editors, *Coordination Models and Languages*, Lecture Notes in Computer Science, pages 131–146. Springer Berlin Heidelberg, 2014. doi: 10.1007/978-3-662-43376-8_9. URL http://dx.doi.org/10.1007/978-3-662-43376-8_9.

[53] Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. SPY: Local Verification of Global Protocols. In *Runtime Verification*, pages 358–363. Springer, 2013.

[54] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *arXiv preprint arXiv:1408.5979*, 2014.

[55] Nicholas Ng and Nobuko Yoshida. Pabble: Parameterised scribble for parallel programming. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 707–714. IEEE, February 2014. doi: 10.1109/pdp.2014.20. URL http://dx.doi.org/10.1109/pdp.2014.20.

[56] Jan H. Nyström. *Analysing Fault Tolerance for Erlang Applications*. PhD thesis, Uppsala University, 2009.

[57] Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. *SIGPLAN Not.*, 44(2):25–36, September 2008. ISSN 0362-1340. doi: 10.1145/1543134.1411290. URL http://dx.doi.org/10.1145/1543134.1411290.

[58] Matthew Sackman and Susan Eisenbach. Session types in haskell. 2008.

[59] Hans Svensson and Lars-Ake Fredlund. A more accurate semantics for distributed Erlang. In *Erlang Workshop*, pages 43–54. Citeseer, 2007.

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

[61] Vasco T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, August 2012. ISSN 08905401. doi: 10.1016/j.ic.2012.05.002. URL http://dx.doi.org/10.1016/j.ic.2012.05.002.

[62] Philip Wadler. Propositions as Sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 273–286, New York, NY, USA, 2012. ACM. doi: 10.1145/2364527.2364568. URL http://doi.acm.org/10.1145/2364527.2364568.