Let It Recover: Multiparty Protocol-Induced Recovery

Rumyana Neykova  Nobuko Yoshida
Imperial College London

Abstract
Fault-tolerant communication systems rely on recovery strategies which are often error-prone (e.g. a programmer manually specifies recovery strategies) or inefficient (e.g. the whole system is restarted from the beginning). This paper proposes a static analysis based on multiparty session types that can efficiently compute a safe global state from which a system of interacting processes should be recovered. We statically analyse the communication flow of a program, given as a multiparty protocol, to extract the causal dependencies between processes and to localise failures. We formalise our recovery algorithm and prove its safety. A recovered communication system is free from deadlocks, orphan messages and recovery errors. Our recovery algorithm incurs less communication cost (only affected processes are notified) and overall execution time (only required states are repeated). On top of our analysis, we design and implement a runtime framework in Erlang where failed processes and their dependencies are soundly restarted from a computed safe state. We evaluate our recovery framework on message-passing benchmarks and a use case for crawling webpages. The experimental results indicate our framework outperforms a built-in static recovery strategy in Erlang when a part of the protocol can be safely recovered.

1. Introduction
1.1 Motivation
Let it Crash recovery model. Despite the importance of fast and correct recovery in distributed systems, it is still difficult and error-prone to implement fault-tolerant components. A well-established fault-tolerance model is the Let it crash model, adopted by the programming language Erlang. In Erlang, rather than trying to handle and recover from all possible exceptional and failure states, one can instead let processes crash and let the runtime automatically recycle them back to their initial state. Failures and errors propagation are managed by organising the running processes in a hierarchical structure, called a supervision tree, where each process and its dependencies are monitored by a parent process, called supervisor.

While the Let it crash model has gained popularity and has recently been adopted in other commercial languages and frameworks (Go, Akka and Scala), it still faces two major technical problems. First, the correctness of recovery strategies relies on the assumption that a global process structure (i.e. a supervision structure) of the system is correctly written by a programmer. A recent study reveals a misconfiguration of the supervision tree is a common source of errors for recovery [27]. Second, supervision strategies have a fixed structure, hence they often recover too little or too many processes, and are unable to capture the dynamic nature of the communication dependencies between processes. This often leads to a redundant and/or a unsound recovery mechanism (as shown in Table 1 and explained in §1.2). Here a question is: Can we soundly generate recovery strategies, minimising the recovery overhead? This paper answers the question affirmatively using a theory of multiparty session types [17].

Let It Recover: Multiparty Protocol-Induced Recovery

Rumyana Neykova  Nobuko Yoshida
Imperial College London

Abstract
Fault-tolerant communication systems rely on recovery strategies which are often error-prone (e.g. a programmer manually specifies recovery strategies) or inefficient (e.g. the whole system is restarted from the beginning). This paper proposes a static analysis based on multiparty session types that can efficiently compute a safe global state from which a system of interacting processes should be recovered. We statically analyse the communication flow of a program, given as a multiparty protocol, to extract the causal dependencies between processes and to localise failures. We formalise our recovery algorithm and prove its safety. A recovered communication system is free from deadlocks, orphan messages and recovery errors. Our recovery algorithm incurs less communication cost (only affected processes are notified) and overall execution time (only required states are repeated). On top of our analysis, we design and implement a runtime framework in Erlang where failed processes and their dependencies are soundly restarted from a computed safe state. We evaluate our recovery framework on message-passing benchmarks and a use case for crawling webpages. The experimental results indicate our framework outperforms a built-in static recovery strategy in Erlang when a part of the protocol can be safely recovered.

1. Introduction
1.1 Motivation
Let it Crash recovery model. Despite the importance of fast and correct recovery in distributed systems, it is still difficult and error-prone to implement fault-tolerant components. A well-established fault-tolerance model is the Let it crash model, adopted by the programming language Erlang. In Erlang, rather than trying to handle and recover from all possible exceptional and failure states, one can instead let processes crash and let the runtime automatically recycle them back to their initial state. Failures and errors propagation are managed by organising the running processes in a hierarchical structure, called a supervision tree, where each process and its dependencies are monitored by a parent process, called supervisor.

While the Let it crash model has gained popularity and has recently been adopted in other commercial languages and frameworks (Go, Akka and Scala), it still faces two major technical problems. First, the correctness of recovery strategies relies on the assumption that a global process structure (i.e. a supervision structure) of the system is correctly written by a programmer. A recent study reveals a misconfiguration of the supervision tree is a common source of errors for recovery [27]. Second, supervision strategies have a fixed structure, hence they often recover too little or too many processes, and are unable to capture the dynamic nature of the communication dependencies between processes. This often leads to a redundant and/or a unsound recovery mechanism (as shown in Table 1 and explained in §1.2). Here a question is: Can we soundly generate recovery strategies, minimising the recovery overhead? This paper answers the question affirmatively using a theory of multiparty session types [17].

Multiparty Session Types (MPSTs) [17] is a typing framework for verifying protocol conformance of a system of distributed processes. When the global interaction pattern is specified as a multiparty session protocol (called global type), it is projected to a localised view of the protocol (called local type), which is then type-checked each process implementation. The framework (1) has been applied to several mainstream languages (e.g. Java [18, 21], Python [11, 24], MPI/C [26], Go [25]) as to ensure safety properties such as deadlock freedom and type-safety; and (2) has been extended with exception handling constructs (e.g. [5, 7, 11]), offering viable solutions for verifying and modelling expected failures. However, none of the above works is applicable to the Let it crash model, which targets recovery when a process fails unexpectedly. The class of unexpected software faults includes, for example, failures caused by corrupted data, a request timeout, buffer overflow, out of memory exceptions.

Our approach. In this paper, we apply MPSTs in a new direction: a MPST protocol is used to automatically ensure soundness of recovered processes. The key idea is to extract a flow of communications from multiparty session types which is in turn used to calculate all affected parties and recover the system from a globally consistent state. Our recovery framework relies on two design ideas: (1) messages that should be re-sent are identified based on the dependencies in the type structure of a process and (2) only affected participants are notified and recovered. To realise these ideas we propose a novel algorithm, which localises the scope of the recovery, analysing the communication flow, given as a multiparty protocol. The algorithm works by traversing a dependency graph, automatically inferred from a given global protocol. We calculate all dependencies in the graph, affected by a given state. On failure, the processes from the calculated paths are notified and recovered.

1.2 Trading Negotiation Use Case
We start by illustrating the difficulty of a sound and efficient recovery on independent (localised) chained interactions which are a common topology found in Erlang applications. Fig. 1 shows the protocol written as a global multiparty session type (Erlang code can be found in Fig. 8). We write $A \rightarrow B : m$ to denote a message exchange from participant $A$ to participant $B$ of a message $m$, we sometimes omit the message from the notation. A sequence of messages is denoted by a semicolon ‘;’. The notation
Our approach

Table 1: Comparison between Erlang and MPST-based recovery

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Fig. 2 (a)</th>
<th>Fig. 2 (b)</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>all</td>
<td>only E (unsound)</td>
<td>all</td>
</tr>
<tr>
<td>2</td>
<td>A1, A2, A3</td>
<td>only E (unsound)</td>
<td>nothing</td>
</tr>
<tr>
<td>3</td>
<td>all (inefficient)</td>
<td>only E (unsound)</td>
<td>nothing</td>
</tr>
</tbody>
</table>

Figure 2: Erlang supervision: (a) all-for-one and (b) one-for-one

others will be lost and E will be stuck (deadlock). Consider that we chose Fig. 2(a). Then all processes are restarted from the beginning, which works in Scenario 2. However, in Scenario 3, the participants have already completed the negotiation. Thus it is redundant to recover any interactions.

To summarise, the advantages of our approach are two-fold. First, executions ensured by our protocol recovery strategy is safe by construction. While supervision trees are manually built, and recovery strategies are sometimes implemented in an ad-hoc manner, our proposed recovery algorithm realises sound supervision structures and guarantees no lost messages and safety of recovered processes. Second, our strategy reduces the communication cost by notifying only the related participants. As shown in Table 1 part of the error propagation can be avoided when our restart strategy is employed.

We provide a prototype implementation in Erlang on top of Erlang’s built-in fault-tolerance semantics. Though we assume some conditions from Erlang such as asynchronous messages passing and ordered queues, our algorithm can be flexibly tuned to various assumptions on queues and messaging semantics (such as synchronous and asynchronous message passing without queues).

**Contributions** of this paper are given as follows:

1. We propose a recovery framework based on MPSTs and show how session type-based analysis is used to provide the correctness of fault-tolerant recovery (§ 2).
2. We formalise a recovery strategy (§ 3) and prove its safety properties. The recovered communication system is free from deadlock, orphan messages and reception error (§ 4).
3. We provide a design and implementation of our recovery strategy on top of runtime monitoring in Erlang (§ 5).
4. We implement several use cases and show that our recovery strategy is more efficient for common message-passing protocols comparing to the Erlang all-for-one supervision (§ 6).

The paper discusses the related work (§ 7) and concludes (§ 8). Appendix includes omitted definitions and proofs; and the implementation, benchmarks and use cases can be found in [29].

2. Overview of Multiparty Protocol-Induced Recovery

Our framework involves two stages: processing a given global protocol (type) and runtime supervision. The global type analysis from the first stage is used for runtime monitoring and recovering during the second stage.

2.1 Global Protocol Processing

The global type processing involves (1) creating a global recovery table (GRT) from a given global protocol (type) and (2) projecting the global protocol into local types and creating a finite state machine per each local type. The GRT prescribes which part(s) of the global protocol to be recovered on failure, and the set of processes to be notified, while the finite state machines are used to track the current state of each process during the session execution.

To generate a GRT, first we create a dependency graph. A dependency graph is a directed graph that models the causal depen-
dependencies between the actions in a protocol. The dependency graph is built by syntactic traversal of the global type. The nodes of the graph model the states of the global protocol, and the edges model the causal dependencies between the states. The recovery analysis is performed on the dependency graph.

Our analysis is built on two key points.

All input-output dependencies before the failed point should be recovered. We model stateless processes, hence a forwarded message should be re-sent by its initial sender, not by intermediate ones. Consider the following protocol (the failed point is underlined):

\[
A \rightarrow B; \quad B \rightarrow C; \quad (2.1)
\]

If \(C\) fails to receive the message from \(B\), then \(A\) should also resend the message to \(B\). This is because \(B\)'s mailbox is emptied and \(B\)'s output data to \(C\) may be depended on the message from \(A\).

All messages in the queue of the failed process should be re-sent. When a process fails, its queue is emptied. At compile time (when the graph processing is done), the state of the queue is unknown. Consider the following two examples:

\[
A \rightarrow B; \quad C \rightarrow B; \quad \text{and} \quad A \rightarrow B; \quad C \rightarrow B; \quad (2.2)
\]

If \(B\) fails after receiving the message from \(A\) (as shown in the left example), the message from \(C\) might be already in \(B\)'s queue. That is why our analysis will list both \(A\) and \(C\) as potential participants which should recover. Hence we notify \(A\) and \(C\) by sending a request for recovery. Since \(A\) already sent the message to \(B\), \(A\) and \(B\) should be recovered. However \(C\) has two choices: If \(C\) has not still sent the message to \(B\), it can ignore the request for recovery message; otherwise \(C\) should be recovered since its message in \(B\)'s queue was lost. Similarly if \(A\) fails (as shown in the right example), then both \(B\) and \(C\) should be notified.

Now consider the following protocol which has additional intermediate communications to (2.2):

\[
A \rightarrow B; \quad C \rightarrow B; \quad D \rightarrow C; \quad C \rightarrow B; \quad (2.3)
\]

There is an input-output chain from the failed node at \(B\) (underlined) to \(C\). The output from \(C\) \((C \rightarrow B)\) depends on an output from \(B\) \((B \rightarrow D)\), which had not occurred due to the failure, hence \(C\) and \(D\) are not notified.

2.2 Global Recovery Table

Fig. 3 (a) shows the dependency graph for our example and Fig. 3 (b) shows the global recovery table, generated by our algorithm. The algorithm explores all paths of the session graph connected to the failed node. Since the syntax of global types is finite, the length of such paths is limited so that the algorithm terminates. For each path, the algorithm works recursively on the edges, and maintains a dictionary that records the recovery points for each participant.

The GRT records the failed node (corresponding to the node of the global graph), which role has failed, and a reset (recovery) point for all roles, which depend on the failed role.

2.3 Runtime Supervision

Our supervision is setup at runtime when a session is started and two types of entities are created: local process supervisors and processes. The local process supervisors (also called monitors) track the state of the process they supervise. Each local supervisor has a finite state machine created from the local type (during the global graph processing stage). Whenever the process performs a communication action, the monitor inspects the current state.

In Fig. 4, we list four possible actions that the local supervisors (LS) can take after a failure. Once a process fails, its LS is notified. The LS performs a lookup in the Global Recovery Table (GRT) and notifies the LSs of the affected processes by sending them a request for recovery, containing the failed state. When the local supervisors receive request for recovery they query the GRT to retrieve their new state. If this state has not been reached yet, they “ignore” the request for recovery; otherwise they “restart” the process. The other processes remain unaffected (the second left-most in Fig. 4).

As an example, consider the case in (2.2). The initial failure message for \(B\)'s failure is sent to \(B\)'s LS; Then \(B\)'s LS sends the recovery messages to \(A\)'s LS and \(C\)'s LS. Then \(A\) is restarted (the second right-most in Fig. 4); and if \(C\) has not sent the message to \(B\) yet, then \(B\)'s LS remains unaffected, otherwise it is recovered.

3. Recovery Algorithm

This section defines a recovery algorithm that given a global type and a failed local type, returns a set of new local types which should be recovered. The recovery algorithm is formalised by defining the causal relation of global types and labelled transition relations.

3.1 Global and Local Types

Syntax For the syntax of types, we follow [13] which is the most widely used syntax in the literature. A global type, written \(G,G',...,\), describes the whole conversation scenario of a multiparty session as a type signature, and a local type, written by \(T,T',...,\), which abstracts session communication structures from each end-point’s view. \(\mathcal{A}\) (a,b,c,...) denotes a finite alphabet and \(\mathcal{P}\) is a set of participants fixed throughout the paper: \(\mathcal{P} \subseteq \{A,B,C,...,a,b,c,...,p,q,...\}\). The syntax of types is given as:

\[
G ::= p \rightarrow p': \{a_j,G_j\}_{j \in J} \mid \mu t.G \mid t \mid \text{end}
\]

\[
T ::= p^i\{a_i,T_i\}_{i \in I} \mid p^i\{a_i,T_i\}_{i \in I} \mid \mu t.T \mid t \mid \text{end}
\]

\(a_j \in A\) corresponds to the usual message labels in session type theory. We omit the carried types from the syntax in this paper, as we
are not directly concerned with typing processes. Global branching
type \( p \rightarrow p' : \{ a_i, G_j \}_{i \in I} \) states that participant \( p \) can send a
message with one of the \( a_i \) labels to participant \( p' \) and that interaction
described in \( G_j \) follows. We require \( p \neq p' \) to prevent self-sent
messages and \( a_i \neq a_i \) for all \( i \neq k \in I \). Recursive types \( \mu t.G \) are
for recursive protocols, assuming that type variables \((t, t', ...) \) are
guarded in the standard way, i.e., they only occur under branchings.
Type \( \text{end} \) represents session termination (often omitted). The function
rules(\( G \)) gives the participants of \( G \). Concerning local types, the
branching type \( p!\{ a_i, T_j \}_{i \in I} \) specifies the reception of a message
from \( p \) with a label among the \( a_i \). The selection type \( p\mid \{ a_i, T_j \}_{i \in I} \) is its dual. The remaining type constructors are as for global types.

**Projection** The relation between global and local types is formalised by projection [13, 17]. For projection of branchings, we use a merge operator [13], written \( T \sqcup T' \), ensuring that if the observable behaviour of the local type is dependent on the chosen branch then it is identifiable via a unique choice/branching label.

**Definition 1 (Projection).** The projection of \( G \) onto \( p \) (written \( G[p] \)) is defined as:

\[
p \rightarrow p' : \{ a_i, G_j \}_{i \in I} \xrightarrow{q} \begin{cases} p!\{ a_i, G_j \} \mid q \in \{ q' \mid \mu t. G[p'] \mid \mu t. G[p] \} & \text{if } q = p \\
\sqcup a_i, T_j & \text{if } q = p' \\
\end{cases}
\]

where \( \text{fv}(G) \) denotes a set of free type variables in \( G \). The merging operation \( \sqcup \) is defined as a partial commutative operator over two types such that:

- \( T \sqcup T = T \) for all types;
- \( p!\{ a_i, T_j \}_{i \in I} \sqcup p!\{ a_i, T_j' \}_{i \in J} = p!\{ a_i, T_j \}_{i \in I} \cup \{ a_i, T_j' \}_{i \in J} \) and homomorphic for other types (i.e., \( \mathcal{C}[T] \sqcup \mathcal{C}[T] = \mathcal{C}[T \sqcup T] \) where \( \mathcal{C} \) is a context for local types). The merging operation between \( T_1 \) and \( T_2 \) is defined only if \( T_1 \sqcup T_2 \) is defined. We say that \( G \) is well-formed if for all \( p \in \mathcal{P}, G[p] \) is defined.

Consider the global type given below.

\[
\mu t. A \rightarrow B : \{ a, B \rightarrow C : b.t, d.B \rightarrow C : e.\text{end} \}
\]

Then the projections on \( A, B \) and \( C \) are given as follows:

- \( A \)'s local type: \( \mu t.B!\{ a.t, d.\text{end} \} \).
- \( B \)'s local type is: \( \mu t.A!\{ a.C!b.t, d.C!e.\text{end} \} \).
- \( C \)'s local type is: \( \mu t.B!\{ b.t, e.\text{end} \} \).

The projection for role \( C \) uses the merge operator. Merge is also used to project role \( D \) from our running example from Fig. 1

### 3.2 Errors Prevented by Protocol-Induced Recovery

The theory of MPST guarantees that a system of communicating processes, where each process conforms to a local type (projected from the same global type), is safe, i.e., it is free from deadlocks, reception errors and orphan messages. In this subsection, we demonstrate that existing recovery approaches do not preserve safety, hence they can introduce all of the above-mentioned errors. We look at two popular approaches of recovery [28] which can be used in addition (or instead of) supervision trees: (1) recovery by resending the undelivered messages and (2) restarting processes from their initial state. For each approach we demonstrate a potential error(s).

\[
\begin{align*}
G_1 &= B \rightarrow C; A \rightarrow C; \quad G_2 = B \rightarrow C; A \rightarrow C; \quad G_3 = A \rightarrow B; \\
B \rightarrow A; C \rightarrow A; D \rightarrow C; & \quad B \rightarrow A; D \rightarrow C; \quad C \rightarrow B : l_1; \\
A \rightarrow B; & \quad C \rightarrow B : l_2;
\end{align*}
\]

We omit the labels from local and global types except branching.

**Figure 5** Recovery errors: (1) deadlock, (2) orphan message error and (3) reception error

Fig. 5 shows global and local types for three protocols. We underline the failed role, e.g., \( A \) in the global type and mark with a box the state of the local types after recovery.

**Recovery by resending a message (a deadlock)** The process \( A \) fails before receiving the message from \( B \). A naive recovery strategy might be to resend the unsuccessfully delivered message from \( B \) to \( A \), not taking into account the existence of other parties (\( C \) and \( D \)). When \( A \) recovers, all contents of \( A \)'s queue are deleted. Since messaging is asynchronous, it is possible that \( C \) has already sent the message to \( A \). This message will be lost when the queue is deleted. \( C \) is at a state of receiving a message from \( D \), while \( A \) is stuck waiting for the message from \( C \). The processes end up in a deadlock.

**Recovery by restarting processes from beginning (orphan message error)** We assume the same failure with the above case, but a different recovery strategy. Instead of resending the failed message we recover both affected processes by restarting them from the beginning. No messages are lost and no deadlock occurs: however, both \( A \) and \( B \) will repeat their interactions (\( B \rightarrow C \) and \( A \rightarrow C \)). Since \( C \) has already received these messages, the orphan messages will stay in the queue of \( C \) and will not be consumed.

**Partial recovery of processes (reception error)** Assume the protocol \( G_1 \) and \( G_2 \) fails at \( A \rightarrow B \). Suppose \( C \) has already sent the message \( l_1 \) when \( B \) failed but only \( A \) and \( B \) are recovered. Then the recovered \( B \), which should receive \( l_1 \) from \( C \), will receive the wrong message \( l_2 \) from \( B \) since \( B \)'s queue is erased by the recovery.

### 3.3 A Dependency Analysis on Global Types

As shown in § 2.1, the recovery algorithm needs a dependency analysis of global types. We define the two key dependency relations (\( <_L \) and \( <_C \)) used in the algorithm.

**Session graphs** Global types can be seen isomorphically as session graphs, that we define in the following way. First, we annotate in \( G \) each syntactic occurrence of subterms of the form \( p \rightarrow p' : \{ a_i, G_j \}_{i \in I} \) with a node name (denoted by \( n, n', n_1, n_2, \ldots \)). Then, we inductively define a function \( \text{node}_G \) that gives a set of nodes (or the special node end) for each of the syntactic subterm of \( G \) as follows:

\[
\begin{align*}
\text{node}_G(\mu t. G') &= \text{node}_G(G') \\
\text{node}_G(\text{end}) &= \text{end} \\
\text{node}_G(n : p \rightarrow p' : \{ k, G_j \}_{j \in J}) &= n \\
\text{node}_G(t) &= \text{node}_G(\mu t. G') \text{ if } \mu t. G' \in G \text{ and } t \in \text{fv}(G')
\end{align*}
\]

We define \( G \) as a session graph in the following way: for each subterm of \( G \) of the form \( n : p \rightarrow p' : \{ k, G_j \}_{j \in J} \), we have edges from \( n \) to each of the \( \text{node}_G(G_j) \) for \( j \in J \). We also define the functions \( \text{prefix}(n) \) and \( \text{roles}(n) \) that respectively give the prefix (\( p \rightarrow p' : a \)) and participants (\( p, p' \)) respectively.

**Example 2 (Session graph).** To illustrate session graphs on recursive global types, we augment the main body of our running exam-
ple with a recursion in the first case of the choice, as shown below:

\[
\mu E \rightarrow D \rightarrow E :\{\text{accept } E \rightarrow D : \text{reject } \epsilon, \text{ reject } E \rightarrow D : \text{accept } E \}\}
\]

Then its graph representation with the initial node \( n_1 \) is given:

![Graph representation of the algorithm](image)

The edges of a given session graph \( G \) define a successor relation between nodes, written \( n < n' \) (omitting \( G \)). Paths in this session graph are referred to by the sequence of nodes they pass through. The empty path is \( \epsilon \). The transitive and reflex closure of \( < \) is \( \ll \).

**Causality and causality chains** We define causality relations in a given \( G \) by the relations \( < \) and \( \ll \). The dependency relation \( < \) represents the order between two nodes which has a common participant; and the IO-relation \( < \ll \) asserts the order between a reception by a principal and the next message it sends. Formally,

\[
n_1 < n_2 \quad \text{if} \quad n_1 < n_2 \text{ and roles}(n_1) \cap \text{roles}(n_2) \neq \emptyset
\]

\[
n_1 \ll n_2 \quad \text{if} \quad n_1 \ll n_2 \text{ and } \text{pxf}(n_1) = p_1 \rightarrow p : a_1 \text{ and } \text{pxf}(n_2) = p_2 \rightarrow p : a_2
\]

An input-output dependency (IO-dependency) from \( n_1 \) to \( n_2 \) (denoted by \( n_1 < \ll n_2 \)) is a chain \( n_1 < \ll ... \ll n_2 \) (\( n \geq 1 \)). In Example 2, we have \( n_1 \ll n_2 \), and \( n_1 \ll n_2 \), but \( n_1, n_2 \) are not related by the IO-dependency.

### 3.4 Recovery Algorithm

**Affected nodes** We define the algorithm to decide the set of the affected nodes \( \mathcal{N} \) when a participant \( p \) in a global type \( G \) fails. The algorithm is shown in Figure 6, where \( n_i \) is the failed node, it corresponds to the state of \( p \) at the time of failure. Below we explain each step of the algorithm. We write \( n = p \rightarrow q \) means \( \text{pxf}(n) = p \rightarrow q : a \) for some \( a \).

**Step 1: Initialisation (Lines 1-2)** We initialise a set of affected nodes \( \mathcal{N} \) to include the failed node \( n_i \) and successors of the failed node \( n_i \) that have \( p \) as a sender in their prefix (\( n = p \rightarrow p \)). The latter ensures that the output dependencies of \( p \) are added to \( \mathcal{N} \) as to recover messages that are lost as a result of deleting \( p \)'s queue. This scenario is shown in example (2.2) in § 2.1.

A set of unaffected nodes \( \mathcal{Y} \) is initialised as nodes \( n' \) that are IO-dependent from the failed node or the node where \( p \) is the sender (\( n = p \rightarrow p \)). As explained in example (2.3), even if \( n_i \) fails, they do not have to be recovered.

**Step 2.1: Traversing input-output dependencies, Backward traversal of \( < \ll \) (Line 4)** A set of backward affected nodes \( \mathcal{N}^{\ll} \) consists of nodes gathered by traversing IO-dependencies. As explained in example (2.1) in § 2.1, to recover a given state, we need to track the IO-chains preceding that state. We also add the forwarding IO-chains from the unaffected node if they are dependent from affected nodes.

**Step 2.2: Forward traversal of \( < \ll \) (Line 5)** A set of forward affected nodes \( \mathcal{N}^{\ll} \) consist of all direct dependent nodes from backward affected nodes. This prevents unspecified reception error. We extract both affected and non-affected nodes since they do not have to be traversed in the next iteration.

**Step 3: Termination condition (Line 5-7)** We stop if there are no new affected nodes. Otherwise we repeat Step 2.

After we obtain the set of affected nodes, following the algorithm in Fig. 6, we calculate the recovery point for each participant in the protocol. Intuitively, a recovery point is a local type that is assigned to a participant as a result of a recovery.

**Algorithm: Calculating affected nodes**

**Input:** \( \mathcal{N} \) (a failed node), \( p \) (a failed role)

**Output:** \( \mathcal{N} \) (a set of affected nodes)

1. \( \mathcal{N}^{\ll} = \mathcal{N}^{\ll} = \{ n \mid n_i < n \land x \rightarrow \} \cup \{ n_i \}
2. \( \mathcal{Y} = \{ n \mid (n_i < n' \land n' = \ll \} \cup \{ n_i \}
3. \text{repeat}
4. \( \mathcal{N}^{\ll} = \{ n \mid n \ll n' \land n' \in \mathcal{Y} \}
5. \mathcal{N}^{\ll} = \{ n \mid n' = n \land n_1 \in \mathcal{N}^{\ll} \}
6. \mathcal{N} = \mathcal{N} \cup \mathcal{N}^{\ll} \quad \mathcal{Y} = \mathcal{Y} \setminus \mathcal{N}^{\ll}
7. \text{until } \mathcal{N}^{\ll} = \mathcal{N}^{\ll} = \emptyset
8. \text{return } \mathcal{N}

Below we write \( G/n_i \) to denote a subterm (subgraph) of \( G \) whose occurrence is \( n_i \). For example, if we take the session graph in Example 2 \( G/n_4 = E \rightarrow D : \text{accept } \epsilon \) and \( G/n_5 = E \rightarrow D : \text{reject } G \), we write the outputs by the syntax of global types.

We write \( T \geq T' \) to denote \( T' \) is a subterm of \( T \). For example, \( \forall b \in B \cdot \exists b \cdot b \cdot \exists b' \cdot b' \cdot b' \cdot b' \cdot b \cdot b' \cdot b' \cdot b' \cdot b' \cdot b' \cdot b' \). We define \( \max(T \cup \{ T_i \mid i \in I \}) = T_j \) if for all \( i \in I, T_i \geq T_j \).

**Definition 3** (Recovery point). Assume that \( \mathcal{N} \) is the set of affected nodes when \( n_i \) in global type \( G \) failed and the participant \( p \in \text{roles}(n_i) \) failed. Assume \( G/n_i \ | \ | = T_p \). We then define, for each \( q \in \mathcal{Y} \) (1) \( f_G(T_p, q) = \max\{G/n_i \mid q \land n \in \mathcal{N}, q \in \text{roles}(n_i)\}; \) or (2) \( f_G(T_p, q) = \emptyset \) if \( q \notin \text{roles}(n) \) for all \( n \in \mathcal{N} \). We call the nodes corresponding to the recovery points, recovery nodes.

By the finiteness of the session graph, we have:

**Proposition 1.** The recovery algorithm in Figure 6 terminates.
We denote the safety properties of recovered processes.

4. Semantics and Properties of Multiparty Induced Recovery

This section first presents the recovery semantics using the recovery point function. Then we prove that our recovery algorithm satisfies the safety properties of recovered processes.

4.1 Recovery Semantics

The labelled transition system ( LTS) for a local type

We start from a labelled transition relation between local (endpoint) types defined in §3.1. We first define observables, called actions ( ℓ, ℓ′, ... ). An action ℓ denotes the sending or the reception of a message of label a from p to p′ and the reaction message ↑.

ℓ := pp′!a | pp′?a ↑

We then define the LTS over local types starting from an individual local type. The relation T → T ′, for the local type of participant p, is defined as:

\[
\begin{align*}
\text{OUT} & : q!\{a_i, T_i\}_{i \in I} \xrightarrow{p\eta a_i} T_j \quad [\text{IN}] \quad q?\{a_i, T_i\}_{i \in I} \xrightarrow{\eta a_i} T_j \\
\text{MU} & : T \xrightarrow{\mu T} T' \quad [\text{REC}] \\
& \quad f_G(T_p, p) = T'_{p}
\end{align*}
\]

The first three rules are standard. The rule [REC] represents the case participant p fails at the point of T and recovers as T ′. It is defined by the function \( f_G(T_p, p) = T'_p \)

The participant p enqueues a value to FIFO queue \( w_{pq} \) of a channel \( pq \), and participant q dequeues a value from \( w_{pq} \). In addition, we define the two cases when participant p fails at \( T_p \) (item 2 below).

The item (a) is the case when participant q needs to recover as local type \( T'_p \). In this case, we clean up its input queues. The item (b) is the case participant q does not need to recover. In this case, we do not have to change the configuration for q. Note by the case (2-a), failed participant p always cleans up its input queues.

Definition 4 (A configuration and its LTS). A configuration \( s = (\vec{T}, \vec{w}) \) of a system of local types \( \{T_p\}_{p \in \mathcal{P}} \) is a pair with \( \vec{T} = (T_p)_{p \in \mathcal{P}} \) and \( \vec{w} = (w_{pq})_{p \in \mathcal{P}, q \in \mathcal{P}} \) with \( w_{pq} \in \mathbb{A}^* \). The initial configuration of G is \( s = (\vec{T}, \vec{w}) \) with \( w_{pq} = \epsilon \) and \( T_p = G \mid p \). A final configuration is \( s = (\vec{T}, \vec{w}) \) with \( T_i = \text{ord} \). We then define the transition system for configurations starting from the initial configuration of G. For a configuration \( s = (\vec{T}, \vec{w}) \), the transitions of \( s \rightarrow s' = (\vec{T}', \vec{w}') \) are defined as:

1. \( T_p \xrightarrow{pq\eta a} T'_p \) and \( w_{pq} = w_{pq} \cdot a \) and \( T'_p = T''_p \) for all \( p' \neq p \); or
2. \( T_p \xrightarrow{pq\eta a} T'_p \) and \( w_{pq} = \epsilon \) and \( T'_p = T''_p \) for all \( p' \neq q \) with \( w''_{pq} = \epsilon \).

We recall Fig. 3. (1) Participant \( B_2 \) fails at node 3. By Definition 3, \( f_G(\{B_1, B_2\}) = B\) \( \mid B \) where \( B_{B_1} = G \mid B_1 \) and \( B_{B_2} = G \mid B_2 \). Also for all \( p \notin \{B_1, B_2\} \), \( f_G(\{B_1, B_2\}) = 0 \). By Definition 4(2), \( B_{B_1} \rightarrow B_2 \). We also set the input queues of \( B_1 \) and \( B_2 \) to be empty. Hence \( w_{B_1} = w_{B_2} = \epsilon \). By (2-a), we set \( B_{B_1} = B_1 \). Except \( B_1 \) and \( B_2 \), the queues and local types are unchanged. Note that if \( p \neq B_1 \), \( B_{B_1} \) is empty before \( B_2 \) fails (since \( f_G(\{B_1, B_2\}) = 0 \)). The cases when \( A_2 \) fails at node 1 and \( C \) fails at node 4 can be calculated similarly.

Example 5 (Trading Negotiation). By the algorithm, nodes 0 and 1 are added to \( \mathcal{N} \) since there is a backward IO-dependency \( n_0 \leftarrow n_1 \leq n_4 \); also node 5 is added to \( \mathcal{N} \) since \( n_4 \leq n_5 \). From node 5, there is a backward IO-dependency \( n_2 \leftarrow n_3 \leq n_5 \), hence nodes 2 and 3 are also added to \( \mathcal{N} \). Hence for all \( p, f_G(\{B_1, B_2\}) = G \mid p \). By Definition 4(2-a), all participants will restart from the beginning of the protocol.

4.2 Main Results: Transparency and Safety

Our recovery algorithm guarantees the transparency of the recovery procedures and the safety of configurations in the presence of failures. Transparency means that once a configuration recovers to some state after a failure, there are always transitions which can reach another state unaffected by failures. Hence we can recover the configuration as if there were no failure. The safety includes the three properties, reception error freedom, orphan message freedom and deadlock-freedom, which were originally introduced in communicating finite state machines [2, 6] as desired properties. These are ensured by the multiparty session type theory without failures [12, 13, 22].
of G. If \( s \xrightarrow{f} s' \xrightarrow{g} s'' \), then there exists \( s \xrightarrow{f} s' \) where \( s \xrightarrow{f} s' \) and \( f \) does not contain \( f \).

**Theorem 1 (Transparency).** Suppose \( s_0 \) is the initial configuration of G. If \( s_0 \xrightarrow{f} s' \xrightarrow{g} s'' \), then there exists \( s \xrightarrow{f} s' \) where \( s \xrightarrow{f} s' \) and \( f \) does not contain \( f \).

**Proof.** We first prove that a set of local types defined by \( f \) for a given failed type \( T_p \) and G, i.e., \( \{ f_G(T_p, q) \} \subseteq \mathcal{P} \) forms a projection of a subgraph of G. We also prove that if \( f_G(T_p, q) \) is empty, then its queue was empty before recovery. This means that when we restart a set of processes after a failure of some process, all processes will always recover from some point of a subprotocol of the original G. See Appendix C.

The following definitions of configuration properties follow [6, Definition 12]. We recall that the examples from Fig. 5 could easily introduce these errors if an incorrect recovery strategy is deployed.

1. Configuration \( s \) is a deadlock configuration if \( s \) is not final, and \( w = \mathcal{E} \) and each \( T_p \) is a branching type, i.e., all types are blocked, waiting for messages.
2. Configuration \( s \) is an orphan message configuration if all \( T_p \in \mathcal{P} \) are end but \( w \neq \mathcal{E} \), i.e., there is at least an orphan message in a buffer.
3. Configuration \( s \) is an unspecified reception configuration if there exists \( q \in \mathcal{P} \) such that \( T_q \) is a branching, and \( T_p \notin \mathcal{P}, T_q' \) implies that \( w_{pq} > 0 \) and \( w_{pq} \notin \mathcal{A} \), i.e., \( T_q \) is prevented from receiving any message from buffer \( p \), meaning type error.

**Theorem 2 (Safety).** Any reachable configuration from \( s_0 \) which is an initial configuration of well-formed G is free from deadlock, an orphan message and a reception error.

**Proof.** By Theorem 1 and [22, Theorem 3.1].

### 5. Implementation of Multiparty Induced Recovery

We implement the recovery semantics and algorithm (as explained in § 4.1) in a new Erlang library. We use the language Scribble [30] to describe multiparty session protocols.

Our system consists of the following three layers:

1. **Scribble module**: a module for processing global Scribble protocols. The module takes a Scribble protocol as an input and generates (1) local types (following Definition 1) and (2) global recovery tables (following the recovery algorithm from § 4.1).

2. **Monitoring runtime**: a component that implements the runtime semantics for protocol verification and recovery (following Definition 4). The runtime creates a monitor process per Erlang process. A monitor checks, at runtime, that messages sent and received by a process correspond to its local type. In the case of failure, monitors restart their respective processes.

3. **An interface for local processes**: this module provides the basic functionality for a process to be eligible for verification and recovery.

For developers to use the system there are two requirements. First, define the process interactions into a Scribble protocol. Second, implement a gen_protocol process for each role in a protocol. The role and the protocol must be specified as a part of the process initialisation. The process is also required to implement message handlers for all interactions in the protocol. Hence, developers are implementing processes using a handler-based API as customary in Erlang/OTP, while a monitor attached to each process ensures that the behaviour of the system follows the semantics in § 4.

We explain the processing of Scribble protocols and how to program with gen_protocol in § 5.1 and § 5.2 respectively.

### 5.1 Scribble Protocols for Recovery and Verification

The correspondence between Scribble constructs and their multiparty session type (MPST) counterparts, as well as the global Scribble protocol for the example in Fig. 1, are given in Appendix B. Here we list only important details for the runtime semantics.

**Protocol processing** Once the protocol is written, the Scribble tool [30] automatically checks the correctness of the protocol and generates finite state machines (FSMs). While our recovery analysis is solely based on global protocols, we use the correspondence between local types and finite state machines [12] to monitor and track the execution state for each process.

**Storing global recovery tables** Our tool parses a global Scribble protocol to create a dependency graph (as explained in § 3.3). Then it calculates the corresponding Global Recovery Table (GRT), following § 4.1, and stores it in a database. This is done statically before an application is started. We generate a GRT per protocol. The scheme for the database table, holding the GRT records, follows the shape, presented in Fig. 3; i.e., for every protocol state, a map of affected roles and their reset points is stored. For a persistent storage, we use Mnesia, which is optimised for fast query processing on a read-only table where locks are not needed, as in our case. At runtime the GRT records are accessed only when a role has to retrieve its new state due to a process failure. Mnesia is a distributed database, which provides replication capabilities, hence the GRT can be accessed from different Erlang nodes.

### 5.2 Erlang Programming with Multiparty Session Protocols

**Monitor** A monitor intercepts all incoming/outgoing messages associated to its linked process. Monitors correspond to the local supervisors shown in Fig. 4 in § 2. When a monitor is spawned, it is parameterised with a local protocol. Every time a process sends a message, its monitor checks the message is correct w.r.t the local type, if so the message is sent to the destination monitor, and from there dispatched to the corresponding process.

**Endpoint processes** Endpoint (gen_protocol) processes implement the business logic for a role in a protocol. The endpoint processes define message handlers and react upon received messages. The order of messages is not specified since the verification process (the monitor associated to the endpoint process) ensures the messages follow the order in the protocol.

For an endpoint process to be part of a protocol, it should implement a custom behaviour gen_protocol. Behaviours in Erlang are similar to abstract classes. They encapsulate a common pattern (behaviour) and expose a set of required methods to be implemented. For example, our gen_protocol behaviour checks at compile-time that all message handlers implemented in a process module have a matching label in the local type of the process. For example, if a partial protocol is: A \( \rightarrow \) B : quote, the protocol implementing B is verified to have, as part of its interface, a function named quote.

We implement the gen_protocol behaviour as a modification of gen_server, which is an implementation of a generic server, part of the standard Erlang/OTP libraries. It receives messages from the process mailbox and dispatches them to the message handlers defined in the process. For example, if a message of the form `{(123, sum, 1, 2)}` is received in the mailbox of the process with process id that equals (123) the gen_server dispatches the message by invoking the function `sum(1, 2)`.

**Communication between endpoints** Endpoint processes communicate via the API function: `role:send(Id, Role, Method, Args)`. The first parameter `Id` is the id of the monitor linked to the process, `Role` is the name of the destination role as given in the protocol. For example, Line 14 in Fig. 8 specifies sending a message to role `E`, the notation `?E` in Erlang is used to annotate local constants, e.g. `?E`. The parameter `method` is a label for the message.
be written for a developer wants to use our recovery strategy they have to replace that follows our recovery strategy. Hence, if protocol_supervisor

5.3 Supervision

function of the supervisor, as customary in Erlang applications. supervisor. Finally, the protocol can be started by invoking the start

supervisor for the endpoint processes. A supervisor should specify

Role

Figure 8: Message handlers for endpoint processes

being sent. For example, if a protocol specifies quote(int) then method is quote. The parameter, Args, stands for payloads.

Message handlers as callbacks An endpoint process implementation consists of defining message handlers for protocol messages. A part of Erlang code for the Trading Negotiation is given in Fig. 8. The code snippet displays the callbacks required for the modules implementing the endpoint processes for roles A, C, D and E. Note that A and B do not have any interactions after sending an initial quote message and no handlers are needed. Hence, A sends a message to A1 in the body of the initialisation function init. Line 3 in Fig. 8. The C process and the D process implement the function quote, accept and reject. In quote both C and D simply resend the received message to the process E. The internal choice on E is implemented as a guard on the message handler quote. The guard compares the values received from C and D and sends accept to whoever sends the highest quote. As customary in Erlang, all handlers have a parameter State which is used to thread the state of the process between message handlers. For example, Line 14 saves, in the variable prev, the received value Val. Then on Line 21 prev is used as a guard to determine which message handler to be invoked.

Starting a protocol The last requirement is implementing a supervisor for the endpoint processes. A supervisor should specify a supervisor type and a list of processes to be started. A simplified supervisor definition for a one-for-all supervisor has the form [{one_for_all, Processes}, where Processes is a list of processes definitions. We have implemented a custom supervisor type protocol_supervisor that follows our recovery strategy. Hence, if a developer wants to use our recovery strategy they have to replace {one_for_one} strategy with protocol_supervisor as a type of the supervisor. Finally, the protocol can be started by invoking the start function of the supervisor, as customary in Erlang applications.

5.3 Supervision

We build our recovery strategy on top of the runtime protocol verification, provided by role and gen_process. Decoupling of a protocol checker (role) and an endpoint process (gen_process) is essential for the recovery. In this way, processes do not send messages directly to the other endpoints and as a result when a process fails only its role is notified about the new process id. Therefore the failure is transparent to the other endpoint processes.

The implementation of our recovery mechanism draws on the Erlang feature of links, a mechanism for creating a bidirectional

Figure 9: Supervision hierarchy

link between processes. It ensures that terminating processes emit exit signals to all linked processes. We use link to connect a running process to its role.

A running system with three participants is shown on Fig. 9. The figure displays a supervision structure and links (denoted as dotted red arrows) created by our runtime. The processes are grouped by a protocol_supervisor, implemented as an extension of the Erlang’s one-to-one supervisor. If a process dies, only this process is restarted by the supervisor. When a role is created during a process initialisation, the role receives the process id and links to it, which ensures the role will be notified if the process fails.

Failure handling The recovery mechanism after a process fails consists of three key parts:

(1) Notification of failure. A role receives the system message EXIT when a process fails. Then the role broadcasts a message FAIL, that contains the failed state, to the other affected roles. The set of affected roles is retrieved from the GRT. Only the state number is broadcasted, because it identifies the state. The projection ensures a unique correspondence between local states and states in the global dependency graph.

(2) Obtain reset points. When a role receives a message FAIL or EXIT it queries the GRT, and retrieves the reset state for the new process. A state represents a node in the local finite state machine (FSM). The role updates the current FSM state.

(3) Restart a process When a role receives a message FAIL it sends a kill message to its linked process. When a process dies, either because of a failure, or as a result of a kill message, it is restarted by the protocol_supervisor. During process initialisation, the role receives the id of its role and sends the role its new id so the role can establish the link.

6. Use Cases and Evaluations

The aim of the evaluation in this section is to demonstrate the applicability of our recovery strategy (called hereafter protocol-recovery) to several typical concurrency patterns from open source projects and the literature [16, 19]. The overhead of protocol-recovery comes from (1) the overhead of propagating the error and (2) the lookups performed on the global recovery table.

We compare protocol-recovery against the Erlang all-for-one supervision strategy. We have organised the benchmarks into three categories: we evaluate (1) a real world use case, a protocol for crawling web pages, by inserting failures at random states as to measure the average performance gain by protocol-recovery; (2) a set of three typical message-passing protocols, by inserting a failure at a specific state in the protocol as to measure the maximum performance gain by protocol-recovery; and (3) the execution times for the three scenarios of our running example, confirming that in two out of the three scenarios from § 1, all-for-one is less efficient than protocol-recovery.

In summary, the use of protocol-recovery results in faster protocol execution times (up to 52%) than all-for-one in cases where fewer processes are recovered. In cases where protocol-recovery re-
covers all processes, the overhead is small (up to 7%). We compare with all-for-one only since other static recovery strategies result in an inconsistent (unsound) monitor state (see Table 1).

We implemented all programs using the Erlang API, presented in § 5. For each example, we give the global type adopting the notation from [24, § 3.2] and [14] as to express parameterised processes. For example, $P[i:1..n]$ denotes a range of processes of type $P$ where $i$ is between 1 and $n$ and $i$ binds the rest of the free occurrences of $i$ in the rest of the global type. The examples are summarised in Table 2 and Fig. 10. Table 2(a) shows the number of roles (participants) for each protocol (#roles), the number of states in the dependency graph (#states) and the time (in seconds) for calculating the global recovery time (GRT). Each example is parameterised on an integer $n$, which determines the number of roles, and thus the number of states in a protocol (as shown in Fig. 2) and the results displayed in Fig. 10 are for $n=100$. We also show the roles that are recovered (affected roles) for the protocols where we report results displayed in Fig. 10 are for $n=100$.

Table 2: GRT generation time ($n = 100$)

<table>
<thead>
<tr>
<th>Example</th>
<th>#roles</th>
<th>#states</th>
<th>GRT (sec)</th>
<th>affected roles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Crawler [19]</td>
<td>$2^n+2$</td>
<td>$4^n$</td>
<td>0.45</td>
<td></td>
</tr>
<tr>
<td>MapReduce [19]</td>
<td>$n+1$</td>
<td>$n+2$</td>
<td>0.11</td>
<td>$W[1]...W[n]$</td>
</tr>
<tr>
<td>Ring [19]</td>
<td>$n$</td>
<td>$2^n$</td>
<td>0.16</td>
<td>$W[1]...W[n]$</td>
</tr>
<tr>
<td>Calculator [16]</td>
<td>$n+1$</td>
<td>$4^n$</td>
<td>0.75</td>
<td>$A[1]$</td>
</tr>
<tr>
<td>Trading Negotiation</td>
<td>$2^n+1$</td>
<td>$2^n+4$</td>
<td>0.17</td>
<td>in Table 1</td>
</tr>
</tbody>
</table>

6.1 Web Crawler Use Case

We start with an open source project, a web crawler example.1 A web crawler protocol specifies the coordination between multiple processes. The task is split into the four stages: (1) connecting to a webpage; (2) downloading its content; (3) parsing the content; and (4) indexing the contents of a set of web pages. Multiple instances of Downloader and Parser processes can run in parallel. In addition, the task of indexing a parsed content is normally delegated to a third-party database with capabilities for storing large volumes of data. The protocol is given below:

```
Downloader[i:1..n] → Parser[i]:parse;
Parser[i] → Indexer:index; Indexer → Master: url;
```

The protocol consists of four processes: a Downloader process downloads the source of a page given the page url address; a Parser process parses the source of the page; an Indexer process delegates the parsed result to an external service for indexing; and a Master process carries the information of visited pages. Note that after the Indexer is done processing a webpage, it notifies the Master which url has been processed.

**Failure injection** A robust implementation of this use case is challenging since an implemented program relies on several external services, notably the pages that are being crawled and connected to an external database for storing indexed results. Therefore, numerous failures are possible, for instance, requests can time out, the parser can choke on the input, an error in the indexing service can occur due to a large number of requests. To test all the different scenarios, we have implemented a randomised failure injection. At the beginning of a protocol execution we chose randomly a failing state and when this state is reached it has a 20% chance to fail by performing division by zero. We execute this scenario 100 times and on the graphs on Fig. 10 (a) the results are ordered based on the number of failures affecting each execution. For example, the last bar shows the execution time when a protocol has to recover five times before reaching a failure-free execution. On average our protocol-recovery gives a better performance, especially in the case of multiple failures.

During the experimental evaluation, we tried different failure strategies, such as assigning a probabilistic failure value to each process. This approach resulted in many processes failing simultaneously. Our recovery managed to complete the tasks, while all-for-one could not since it quickly reached the maximum number of allowed restarts. Although it is anecdotal, this experiment shows that supervision strategies are a source of errors, and highlights the importance of a sound recovery.

6.2 Micro Benchmarks

For each example, we give the global type and discuss the result of protocol-recovery in terms of (1) number of affected participants (Table 2) and (2) overhead (Fig. 10 (b)). For the latter, we compare the execution times for completing the protocol without a failure, and with one failure followed by a subsequent recovery.

**1 MapReduce** Below we show a typical parallel protocol, where a Master process splits a task between several Workers. Each worker performs its sub task and notify the Master by sending a reduce message.

---

1 The example is implemented in several github projects such as [https://github.com/Foat/articles/tree/master/akka-web-crawler](https://github.com/Foat/articles/tree/master/akka-web-crawler)

Figure 10: Execution time for (a) Web Crawler, (b) Ring, MapReduce and Calculator and (c) Trading Negotiation; the colours of the bars on the graph correspond to (no failure protocol-recovery all-for-one)
Master → Worker| i : 1..n | : map;
Worker| i | → Master : reduce;

Failure: The Master fails after sending map to everyone.
Result: The execution time with recovery depends on the computation intensity of the task performed by each worker. In our benchmark each Worker|i| sorts a list of 10000 elements. MapReduce in Fig. 10(b) shows that after a failure, protocol-recovery outperforms the all-for-one recovery taking only 20% of the time to complete the protocol (the time before the recovery starts).

(2) Ring We consider a common pattern of chained interactions between the number of processes where each process (A[i]) sends a ping message to its neighbour (A[i+1]). When process A[n] receives ping, it starts a chain of pong messages. Its protocol is:

A[1:1..n-1] → A[i+1]: ping;
A[i:1..n-2] → A[i-1]: pong;

Failure: A process A[k] fails before sending a pong message.
Result: The total of n-k+1 processes are restarted (these are processes n, n-1, ..., k). Fig. 10(b) shows the execution time when k = n-1 which requires restart of only two processes and thus the significant performance gain (protocol-recovery outperforms the all-for-one recovery by 52%).

(3) Distributed Calculus This protocol is a modification of the ring protocol. Processes cooperate to solve an equation. Each process (A[i]) calculates an expression (by sending expr to a calculator C) and resends the continuation (the rest of the equation) to its neighbour. Then the process A[i] waits for result from C and for the result of its neighbour A[i+1]. When both are received it sends the total to A[i-1]. Its global type is given as:

A[1:1..n-1] → C: expr;
A[i] → A[i+1]: cont;
C → A[i]: result;
A[n] → C: expr;
C → A[n]: result;
A[i:1..n-2] → A[i-1]: result;

Failure: C fails after processing a message form A[i].
Result: All processes are recovered since they are all connected by the IO-dependencies and each output depends on a previous input, hence this benchmark does not incur performance gain.

6.3 Trading Negotiation
We demonstrate the result of recovery on the failure scenarios explained in § 1.2 (displayed in Fig. 10(c) as Sc1, Sc2 and Sc3). We have spawned 100 processes for Alice’s and Bob’s group. The results of an execution without recovery of the protocol is given in Fig. 10(a) (denoted as Sc0). In case of recovering fewer processes, as in Sc1 and Sc3, the protocol completes faster as we apply protocol-recovery than if we restart all interactions by the all-for-one supervision. In case of Sc2, we need to restart all processes. In this case, protocol-recovery induces only a small overhead 7% in comparison to all-for-one supervision.

6.4 Summary
Our protocol-recovery strategy outperforms Erlang all-for-one strategy when there are more intensive local computations; and protocols are more parallelised (i.e. they are more disconnected, hence there are less IO-dependencies from the failing node). It induces only a small overhead 7% in comparison to all-for-one supervision.

7. Related and Future Work
We summarise closely related works on recovery in Erlang and/or based on session types. See Appendix A for a wider overview on recovery techniques such as checkpoint-based approaches.

7.1 Implementations of Recovery in Erlang
A few practical works have proposed to improve the recovery mechanisms of Erlang supervision trees. Currently, to obtain the process structure of an application, one must manually inspect the source code or rely on external documentation. To resolve this issue, the work [27, 28] presents a static analysis that extracts sets of possible process structures from source code, and automatically checks the effects of a process failure in each process structure. More precisely, it checks if best practices are followed when creating a supervision tree. The authors do not prove any formal guarantees and the tool does not ensure soundness of the recovered processes. In our work, we define a protocol first for checking contracts between processes, and implement an automatic recovery strategy based on that protocol. Hence, for a given session typed program, we calculate recovery points statically.

The work [8] proposes runtime monitoring for Erlang to detect messages which do not conform to a specification. The main aim is porting their former synchronous monitoring Larva [23] for object-oriented languages to asynchronous monitoring for Erlang. They do not aim to study efficiency of recovery or ensure safety properties such as deadlock-freedom as studied in this paper.

Recently the work [11] formalised non-blocking interrupts based on multiparty session types and integrated this construct into Scribble framework and runtime monitoring in Python. Unlike our work, a programmer needs to write an explicit syntax for interrupts which follows specific exception handling procedure, see § 7.2.

The work [13] presents a design and an implementation of a runtime verification for Erlang where interactions are checked against Scribble specifications. The framework allows a session to continue when a failed role is not involved in the remainder of the session. This class of failures is subsumed by one of our recovery cases, Scenario 3 in § 1. The work [15] assumes only unrecoverable failures and does not reason about consistency guarantees when processes are restarted. They combine the error handling and suspensions [10] in order to localise failures. Their work is implementation only; neither formalisation nor its correctness was given.

7.2 Session Types on Adaptations and Exception Handling
We list some related works on session-type based adaptations or exceptions. Their main focus is modelling application-level constructs for these facilities, not an error recovery (fail-fast) framework studied in this paper. All of the following works are limited to formal modelling and have not been implemented.

There are several works which use session types to reason about dynamic reconfiguration of processes. The work [3] investigates the integration of constructs for runtime adaptations in a session type discipline and presents a session type framework for adaptable processes. The processes can be suspended, restarted, upgraded
or discarded at runtime. They extend a session π-calculus with primitives for updates. The authors prove session consistency (the update does not impact session behaviour) and give an encoding of two Erlang supervision strategies, one-for-one strategy and one-for-all strategy. Our approach proposes a new method for recovery with dynamic checking and repairing active sessions, and does not aim to model existing supervision strategies.

The work [9] proposes a formal model based on multiparty session types for data-driven reconfiguration for monitored processes, where adaptations are parameterised by a set of killed roles. After reconfiguration, new monitors are created by deleting all traces of killed roles. They do not restore processes and the semantics is synchronous, hence it is not directly applicable to the Erlang setting.

The work [1] investigates a use of session types for an analysis of deadlock freedom and typable communication in the presence of dynamically changing code. The session calculus is enriched with annotations of a code block that can be updated. The arrival of an update is treated as an event external to the program. They proved processes are safe and live when processes with empty queues are updated and typed by a set of local types projected from some global type. The approach uses static dependency analysis of global types for performing recovery in Erlang without any assumptions on queue conditions and our algorithm can recover processes from the middle of an existing session.

Several works [4, 5, 7, 20] studied exception handling constructs for session types. The work [5] proposes interactional exceptions for binary sessions where the try-catch blocks are built upon session-connections for a single session. This work was extended to [4] for multiparty session types. The work [20] introduces a process for a failure of communications where the recovery process is included as a part of the syntax of processes with explicit locations. The work [7] proposes a calculus with explicit exception blocks with handlers. The user needs to explicitly write exception handling constructs in each message interaction in a protocol. Their model requires synchronisation for each occurrence in an exception block, independently whether an error has been raised or not. In the case of a recursion, synchronisation is required at every unfolding. All of the above works [4, 5, 7, 20] and [11] in § 7.1 constrain specific exception handling procedures, hence a programmer needs to write an explicit syntax for handling errors. Our work offers fail-fast (error recovery) programming framework, opposed to defensive (error prevention) provided by exception handling.

8. Conclusion

In this work, we propose an algorithm to analyse and extract causal dependences from a given multiparty session protocol, and use it to ensure that communicating processes are recovered from consistent states in the presence of a failure. A recovered system is proved to be free from deadlocks, orphan messages and reception errors. The messages to be re-sent and the set of processes to be notified are computed statically based on the dependences in the process structures. To our best knowledge, this is the first work to apply session types for defining a sound recovery strategy. Our approach can automatically generate sound supervision structures in Erlang from types, and we implement the recovery strategy on top of runtime monitoring. Our programming model uses session type-based abstractions to increase robustness of the system which is often undermined when the dependencies between processes are manually specified. We apply our recovery strategy to common concurrency patterns and a real world use case to guarantee safe executions and to reducing the recovery overhead.

Erlang philosophy is “Fail fast and recover quickly”. We believe that our work would offer an important step towards a new philosophy: “Fail fast, recover quickly and safely”.

References


A. Appendix: Additional Related Work
This appendix lists additional related work in checkpoint and recovery algorithms.

A.1 Checkpoint and Recovery Algorithms
Our recovery method which calculates a recovery point is related to checkpoint techniques studied in software recovery for message-passing systems [31] and for web-service choreographies [34, 35]. The survey [31] categorizes their checkpointing protocols as uncoordinated, coordinated, and communication-induced. Our approach belongs to the third class, where the coordination is done in a lazy fashion by piggy-backing control information on application messages where usually a local checkpoint is on the fly associated to a consistent global checkpoint. Both works [31, 34] study how to reach a consistent global state. A global checkpoint consists of a set of local checkpoints, one for each process, from which a distributed computation can be restarted after a failure. In [31], recovery relies on the knowledge and skillset of the user; and in [34], the faulty service has to restart from the beginning. Our approach does not require these assumptions that rely on the user and the restart point is not required to be always from the beginning.

The work [35] classifies different interaction patterns and defines a checkpoint based on the executed pattern. The supported patterns are nesting, sequencing, concurrent and iterative. The checkpoints are generated using rules (heuristic) defined on the pattern structures. For example, for each request-reply a checkpoint is generated in the initiator after receiving the reply. Their tool takes a choreography of web services (given as a UML diagram) as an input and generates checkpointing locations in the given choreography (the initial UML diagram augmented with checkpoint locations) as outputs. Although such an approach may reduce the recovery time, it requires additional assumptions and restrictions at the code level. The targets of the applications studied in [31, 34, 35] are clearly different from ours.

As a different approach, in [33], binary session types are used at runtime to calculate the lost messages that can happen in an asynchronous delegation. When the connection is moved, the messages that may be lost could be regarded as a failure, and the session types from both sides provide checkpoints of recovery and are used to work out where the last synchronised state is, thus the relevant messages can be re-sent to get back to the correct state. This approach only concerns about delegations between two parties and is not applicable to recover multiple processes at once in the situation where messages are propagated following a global protocol and stored in mailboxes some of which are emptied at the failure. [32] presents an algorithm to identify checkpoints of a system of interacting actors. The authors define a consistency condition in a configuration of transactors (which are actors with rollback primitives). A rollback recovers a transaction to its checkpoint state (if any). Well-behaved system obeys soundness and liveness, but allow message losses. The work is theoretical and not implemented.

B. Appendix: Scribble examples
We first list the correspondence between Scribble constructs and their multiparty session type (MPST) counterparts in Table 3.

An interaction label[payload] from p1 to p2;G specifies that a message label should be sent from role p1 to role p2 and that the protocol should then continue as prescribed by the continuation G. The choice specifies a branching where role p chooses to engage in the interactions prescribed by one of the options Gj. Recursion rec Loop G defines a scope with a protocol name Loop and body G. Any call continue Loop occurring inside G executes another recursion instance (if continue Loop is not in an appropriate scope than it remains idle). The parallel construct expresses parallel executions of interactions in G.

Figure 11 displays the Scribble protocol for the Trading Negotiation example in Fig. 1. A protocol starts with a protocol declaration followed by a protocol name and names of the participating processes, which are also called roles (Line 2-3). Then the communication between the roles is described as a sequence of message interactions of the form label[payload] from role to role. For example, Line 5 displays sending a message of type quote with payload int from process A to process C. The causality of a Scribble protocol follow those of the counterpart global type.

C. Appendix: Proofs of Main Theorems
This section proves the termination of the algorithm and the main theorems (Theorem 1 and Theorem 2) in the paper.

We first give examples of recovery points.

Example 6 (Recovery Points).
1. Branching. Consider:
   \[ A \rightarrow B: \{m_1.B \rightarrow \emptyset; m_2.C \rightarrow A : m_5, m_3.B \rightarrow C : m_4,C \rightarrow A : m_6\} \]
   We consider the case when C fails after receiving the message m_1 from B. Then C should be recovered from the beginning of the protocol. The first occurrence of C in the protocol is a choice. Therefore, C is recovered from B?\{m_2.A|m_5, m_4.A|m_6\}.

2. Branching with recursion. Consider:
   \[ \mu t.A \rightarrow B: \{m_1.B \rightarrow \emptyset; m_2.t.m_3.B \rightarrow C : m_4\} \]
   We consider the same failure, i.e C fails after receiving the message m_1 from B. The first occurrence of C in the protocol is a recursion. Therefore, C is recovered from \(\mu t.B?\{m_2.t.m_4\}\).

Lemma 1 (Generation). Given G and failed node n, and failed participant p, we generate the set inductively as in Fig. 12. When there is no more backward IO dependency (3n+2) or backward or forward dependent chain (after 3n+1 or 3n+3 steps), it reaches the maximum index n. Then \(N_0 \cup (\bigcup_{0 \leq i \leq n} N_{2+3i})\) and a recovery set \(N\) coincide.

Proof. By definition. \(\square\)

By the above lemma, Proposition 1 is immediate.

Auxiliary Definitions. We use the following definitions.

Let \(s_0\) an initial configuration of \(G\). A configuration \(s\) is reachable if \(s_0 \rightarrow^* s\) and we define the reachable set of \(S\) as \(RS(S) =\)
Table 3: The correspondence between Scribble and multiparty session types

<table>
<thead>
<tr>
<th>Multiparty Session Types</th>
<th>Scribble Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p \to p' : {a_i } \subseteq I )</td>
<td>label{payload} from ( p ) to ( p' ); ( G ) singleton branching type</td>
</tr>
<tr>
<td>( p \to p' : {a_i } \subseteq I )</td>
<td>choice at ( p ) ( {G_i} ) or ( {G_2} \ldots ) or ( {G_J} ) branching type</td>
</tr>
<tr>
<td>( \mu t.G )</td>
<td>rec Loop ( G ) recursion type</td>
</tr>
<tr>
<td>( t )</td>
<td>continue Loop ( G ) recursion variable</td>
</tr>
</tbody>
</table>

Lemma 4 (Subglobal types). Suppose \( G \) is well-formed and closed and \( p \to q \in G \). Then there exists \( G_1 \) such that \( G \simeq G_1 \geq G_2 = p \to q : \{a_i\}_{i \in I} \) and \( G_2 \) connected.

Proof. By induction of well-formed \( G \).

Lemma 5 (Recruitment points). Assume \( \mathcal{P}_0 = \{ q \mid f_G(T_p, q) = T_q \) and \( q \in \text{roles}(G) \). Then there exists \( G \simeq G_1 \geq G_2 \) such that \( q = T_q = \mathcal{P}_0 \). By Lemma 4, there exists \( G \simeq G_1 \geq G_2 \) such that \( G_2 \) is connected. Take one of the nodes in which appears in the maximum occurrence \( n \) in \( G \) (determined by Lemma 1). Take \( G/n \). Then by definition, \( f_G(T_p, q) = T_q \) for all \( q \in \mathcal{P}_0 \). Since all nodes in \( \mathcal{N} \) are causally related, we can find some connected \( G' \) such that \( G' \simeq G_1 \geq G/n \) and \( f_G(T_p, q) = T_q \) for all \( q \in \mathcal{P}_0 \).

Lemma 6 (Queues). Suppose \( (\vec{T'}, \vec{w}) \succeq (\vec{T'}, \vec{w})' \) with \( T_p \succeq T_q \) and \( f_G(T_p, q) = \emptyset \). Then \( w_{\emptyset} = w_{\emptyset} = w_{\emptyset} = \emptyset \), i.e., if the participant \( q \) is not affected by a recovery of \( p \), then the queue from \( q \) to \( p \) are empty before the failure of \( T_p \).

Proof. If \( f_G(T_p, q) = \emptyset \), then \( q \) is not affected by the recovery point or it does not start the interaction with \( p \) (hence the message from \( q \) to \( p \) or to \( q \) is not sent yet). Hence \( w_{\emptyset} = w_{\emptyset} = \emptyset \).

Lemma 7 (Permutation). Suppose \( (\vec{T'}, \vec{w}) \succeq (\vec{T'}, \vec{w}) \) with \( T_p \succeq T_q \) and \( f_G(T_p, q) = \emptyset \). Then there exists \( (\vec{T'}, \vec{w})' \) such that \( (\vec{T'}, \vec{w})' \succeq (\vec{T'}, \vec{w}) \) and \( f_G(T_p, q) = \emptyset \) (by Lemma 6). Hence we can set \( T'_q = T_q \) and \( T'_r = T_r \) and for all \( r \), \( w_{\emptyset} = w_{\emptyset} \) (by definition) and \( w_{\emptyset} = w_{\emptyset} \) (by Lemma 6). Hence we can set \( T'_q = T_q \) and \( T'_r = T_r \) and for all \( r \), \( w_{\emptyset} = w_{\emptyset} \) and \( w_{\emptyset} = w_{\emptyset} \). Hence done.
A sequence of transitions $s_1 \xrightarrow{\ell_1} s_2 \cdots \xrightarrow{\ell_m} s_{m+1}$ is said to be \textit{stable} if no channel of any intermediate configuration $s_i$ contains more than 1 message. More precisely, in each $w_{pq}$ in $s_i$, $|w_{pq}| \leq 1$.

The following is proved in [12, 13, 22].

\textbf{Proposition 2} (Stability). Suppose that $s_0$ is an initial configuration of $G$ and $s_0 \rightarrow_s \ell \rightarrow_s'$. Then there exists stable $s_0 \rightarrow_s' \ell \rightarrow_s''$ such that $s_0 \rightarrow_s''$.

\textbf{Lemma 8} (Reachability). Suppose $s_0$ is an initial configuration of $G$ and $G' \subseteq G$ and $G'$ is closed. Then there is stable $s_0 \rightarrow_s' \ell \rightarrow_s''$ such that $\nexists \ell \rightarrow_s''$.

\textbf{Lemma 9} (Recovery). Suppose $s_0$ is the initial configuration of $G$. Assume $s_0 \rightarrow_s \ell \rightarrow_s''$ and $\nexists \ell \rightarrow_s''$. Then there exists $s_0 \rightarrow_s' \ell \rightarrow_s''$ such that $s_0 \rightarrow_s''$.

\textbf{Proof}. By Lemma 4 and Proposition 2.

\textbf{Case (1)} Assume $\nexists \ell \rightarrow_s'$.\n
\textbf{Case (1-1)} Assume $s_0 \rightarrow_s' \ell \rightarrow_s''$. Hence by Proposition 2, we conclude.

\textbf{Case (1-2)} Assume $s_0 \rightarrow_s' \ell \rightarrow_s''$ by Definition 4(2-a). Let $s = (T; \bar{w})$ with $\bar{T} = T^1; T^2$ and $\bar{w} = w^1; w^2$ where $T^1$ contains the local types which will be recovered by the failure of $p$. Hence we can set $s' = (T^1; \bar{w}')$ such that $T^1 = T^3; T^2$ and $\bar{w}' = w^3; w^2$ where $w_{pq} = \emptyset$ in $\bar{w}^3$ such that $q$ in $T^1$ and $T^3$. By applying Lemma 7 repeatedly, all transitions from/to $T^2$ can be permuted. Hence by Lemma 2, we have $s_0 \rightarrow_s s_3 = (T^3; T^4; w^3; w^4)$ where $\bar{w}^4 = \bar{e}$. Now applying Lemma 5 to $p \in \text{roles}(\bar{T}^3)$ and set $\bar{T}^4$ appropriately, we have done.

\textbf{Case (2)} Assume $\nexists$ occurs $n + 1$ times in $\ell$. Then straightforward by the induction of $n$.\hfill \square

Then Theorem 1 is by Lemma 8 and Lemma 9; and Theorem 2 is by Theorem 1 and Proposition 2.

\textbf{Appendix References}


