Let It Recover: Multiparty Protocol-Induced Recovery

Rumyana Neykova
Imperial College London, UK
rumyana.neykova10@imperial.ac.uk

Nobuko Yoshida
Imperial College London, UK
yoshida@imperial.ac.uk

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 reception 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.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed Programming; D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.2 [*Semantics of Programming Languages*]: Program analysis

Keywords Multiparty Session Types, Recovery, Distributed Programming, Erlang, Fault-tolerance

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 [30]. 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 [19].

Multiparty Session Types (MPSTs) [19] 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 used to type-check each process implementation. The framework (1) has been applied to several mainstream languages (e.g, Java [20, 23], Python [11, 26], MPI/C [29], Go [28]) 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.

CC'17, February 5–6, 2017, Austin, TX, USA
ACM. 978-1-4503-5233-8/17/02...$15.00
http://dx.doi.org/10.1145/3033019.3033031

$$G = \begin{pmatrix} \mathtt{A}_1 \to \mathtt{A}_2; \\ \dagger^1 \mathtt{A}_2 \to \underline{\mathtt{A}_3}; \\ \cdots \\ \mathtt{A}_n \to \mathtt{C} \end{pmatrix} ; \begin{pmatrix} \mathtt{B}_1 \to \mathtt{B}_2; \\ \mathtt{B}_2 \to \mathtt{B}_3; \\ \cdots \\ \mathtt{B}_n \to \mathtt{D} \end{pmatrix} ; \dagger^2 \mathtt{C} \to \underline{\mathtt{E}} \; ; \; \mathtt{D} \to \mathtt{E}; \; \mathtt{E} \to \mathtt{C} : \begin{Bmatrix} \mathsf{accept}.\mathtt{E} \to \mathtt{D} : \mathsf{reject}.\dagger^3 \underline{\mathsf{end}}, \\ \mathsf{reject}.\ \mathtt{E} \to \mathtt{D} : \mathsf{accept}.\mathsf{end} \end{Bmatrix}$$

Figure 1: Trading Negotiation Global Protocol

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 $\mathtt{A} \to \mathtt{B} : \mathsf{m}$ 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 $\mathtt{A} \to \mathtt{B} : \{\mathsf{m}_1.G_1,\ \mathsf{m}_2.G_2\}$ represents a choice at a participant A to send to B either a message m_1, or a message m_2; then depending on the chosen action the protocol continues as G_1 or G_2, respectively. Message actions (send and receive) are ordered on the sender and receiver side. For example, $\mathtt{C} \to \mathtt{E}; \mathtt{D} \to \mathtt{E}$ describes messages to be processed by E in the specified order.

The trading negotiation process is split into three phases: In the first phase, two groups of participants, the team of Alice ($\mathtt{A}_i$) and the team of Bob ($\mathtt{B}_i$) forward messages to their group leaders, C and D respectively, with their suggested trading quote. The communication between the groups is independent (represented by two blocks of composed global protocols). In the second phase, the leaders of the groups notify the trader E regarding their proposals (represented by $\mathtt{C} \to \mathtt{E}; \mathtt{D} \to \mathtt{E}$). Finally, E chooses the best quote and sends accept or reject to each group leader (represented by a choice).

In the above protocol the dependencies between the processes are dynamic and change with the execution of the protocol. We consider, as examples, three possible failures and give the set of affected participants in each case using our MPST-induced approach. Table 1 summarises the result and compares it with two Erlang recovery strategies. In Fig. 1 we use the notation $\dagger^i \mathtt{A} \to \underline{\mathtt{B}}$ to denote that B fails after receiving a message from A, and $\dagger^i \underline{\mathtt{A}} \to \mathtt{B}$ to denote that A fails after sending a message to B, $\dagger^i$ corresponds to the scenario number as given below.

Scenario $\dagger^1$: $\mathtt{A}_3$ fails before sending the message to $\mathtt{A}_4$. $\mathtt{A}_3$ affects only its predecessors. Thus, if $\mathtt{A}_3$ fails, only $\mathtt{A}_1$, $\mathtt{A}_2$ and $\mathtt{A}_3$ are restarted. More generally, if $\mathtt{A}_i$ fails before sending to $\mathtt{A}_{i+1}$, then $\mathtt{A}_1$,...,$\mathtt{A}_i$ are restarted.

Scenario $\dagger^2$: E fails after receiving a message from C. All participants are restarted. Note that since the network is asynchronous C might have already sent the message to E, although E might not have selected the message from the queue. When E fails, the queue will be erased and the messages will be lost. Thus, inevitably C must be informed about E's failure.

Scenario $\dagger^3$: E fails after its last protocol interaction. Since E is not involved in the protocol any more, no other roles should be notified or recovered.

Erlang recovery strategies In Erlang, there are three types of supervisions. In *one-for-one* supervision, if a process fails, only this process is restarted by the supervisor; in *all-for-one* supervision, if any process dies, all the processes are restarted; and in *rest-for-one* supervision, if a process terminates, only the rest of the processes (the ones on the left of the terminated process in the supervision tree) are terminated.

Scenario	Fig. 2 (a)	Fig. 2 (b)	Our approach
1	$\mathtt{A}_1 \ldots \mathtt{A}_3$	$\mathtt{A}_1 \ldots \mathtt{A}_3$	$\mathtt{A}_1 \ldots \mathtt{A}_3$
2	all	only E (unsound)	all
3	all (inefficient)	only E (unsound)	nothing

Table 1: Comparison between Erlang and MPST-based recovery

In Fig. 2 we present two representative supervision hierarchies with combined strategies and compare them with our approach in Table 1. The *rest-for-one* supervision is suitable for disjoint groups of chained interactions. Hence we connect Alice's group $\mathtt{A}_i$ by a *rest-for-one* supervisor and Bob's group $\mathtt{B}_i$ by another supervisor of the same type. Then a worker E is grouped with the rest by an *all-for-one* supervisor (Fig. 2(a)) or *one-for-one* supervisor (Fig. 2(b)). The recovery driven by (a) or (b) is, as explained below, either inefficient or unsound.

Suppose E dies as assumed in Scenario 2. If we chose the supervision in Fig. 2(b), only E is restarted from the beginning. However, since C has already sent the message to E, the message by 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 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 message 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). Omitted definitions and proofs can be found in [27], the implementation and additional examples are availabble from [1].

2. Overview of 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.

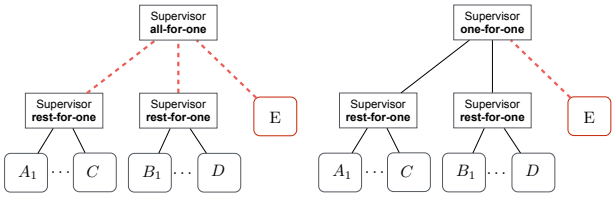


Figure 2: Erlang supervision: (a) *all-for-one* and (b) *one-for-one*

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 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):

$$\mathtt{A} \to \mathtt{B};\ \mathtt{B} \to \underline{\mathtt{C}}; \tag{2.1}$$

If C fails to receive the message from B, then A should also resend the message to B. This is because B's mail box 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:

$$\mathtt{A} \to \underline{\mathtt{B}};\ \mathtt{C} \to \mathtt{B}; \quad \text{and} \quad \underline{\mathtt{A}} \to \mathtt{B};\ \mathtt{C} \to \mathtt{B}; \tag{2.2}$$

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 still not 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.

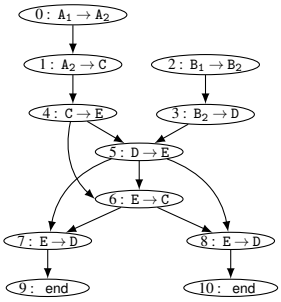
Now consider the following protocol which has additional intermediate communications to (2.2):

$$\mathtt{A} \to \underline{\mathtt{B}};\mathtt{B} \to \mathtt{D};\mathtt{D} \to \mathtt{C};\mathtt{C} \to \mathtt{B}; \tag{2.3}$$

There is an input-output chain from the failed node at B (underlined) to C. The output from C ($\mathtt{C} \to \mathtt{B}$) depends on an output from B ($\mathtt{B} \to \mathtt{D}$), 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



n	r	recovery points
0	A_1	$\{A_1:0\}$
0	A_2	$\{A_2:0, A_1:0\}$
1	A_2	$\{A_2:0, A_1:0\}$
1	C	$\{C:1, A_2:0, A_1:0\}$
3	B_2	$\{B_2:3, B_1:2\}$
3	D	$\{D:3, B_2:2, B_1:2\}$
4	C	$\{C:1, A_1:0, A_2:0\}$
4	E	$\{C:1, A_1:0, A_2:0, D:3, B_2:2, B_1:2, E:4\}$
5	D	$\{D:3, B_2:2, B_1:2\}$
5	E	$\{C:1, A_1:0, A_2:0, D:3, B_2:2, B_1:2, E:4\}$
6	C,E	$\{C:1, A_1:0, A_2:0, D:3, B_2:2, B_1:2, E:4\}$
7	D, E	$\{C:1, A_1:0, A_2:0, D:3, B_2:2, B_1:2, E:4\}$
8	E,D	$\{C:1, A_1:0, A_2:0, D:3, B_2:2, B_1:2, E:4\}$
9	any	$\varnothing$
10	any	$\varnothing$

Figure 3: Dependency graph (a) and Global Recovery Table (b) for the Trading Negotiation example

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; and if C has not sent the message to B yet, then C 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. $\mathbb{A}$ $(a, b, c, ...)$ denotes a finite alphabet and $\mathscr{P}$ is a set of *participants* fixed throughout the paper: $\mathscr{P} \subseteq \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \ldots, \mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots, \mathtt{p}, \mathtt{q}, \ldots\}$. The syntax of types is given

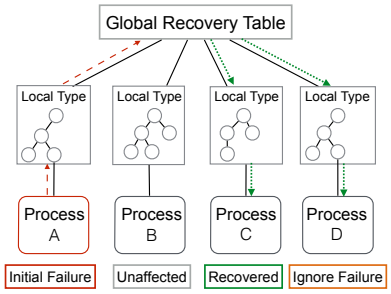


Figure 4: Supervisor's actions after the failure

as:

$$G \quad ::= \quad \mathtt{p} \to \mathtt{p}' \colon \{a_j.G_j\}_{j\in J} \mid \mu \mathtt{t}.G \mid \mathtt{t} \mid \mathsf{end}$$
$$T \quad ::= \quad \mathtt{p}?\{a_i.T_i\}_{i\in I} \mid \mathtt{p}!\{a_i.T_i\}_{i\in I} \mid \mu \mathtt{t}.T \mid \mathtt{t} \mid \mathsf{end}$$

$a_j \in \mathbb{A}$ 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 $\mathtt{p} \to \mathtt{p}' \colon \{a_j.G_j\}_{j\in J}$ states that participant $\mathtt{p}$ can send a message with one of the a_i labels to participant $\mathtt{p}'$ and that interaction described in G_j follows. We require $\mathtt{p} \neq \mathtt{p}'$ to prevent self-sent messages and $a_i \neq a_k$ for all $i \neq k \in J$. Recursive types $\mu \mathtt{t}.G$ are for recursive protocols, assuming that type variables $(\mathtt{t}, \mathtt{t}', \ldots)$ are guarded in the standard way, i.e. they only occur under branchings. Type end represents session termination (often omitted). The function $\mathsf{roles}(G)$ gives the participants of G. Concerning local types, the *branching type* $\mathtt{p}?\{a_i.T_i\}_{i\in I}$ specifies the reception of a message from $\mathtt{p}$ with a label among the a_i. The *selection type* $\mathtt{p}!\{a_i.T_i\}_{i\in I}$ is its dual. The remaining type constructors are as for global types. When branching is a singleton, we write $\mathtt{p} \to \mathtt{p}' : a; G'$ for global, and $\mathtt{p}!a.T$ or $\mathtt{p}?a.T$ for local.

Projection The relation between global and local types is formalised by projection [13, 19]. 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* $\mathtt{p}$ (written $G \upharpoonright \mathtt{p}$) is defined as:

$$\mathtt{p} \to \mathtt{p}' \colon \{a_j.G_j\}_{j\in J} \upharpoonright \mathtt{q} = \begin{cases} \mathtt{p}'!\{a_j.G_j \upharpoonright \mathtt{q}\}_{j\in J} & \mathtt{q} = \mathtt{p} \\ \mathtt{p}?\{a_j.G_j \upharpoonright \mathtt{q}\}_{j\in J} & \mathtt{q} = \mathtt{p}' \\ \sqcup_{j\in J} G_j \upharpoonright \mathtt{q} & \text{otherwise} \end{cases}$$

$$(\mu \mathtt{t}.G) \upharpoonright \mathtt{p} = \begin{cases} \mu \mathtt{t}.G \upharpoonright \mathtt{p} & G \upharpoonright \mathtt{p} \neq \mathtt{t} \\ \mathsf{end} & \text{otherwise} \end{cases} \qquad \mathtt{t} \upharpoonright \mathtt{p} = \mathtt{t} \qquad \mathsf{end} \upharpoonright \mathtt{p} = \mathsf{end}$$

where $\mathsf{fv}(G)$ 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;
- $\mathtt{p}?\{a_k.T_k\}_{k\in K} \sqcup \mathtt{p}?\{a_j.T'_j\}_{j\in J}$
  $= \mathtt{p}?(\{a_k.(T_k \sqcup T'_k)\}_{k\in K\cap J} \cup \{a_k.T_k\}_{k\in K\setminus J} \cup \{a_j.T'_j\}_{j\in J\setminus K})$

and homomorphic for other types (i.e. $\mathscr{C}[T_1] \sqcup \mathscr{C}[T_2] = \mathscr{C}[T_1 \sqcup T_2]$ where $\mathscr{C}$ 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 $\mathtt{p} \in \mathscr{P}$, $G \upharpoonright \mathtt{p}$ is defined.

Consider the global type given below.

$$\mu \mathtt{t}.\mathtt{A} \to \mathtt{B} \colon \{a.\mathtt{B} \to \mathtt{C} \colon b.\mathtt{t}\ ,\ d.\mathtt{B} \to \mathtt{C} \colon e.\mathsf{end}\}$$

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

- A's local type: $\mu \mathtt{t}.\mathtt{B}!\{a.\mathtt{t},\ d.\mathsf{end}\}$.
- B's local type is: $\mu \mathtt{t}.\mathtt{A}?\{a.\mathtt{C}!b.\mathtt{t},\ d.\mathtt{C}!e.\mathsf{end}\}$.
- C's local type is: $\mu \mathtt{t}.\mathtt{B}?\{b.\mathtt{t},\ e.\mathsf{end}\}$.

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

$G_1 = \mathtt{B} \to \mathtt{C};\ \mathtt{A} \to \mathtt{C};\ \mathtt{B} \to \underline{\mathtt{A}};\ \mathtt{C} \to \mathtt{A};\ \mathtt{A} \to \mathtt{D};\ \mathtt{D} \to \mathtt{C};$

A : C!. B?.C?.D!.end
B : C!. A!.end
C : B?.A?.A!. D?.end
D : A?.C!.end

$G_2 = \mathtt{B} \to \mathtt{C};\ \mathtt{A} \to \mathtt{C};\ \mathtt{B} \to \underline{\mathtt{A}};\ \mathtt{A} \to \mathtt{D};\ \mathtt{D} \to \mathtt{C};$

A : C!.B?.D!.end
B : C!.A!.end
C : B?.A?. D?.end
D : A?.C!.end

$G_3 = \mathtt{A} \to \underline{\mathtt{B}};\ \mathtt{C} \to \mathtt{B} : l_1;\ \mathtt{C} \to \mathtt{B} : l_2;$

A : B!.end
B : A?.C?l_1.C?l_2.end
C : B!l_1. B!l_2.end

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

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 [31] 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).

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 their initial state (orphan message error) In G_2 we assume the same failure as in G_1, 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 ($\mathtt{B} \to \mathtt{C}$ and $\mathtt{A} \to \mathtt{C}$). 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_3 and B fails at $\mathtt{A} \to \mathtt{B}$. 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 C 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 ($\prec_{\mathtt{IO}}$ and $\lhd$) 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

$\mathtt{p} \to \mathtt{p}' : \{a_j.G_j\}_{j\in J}$ with a node name (denoted by $\mathtt{n}, \mathtt{n}', \mathtt{n}_1, \mathtt{n}_2, \ldots$) Then, we inductively define a function $\mathtt{node}_G$ that gives a set of nodes (or the special node end) for each of the syntactic subterm of G as follows:

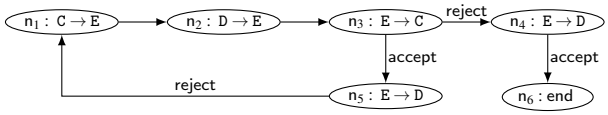
- $\mathtt{node}_G(\mu \mathtt{t}.G') = \mathtt{node}_G(G')$; and $\mathtt{node}_G(\mathsf{end}) = \mathsf{end}$
- $\mathtt{node}_G(\mathtt{n} : \mathtt{p} \to \mathtt{p}' : \{k_j.G_j\}_{j\in J}) = \mathtt{n}$
- $\mathtt{node}_G(\mathtt{t}) = \mathtt{node}_G(\mu \mathtt{t}.G')$ if $\mu \mathtt{t}.G' \in G$ and $\mathtt{t} \in \mathsf{fv}(G')$

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

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

$$\mu \mathtt{t}.\mathtt{C} \to \mathtt{E}; \mathtt{D} \to \mathtt{E}; \mathtt{E} \to \mathtt{C} : \left\{ \begin{array}{l} \mathtt{accept}.\mathtt{E} \to \mathtt{D} : \mathtt{reject}.\mathtt{t}, \\ \mathtt{reject}.\mathtt{E} \to \mathtt{D} : \mathtt{accept}.\mathsf{end} \end{array} \right\}$$

Then its graph representation with the initial node $\mathtt{n}_1$ is given as:



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

Causality and causality chains We define causality relations in a given G by the relations $\lhd$ and $\prec_{\mathtt{IO}}$. The dependency relation $\lhd$ represents the order between two nodes which has a common participant; and the IO-relation $\prec_{\mathtt{IO}}$ asserts the order between a reception of a message and a send action. Formally,

$$\begin{array}{ll} \mathtt{n}_1 \lhd \mathtt{n}_2 & \text{if } \mathtt{n}_1 \preccurlyeq \mathtt{n}_2 \text{ and } \mathsf{roles}(\mathtt{n}_1) \cap \mathsf{roles}(\mathtt{n}_2) \neq \emptyset \\ \mathtt{n}_1 \prec_{\mathtt{IO}} \mathtt{n}_2 & \text{if } \mathtt{n}_1 \preccurlyeq \mathtt{n}_2 \text{ and } \mathtt{pfx}(\mathtt{n}_1) = \mathtt{p}_1 \to \mathtt{p} : a_1 \\ & \qquad \text{and } \mathtt{pfx}(\mathtt{n}_2) = \mathtt{p} \to \mathtt{p}_2 : a_2 \end{array}$$

An *input-output dependency (IO-dependency)* from $\mathtt{n}_1$ to $\mathtt{n}_n$ (denoted by $\mathtt{n}_1 \preccurlyeq_{\mathtt{IO}} \mathtt{n}_n$) is a chain $\mathtt{n}_1 \prec_{\mathtt{IO}} \cdots \prec_{\mathtt{IO}} \mathtt{n}_n$ ($n \geq 1$). In Example 2, we have $\mathtt{n}_1 \prec_{\mathtt{IO}} \mathtt{n}_3$, $\mathtt{n}_2 \prec_{\mathtt{IO}} \mathtt{n}_3$ and $\mathtt{n}_1 \lhd \mathtt{n}_2$, but $(\mathtt{n}_1, \mathtt{n}_2)$ 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* $\mathscr{N}$ when a participant $\mathtt{p}$ in a global type G fails. The algorithm is shown in Figure 6, where $\mathtt{n}_i$ is the failed node, it corresponds to the state of $\mathtt{p}$ at the time of failure. Below we explain each step of the algorithm. We write $\mathtt{n} = \mathtt{p} \to \mathtt{q}$ means $\mathtt{pfx}(\mathtt{n}) = \mathtt{p} \to \mathtt{q} : a$ for some a.

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

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

Step 2.1: Traversing input-output dependencies, Backward traversal of $\prec_{\mathtt{IO}}$ (Line 4) A set of ***backward affected nodes*** $\mathscr{N}^{\leftarrow}$ 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 unaffected nodes if they are dependent from affected nodes.

Step 2.2: Forward Traversal of $\lhd$-dependencies (Line 5) A set of ***forward affected nodes*** $\mathscr{N}^{\rightarrow}$ 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.

Below we write $G/\mathtt{n}_\mathtt{q}$ to denote a subterm (subgraph) of G whose occurrence is $\mathtt{n}_\mathtt{q}$. For example, if we take the session graph in Example 2 $G/\mathtt{n}_4 = \mathtt{E} \to \mathtt{D} : \mathtt{accept}.\mathsf{end}$ and $G/\mathtt{n}_5 = \mathtt{E} \to \mathtt{D} : \mathtt{reject}.G$, where we write the outputs using the syntax of global types.

We write $T \supseteq T'$ to denote T' is a subterm of T. For example, $\mathtt{A}?a.\mathtt{B}!b.\mathsf{end} \supseteq \mathtt{B}!b.\mathsf{end}$. We define $\max(\{T_i\}_{i\in I}) = T_j$ if for all $i \in I$, $T_j \supseteq T_i$.

Definition 3 (Recovery point)**.** Assume that $\mathscr{N}$ is the set of affected nodes when $\mathtt{n}_i$ in global type G failed and the participant $\mathtt{p} \in \mathsf{roles}(\mathtt{n}_i)$ failed. Assume $G/\mathtt{n}_i \upharpoonright \mathtt{p} = T_\mathtt{p}$. We then define, for each $\mathtt{q} \in \mathscr{P}$, (1) $f_G(T_\mathtt{p}, \mathtt{q}) = \max(\{G/\mathtt{n} \upharpoonright \mathtt{q} \mid \mathtt{n} \in \mathscr{N}, \mathtt{q} \in \mathsf{roles}(\mathtt{n})\})$; or (2) $f_G(T_\mathtt{p}, \mathtt{q}) = \emptyset$ if $\mathtt{q} \notin \mathsf{roles}(\mathtt{n})$ for all $\mathtt{n} \in \mathscr{N}$. 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.*

3.5 Examples of Protocol Recovery

We explain the necessity of the conditions given in Fig. 6 via examples, presented in Fig. 7. We underline and colour the failed role and node and list the recovery nodes for $\mathbf{G_{4,5,6,7}}$.

($\mathbf{G_1}$) We assume the failed role ($\mathtt{p}$) is $\mathtt{A}$ and the failed node ($\mathtt{n}_i$) is $\mathtt{n}_1$. $\mathtt{n}_1$ and $\mathtt{n}_2$ are added to the initial affected nodes set $\mathscr{N}$ since $\mathtt{n}_i = \mathtt{n}_1 \lhd \mathtt{n}_2 = \mathtt{r} \to \mathtt{p}$. ($\mathbf{G_2}$) At the initialisation, $\mathscr{N} = \{\mathtt{n}_1, \mathtt{n}_3\}$ since $\mathtt{n}_i = \mathtt{n}_1 \lhd \mathtt{n}_3 = \mathtt{r} \to \mathtt{p}$. Then $\mathtt{n}_2$ is added in Step 2.1 since it is backward IO-dependent from $\mathtt{n}_3$. ($\mathbf{G_3}$) This demonstrates a need for the non-affected set $\mathscr{S}$. We have $\mathscr{S} = \{\mathtt{n}_2, \mathtt{n}_3\}$ since $\mathtt{n}_1 \lhd \mathtt{n}_2 = \mathtt{p} \to \mathtt{r}$, i.e. $\mathtt{n}_2$ is an output from the failed role; and $\mathtt{n}_2 \preccurlyeq_{\mathtt{IO}} \mathtt{n}_3$ (Line 2). These nodes do not have to recover.

($\mathbf{G_4}$) We demonstrate a need of condition $\mathtt{n} \lhd \mathtt{n}' \wedge \mathtt{n} \in \mathscr{S}$ (Line 4 in Fig. 6). Initially, we set $\mathscr{S} = \{\mathtt{n}_2\}$ and $\mathscr{N} = \{\mathtt{n}_1, \mathtt{n}_3\}$. Assume that we do not consider $\mathtt{n} \lhd \mathtt{n}' \wedge \mathtt{n} \in \mathscr{S}$. Then after four iterations, we (wrongly) obtain $\mathscr{N}_{wrong} = \{\mathtt{n}_1, \mathtt{n}_3, \mathtt{n}_4\}$ as the final affected set,

Algorithm *Calculating affected nodes*
Input: $\mathtt{n}_i$ (a failed node), $\mathtt{p}$ (a failed role)
Output: $\mathscr{N}$ (a set of affected nodes)
1. $\mathscr{N} = \mathscr{N}^{\rightarrow} = \{\mathtt{n} \mid \mathtt{n}_i \lhd \mathtt{n} \ \wedge \ \mathtt{n} = \mathtt{r} \to \mathtt{p}\} \cup \{\mathtt{n}_i\}$
2. $\mathscr{S} = \{\mathtt{n} \mid ((\mathtt{n}_i \lhd \mathtt{n}' \wedge \mathtt{n}' = \mathtt{p} \to \mathtt{r}) \vee \mathtt{n}' = \mathtt{n}_i) \wedge \mathtt{n}' \preccurlyeq_{\mathtt{IO}} \mathtt{n}\} \setminus \{\mathtt{n}_i\}$
3. **repeat**
4. $\quad \mathscr{N}^{\leftarrow} = \{\mathtt{n} \mid \mathtt{n} \preccurlyeq_{\mathtt{IO}} \mathtt{n}' \ \vee \ (\mathtt{n} \lhd \mathtt{n}' \wedge \mathtt{n} \in \mathscr{S}) \ \wedge \ \mathtt{n}' \in \mathscr{N}^{\rightarrow}\}$
5. $\quad \mathscr{N}^{\rightarrow} = \{\mathtt{n} \mid \mathtt{n}' \lhd \mathtt{n} \ \wedge \ \mathtt{n}' \in \mathscr{N}^{\leftarrow}\} \setminus (\mathscr{N} \cup \mathscr{S})$
6. $\quad \mathscr{N} = \mathscr{N} \cup \mathscr{N}^{\leftarrow} \quad \mathscr{S} = \mathscr{S} \setminus \mathscr{N}^{\leftarrow}$
7. **until** $\mathscr{N}^{\leftarrow} = \mathscr{N}^{\rightarrow} = \emptyset$
8. **return** $\mathscr{N}$

Figure 6: Calculating affected nodes

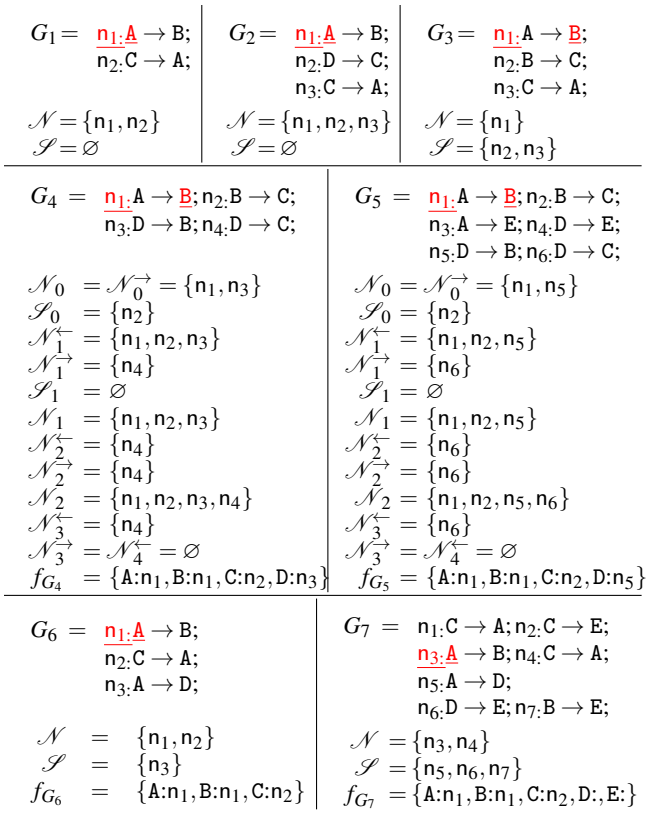


Figure 7: Examples for Recovery Algorithm and Recovery Points

i.e. n_2 is missing. Assume $\mathcal{N}_{wrong}$. Then the role C will recover from node $n_4{:}D \to C$, while B will recover from node n_1. After this recovery, B will proceed by sending a message to C, although C will expect a message from D. Our algorithm checks at every iteration step, which nodes in the non-affected set $\mathcal{S}$ should be added taking forward dependencies in Line 4 into account. Since we have $n_2 \lhd n_3$ and $n_3 \in \mathcal{N}^{\to}$, $n_2 \in \mathcal{S}$, and the final affected set should be $\mathcal{N} = \{n_1, n_2, n_3, n_4\}$, hence no such error exists. (**G_5**) We add nodes n_3 and n_4 to G_4, but the affected nodes stay unchanged (by replacing n_3 by n_5 and n_4 by n_6).

(**G_6**) At Step 2.2, $\mathcal{N}^{\leftarrow} = \{n_3\}$ since $n_1 \lhd n_2 \lhd n_3$. Because $n_3 \in \mathcal{S}$ and by Steps 2.3 and 2.4, n_3 is not added to the affected nodes set. Hence we have $\mathcal{N} = \{n_1, n_2\}$ and $\mathcal{S} = \{n_3\}$. (**G_7**) we have the same set of affected nodes with G_6. The added nodes are either in $\mathcal{S}$ or are preceding from the failed node so they are not affected. See [27] for more examples.

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 $(\ell, \ell', ...)$. An action ℓ denotes the sending or the reception of a message of label a from p to p′ and the *recovery message* †.

$$\ell ::= \mathtt{pp'}!a \mid \mathtt{pp'}?a \mid \dagger$$

We then define the LTS over local types starting from an individual local type. The relation $T \xrightarrow{\ell} T'$, for the local type of participant p, is defined as:

$$[\text{OUT}]\ \ \mathtt{q}!\{a_i.T_i\}_{i\in I} \xrightarrow{\mathtt{pq}!a_i} T_i \qquad [\text{IN}]\ \ \mathtt{q}?\{a_i.T_i\}_{i\in I} \xrightarrow{\mathtt{qp}?a_j} T_j$$

$$[\text{MU}]\ \ \frac{T[\mu \mathtt{t}.T/\mathtt{t}] \xrightarrow{\ell} T'}{\mu \mathtt{t}.T \xrightarrow{\ell} T'} \qquad [\text{REC}]\ \ \frac{f_G(T_\mathtt{p}, \mathtt{p}) = T'_\mathtt{p}}{T_\mathtt{p} \xrightarrow{\dagger} T'_\mathtt{p}}$$

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 $f_G(T_\mathtt{p}, \mathtt{p}) = T'$ which means, given global type G, the participant p recovers to T' if it fails at T.

The main function $f_G(T_\mathtt{q}, \mathtt{p})$ returns the new local type for a participant p when q fails at a state T_q. In [REC], since participant p fails at $T_\mathtt{p}$, we calculate $f_G(T_\mathtt{p}, \mathtt{p})$.

LTS over a configuration We define the LTS for a configuration which consists of a collection of local types and FIFO queues. The item (1) below defines the standard asynchronous communication rules, adapted from communicating finite state machines [3]. The participant p enqueues a value to FIFO queue $w_\mathtt{pq}$ of a channel pq, and participant q dequeues a value from $w_\mathtt{pq}$. In addition, we define the two cases when participant p fails at $T_\mathtt{p}$ (item 2 below). The item (a) is the case when participant q needs to recover as local type $T'_\mathtt{q}$. 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 $\{T_\mathtt{p}\}_{\mathtt{p}\in\mathscr{p}}$ is a pair with $\vec{T} = (T_\mathtt{p})_{\mathtt{p}\in\mathscr{p}}$ and $\vec{w} = (w_\mathtt{pq})_{\mathtt{p}\neq\mathtt{q}\in\mathscr{p}}$ with $w_\mathtt{pq} \in \mathbb{A}^*$. The *initial configuration* of G is $s = (\vec{T}; \vec{w})$ with $w_\mathtt{pq} = \varepsilon$ and $T_\mathtt{p} = G \upharpoonright \mathtt{p}$. A *final configuration* is $s = (\vec{T}; \vec{\varepsilon})$ with $T_i = \mathsf{end}$. 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 $s \xrightarrow{\ell} s' = (\vec{T}'; \vec{w}')$ are defined as:

1. $T_\mathtt{p} \xrightarrow{\mathtt{pq}!a} T'_\mathtt{p}$ and $w'_\mathtt{pq} = w_\mathtt{pq} \cdot a$ and $T'_{\mathtt{p}'} = T_{\mathtt{p}'}$ for all $\mathtt{p}' \neq \mathtt{p}$; or
   $T_\mathtt{q} \xrightarrow{\mathtt{pq}?a} T'_\mathtt{q}$ and $w_\mathtt{pq} = a \cdot w'_\mathtt{pq}$ and $T'_{\mathtt{p}'} = T_{\mathtt{p}'}$ for all $\mathtt{p}' \neq \mathtt{q}$ with $w'_{\mathtt{p}'\mathtt{q}'} = w_{\mathtt{p}'\mathtt{q}'}$ for all $\mathtt{p}'\mathtt{q}' \neq \mathtt{pq}$; or
2. $T_\mathtt{p} \xrightarrow{\dagger} T'_\mathtt{p}$ then
   (a) if $f_G(T_\mathtt{p}, \mathtt{q}) = T''_\mathtt{q}$ then $T'_\mathtt{q} = T''_\mathtt{q}$ and $w'_\mathtt{rq} = \varepsilon$ for all $\mathtt{r} \neq \mathtt{q}$; and
   (b) if $f_G(T_\mathtt{p}, \mathtt{q}) = \emptyset$ then $T'_\mathtt{q} = T_\mathtt{q}$ and $w'_\mathtt{rq} = w_\mathtt{rq}$ and $w'_\mathtt{qr} = w_\mathtt{qr}$ for all $\mathtt{r} \neq \mathtt{q}$

We denote $s \xrightarrow{\ell_1 \cdots \ell_n} s'$ (or $s \to^* s'$) for $s \xrightarrow{\ell_1} s_1 \cdots s_{n-1} \xrightarrow{\ell_n} s'$. A configuration s is *reachable* if $s_0 \to^* s$.

Example 5 (Trading Negotiation)**.** We recall Fig. 3.

(1) Participant B_2 fails at node 3. By Definition 3, $f_G(T_{B_2}, B_2) = T_{B_2}$; $f_G(T_{B_2}, B_1) = T_{B_1}$ where $T_{B_1} = G \upharpoonright B_1$ and $T_{B_2} = G \upharpoonright B_2$. Also for all $\mathtt{p} \notin \{B_1, B_2\}$, $f_G(T_{B_2}, \mathtt{p}) = \emptyset$. By Definition 4(2), $T_{B_2} \xrightarrow{\dagger} T_{B_2}$. We also set the input queues of B_1 and B_2 to be empty. Hence $w'_{B_1B_2} = w'_{B_2B_1} = \varepsilon$. By (2-a), we set $T'_{B_1} = T_{B_1}$. Except B_1 and B_2, the queues and local types are unchanged. Note that if $\mathtt{p} \neq B_1$, $w_{\mathtt{p}B_2}$ is empty before B_2 fails (since $f_G(T_{B_2}, \mathtt{p}) = \emptyset$). The cases when A_2 fails at node 1 and C fails at node 4 can be calculated similarly.

(2) Participant E fails at node 4. By the algorithm, nodes 0 and 1 are added to $\mathcal{N}$ since there is a backward IO-dependency $n_0 \prec\!\!\prec_{IO} n_1 \prec\!\!\prec_{IO} n_4$; also node 5 is added to $\mathcal{N}^{\to}$ since $n_4 \lhd n_5$. From node 5, there is a backward IO-dependency $n_2 \prec\!\!\prec_{IO} n_3 \prec\!\!\prec_{IO} n_5$, hence

nodes 2 and 3 are also added to $\mathscr{N}$. Hence for all $\mathtt{p}$, $f_G(T_{\mathtt{B}_2}, \mathtt{p}) = G \upharpoonright \mathtt{p}$. 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 [3, 6] as desired properties. These are ensured by the multiparty session type theory without failures [12, 13, 24].

Theorem 1 (Transparency)**.** *Suppose s_0 is the initial configuration of G. If $s_0 \xrightarrow{\vec{\ell}} {}^* s \xrightarrow{\dagger} s'$, then there exists $s' \xrightarrow{\vec{\ell}_1} s''$ where $s_0 \xrightarrow{\vec{\ell}_2} {}^* s''$ and $\vec{\ell}_2$ does not contain $\dagger$.*

Proof. We first prove a set of local types defined by f for a given failed type $T_{\mathtt{p}}$ and G, i.e. $\{f_G(T_{\mathtt{p}}, \mathtt{q})\}_{\mathtt{q} \in \mathscr{P}}$ forms a projection of a subgraph of G. We then show if $f_G(T_{\mathtt{p}}, \mathtt{q})$ 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. □

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 $\vec{w} = \vec{\varepsilon}$ and each $T_{\mathtt{p}}$ is a branching type, i.e. all types are blocked, waiting for messages.
2. Configuration s is an *orphan message configuration* if all $T_{\mathtt{p}} \in \vec{T}$ are end but $\vec{w} \neq \vec{\varepsilon}$, i.e., there is at least an orphan message in a buffer.
3. Configuration s is an *unspecified reception configuration* if there exists $\mathtt{q} \in \mathscr{P}$ such that $T_{\mathtt{q}}$ is a branching, and $T_{\mathtt{q}} \xrightarrow{\mathtt{pq}?a} T'_{\mathtt{q}}$ implies that $|w_{\mathtt{pq}}| > 0$ and $w_{\mathtt{pq}} \notin a\mathbb{A}^*$, i.e., $T_{\mathtt{q}}$ is prevented from receiving any message from buffer $\mathtt{pq}$, 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 massage and a reception error.*

Proof. By Theorem 1 and [24, 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 [32] 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 (`gen_protocol behaviour`): 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 how to program with `gen_protocol` in § 5.1.

5.1 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: $\mathtt{A} \to \mathtt{B} : \mathtt{quote}$, the process 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 ? in Erlang is used to annotate local constants, e.g `?E`. The parameter `method` is a label for the message 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

```
1  % Initialisation of A
2  init(Val) →
3    role:send(State#state.role, ?A1, quote, Val).
4
5  % Handlers for C and D
6   quote({msg,Val},State) →
7    role:send(State#state.role, ?E, quote, Val).
8
9   accept({msg,_},State) → {ok,State}.
10  reject({msg,_},State) → {ok,State}.
11
12 % Handlers for E
13 quote({msg, Val},State) when State.prev==undef →
14   {noreply,State#state{prev=Val}};;
15
16 quote({msg, Val},State) when State#state.prev>Val →
17   role:send(State#state.role, ?C, reject, empty),
18   role:send(State#state.role, ?D, accept, empty),
19   {noreply,State};
20
21  quote({msg, Val},State) when State#state.prev<Val →
22   role:send(State#state.role, ?C, accept, empty),
23   role:send(State#state.role, ?D, reject, empty),
24   {noreply,State}.
```

Figure 8: Message handlers for endpoint processes

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.

5.2 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 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.

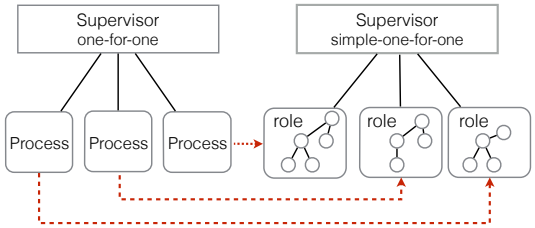


Figure 9: Supervision hierarchy

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 since it uniquely identifies the state. The projection ensures a 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 process 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 [18, 21]. 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* recovers 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 [26, § 3.2] and [14] as to express parametrised 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

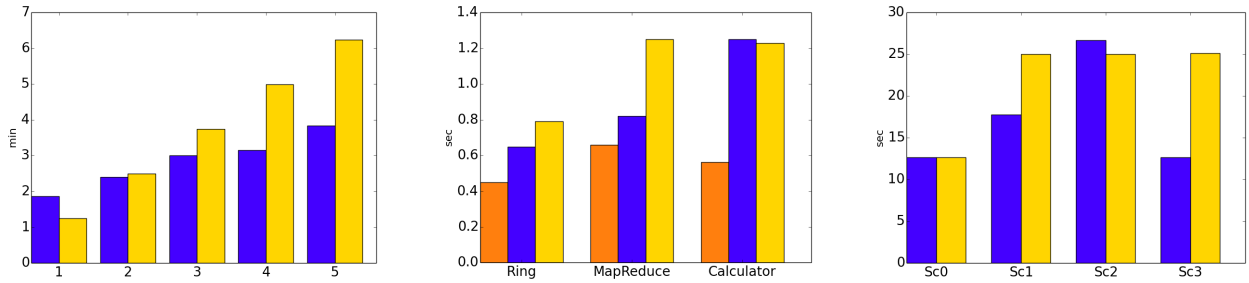


Figure 10: Execution time for (a) Web Crawler, (b) Ring, MapReduce and Calculator and (c) Trading Negotiation; the colours of the bars on the graph correspond to (*no failure* *protocol-recovery* *all-for-one*)

Example	#roles	#states	GRT (sec)	affected roles
Web Crawler [21]	`2*n+2`	`4*n`	0.45	–
MapReduce [21]	`n+1`	`n+2`	0.11	`W[1] ... W[n]`
Ring [21]	`n`	`2*n`	0.16	`W[1] ... W[n]`
Calculator [18]	`n+1`	`4*n`	0.75	`A[1]`
Trading Negotiation	`2*n+1`	`2*n+4`	0.17	in Table 1

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

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 table (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 on a specific failure. For the web crawler use case this column is not applicable since we insert failures randomly.

Setup We use the version Erlang 17.0. All processes run on the same Erlang node with Mnesia running on a separate node. The configuration for the machine is Ubuntu 13.04 64bits GNU/Linux; 8 Cores: Intel(R) Core i7-4770 CPU @ 3.40GHz 16Gb of RAM.

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 splitted 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 `Paser` 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:

$$\texttt{Downloader[i:1..n]} \rightarrow \texttt{Parser[i]:parse};$$
$$\texttt{Parser[i]} \rightarrow \texttt{Indexer:index}; \texttt{Indexer} \rightarrow \texttt{Master:url};$$

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.

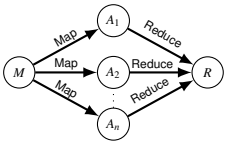
During the experimental evaluation, we tried different failure strategies, such as assigning a probabilistic failure value to each process. This approach resulted in many processes failing simultaneously. Our recovery managed to complete the tasks, while *all-for-one* could not since it quickly reached 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 `Wrorkers`. Each worker performs its sub task and notify the `Master` by sending a `reduce` message.

$$\texttt{Master} \rightarrow \texttt{Worker[i:1..n]:map};$$
$$\texttt{Worker[i]} \rightarrow \texttt{Master:reduce};$$



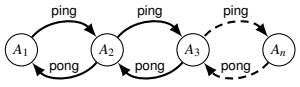
Failure: The `Master` fails after sending `map` to everyone.

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 `n` 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:

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

$\mathtt{A}[\mathtt{i}:1..\mathtt{n}-1] \to \mathtt{A}[\mathtt{i}+1]:\mathtt{ping};$
$\mathtt{A}[\mathtt{i}:\mathtt{n}..2] \to \mathtt{A}[\mathtt{i}-1]:\mathtt{pong};$

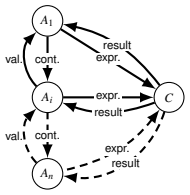


Failure: A process $\mathtt{A}[\mathtt{k}]$ fails before sending a `pong` message.

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

$\mathtt{A}[\mathtt{i}:1..\mathtt{n}-1] \to \mathtt{C}\colon \mathtt{expr};$
$\mathtt{A}[\mathtt{i}] \to \mathtt{A}[\mathtt{i}+1]\colon \mathtt{cont};$
$\mathtt{C} \to \mathtt{A}[\mathtt{i}]\colon \mathtt{result};$
$\mathtt{A}[\mathtt{n}] \to \mathtt{C}\colon \mathtt{expr};$
$\mathtt{C} \to \mathtt{A}[\mathtt{n}]\colon \mathtt{result};$
$\mathtt{A}[\mathtt{i}:\mathtt{n}..2] \to \mathtt{A}[\mathtt{i}-1]\colon \mathtt{result};$



Failure: `C` fails after processing a message form $\mathtt{A}[\mathtt{i}]$.

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 spawn 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 if 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 incurs a little overhead comparing to the case with no failure. The motivation of our work is not performance gain, but an automatic error prevention both at the process level (we ensure processes are safe and conform to a protocol); and at the level of the supervision trees and dynamic process linking (we create supervision trees and link processes dynamically based on a protocol structure). Specific performance optimisations (such as using persistent storage to cache the intermediate messages) and recovery actions (such as changing the database connection if this was the reason for the initial failure) are orthogonal/complementary concerns to our work.

7. Related and Future Work

We summarise closely related works on recovery in Erlang and/or based on session types. See [27] 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 [30, 31] 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 [25] 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 [16] 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 [16] assumes only unrecoverable failures and does not reason about consistency guarantees when processes are restarted. They combine the error handling and subsessions [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 [15] 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 district session behaviour) and give in [17] 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 [2] 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. Our 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, 22] 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 [22] 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, 22] 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 dependencies 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 dependencies 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. We apply our recovery strategy to common concurrency patterns and a real world use case to guarantee safe executions and to reduce 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"*.

Acknowledgements

We thank Ferran Obiols Jordan for his work on implementing the Scribble monitoring framework in Erlang, Raymond Hu and Dominic Orchard for fruitful discussions on this work, and anonymous referees for their comments. This work is partially supported by EPSRC projects EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1 and EP/N028201/1; by EU FP7 612985 (UPSCALE), COST Actions IC1201 (BETTY) and IC1405 (RC).

References

[1] Repository for the implementation. `http://gitlab.doc.ic.ac.uk/rn710/codeINspire`.

[2] G. Anderson and J. Rathke. Dynamic software update for message passing programs. In *APLAS*, volume 7705 of *LNCS*, pages 207–222. Springer, 2012.

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

[4] S. Capecchi, E. Giachino, and N. Yoshida. Global escape in multiparty session. In *FSTTCS'10*, volume 8 of *LIPICS*, pages 338–351, 2010.

[5] M. Carbone, K. Honda, and N. Yoshida. Structured interactional exceptions in session types. In *CONCUR*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.

[6] G. Cécé and A. Finkel. Verification of programs with half-duplex communication. *I&C*, 202(2):166–190, 2005.

[7] T. Chen, M. Viering, A. Bejleri, L. Ziarek, and P. Eugster. A type theory for robust failure handling in distributed systems. In *FORTE*, volume 9688 of *LNCS*, pages 96–113. Springer, 2016.

[8] C. Colombo, A. Francalanza, and R. Gatt. Elarva: A monitoring tool for Erlang. In *RV*, volume 7186 of *LNCS*, pages 370–374, 2011.

[9] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Parallel monitors for self-adaptive sessions. In *PLACES*, volume 211 of *EPTCS*, pages 25–36, 2016.

[10] R. Demangeon and K. Honda. Nested protocols in session types. In *CONCUR*, volume 7454 of *LNCS*, pages 272–286. Springer, 2012.

[11] R. Demangeon, K. Honda, R. Hu, R. Neykova, and N. Yoshida. Practical interruptible conversations: Distributed dynamic verification with multiparty session types and python. *FMSD*, pages 1–29, 2015.

[12] P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.

[13] P.-M. Deniélou and N. Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP*, volume 7966 of *LNCS*, pages 174–186, 2013.

[14] P.-M. Denielou, N. Yoshida, A. Bejleri, and R. Hu. Parameterised multiparty session types. *LMCS*, 8, 2012.

[15] C. Di Giusto and J. A. Pérez. Disciplined structured communications with disciplined runtime adaptation. *Sci. Comput. Program.*, 97(P2): 235–265, Jan. 2015. ISSN 0167-6423.

[16] S. Fowler. An Erlang implementation of multiparty session actors. In *ICE*, volume 223 of *EPTCS*, pages 36–50, 2016.

[17] C. D. Giusto and J. A. Pérez. Session types with runtime adaptation: Overview and examples. In *PLACES*, pages 21–32, 2013.

[18] J. He, P. Wadler, and P. Trinder. Typecasting actors: From Akka to TAkka. In *SCALA*, SCALA, pages 23–33. ACM, 2014.

[19] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008. A full version appeared in *JACM*(63)9:1–67.

[20] R. Hu and N. Yoshida. Hybrid session verification through endpoint API generation. In *FASE 2016*, volume 9633 of *LNCS*, 2016.

[21] S. Imam and V. Sarkar. Savina - An Actor Benchmark Suite. In *AGERE!*, AGERE!, 2014.

[22] D. Kouzapas, R. Gutkovas, and S. J. Gay. Session types for broadcasting. In *PLACES*, volume 155 of *EPTCS*, pages 25–31, 2014.

[23] D. Kouzapas, O. Dardha, R. Perera, and S. J. Gay. Typechecking protocols with Mungo and StMungo. In *PPDP*, pages 146–159, 2016.

[24] J. Lange, E. Tuosto, and N. Yoshida. From communicating machines to graphical choreographies. In *POPL*, pages 221–232. ACM, 2015.

[25] LARVA project. `http://www.cs.um.edu.mt/svrg/Tools/LARVA/`.

[26] R. Neykova and N. Yoshida. Multiparty Session Actors. In *COORDINATION*, volume 8459 of *LNCS*, 2014. A full version in *LMCS*.

[27] R. Neykova and N. Yoshida. Let it recover: Multiparty protocol-induced recovery. Technical Report DTRS16-10, Department of Computer Science, Imperial College London, 2016.

[28] N. Ng and N. Yoshida. Static deadlock detection for concurrent Go by global session graph synthesis. In *CC*, pages 174–184. ACM, 2016.

[29] N. Ng, J. G. Coutinho, and N. Yoshida. Protocols by default: Safe mpi code generation based on session types. In *CC*, volume 9031 of *LNCS*, pages 212–232. Springer, 2015.

[30] J. H. Nyström. Automatic assessment of failure recovery in Erlang applications. In *ERLANG*, ERLANG, pages 23–32. ACM, 2009.

[31] J. H. Nyström. *Analysing Fault Tolerance for ERLANG Applications*. PhD thesis, ACTA UNIVERSITATIS UPSALIENSI, 2009.

[32] Scribble project home page. `http://www.scribble.org`.