A Type Theory for Robust Failure Handling in Distributed Systems*

Tzu-Chun Chen[1], Malte Viering[1], Andi Bejleri[1], Lukasz Ziarek[2], and Patrick Eugster[1,3]

[1] Department of Computer Science, TU Darmstadt, Darmstadt, Germany
{tc.chen,viering,bejleri,peugster}@dsp.tu-darmstadt.de
[2] Department of Computer Science and Engineering, SUNY Buffalo, New York, USA
lziarek@buffalo.edu
[3] Department of Computer Science, Purdue University, West Lafayette, USA

Abstract. This paper presents a formal framework for programming distributed applications capable of handling partial failures, motivated by the non-trivial interplay between failure handling and messaging in asynchronous distributed environments. Multiple failures can affect protocols at the level of individual interactions (**alignment**). At the same time, only participants affected by a failure or involved in its handling should be informed of it, and its handling should not be mixed with that of other failures **(precision)**. This is particularly challenging, as through the structure of protocols, failures may be linked to others in subsequent or concomitant interactions **(causality)**. Last but not least, no central authority should be required for handling failures **(decentralisation)**. Our goal is to give developers a description language, called protocol types, to specify robust failure handling that accounts for alignment, precision, causality, and decentralisation. A type discipline is built to statically ensure that asynchronous failure handling among multiple endpoints is free from orphan messages, deadlocks, starvation, and interactions are never stuck.

Keywords: Session Types, Partial Failure Handling, Distributed Systems

1 Introduction

For distributed systems where application components interact asynchronously and concurrently, the design and verification of communication protocols is critical. These systems are prone to *partial failures*, where some components or interactions may fail, while others must continue while respecting certain invariants. Since not all failures can be simply *masked* [10], programmers must *explicitly* deal with failures.

To ease the burden on programmers for constructing resilient communication protocols in the presence of partial failures, we propose a framework for *robust failure handling*. Our framework ensures safety during normal execution and in case of failures. In particular our framework provides the following properties:

* Financially supported by ERC grant FP7-617805 "LiVeSoft – Lightweight Verification of Software".

1. **P1 (alignment)**: The occurrences of failures are specified at the level of individual interactions, which they be raised.
2. **P2 (precision)**: If a failure occurs, an endpoint is informed iff it is affected by the failure or involved in handling it, and its handling is not mixed with that of other failures.
3. **P3 (causality)**: Dependencies between failures are considered, i.e., a failure can affect (enable, disable) others which may occur in subsequent or concomitant interactions.
4. **P4 (decentralisation)**: No central authority or component controls the decisions or actions of the participants to handle a failure.

Inspired by session types [12,19], we introduce *protocol types* to achieve these properties. The basic design is shown in Eq. (1):

$$\mathbf{T}[p_1 \to p_2 : \widetilde{S} \vee f_1, ..., f_n; \mathbf{G}]\mathbf{H}[f_1 : \mathbf{G}_1, ..., f_n : \mathbf{G}_n, ...] \tag{1}$$

where the first term expresses that a participant p_1 either sends a message of type $\widetilde{S}$ to another participant p_2, or raises one of several failures (i.e., $f_1, ..., f_n$). $\mathbf{G}$ specifies the subsequent interactions and $\mathbf{G}_i$ specifies the handling protocols for f_i, $i = 1..n$ in $\mathbf{H}$. In short, failures are thus associated with elementary interactions (**P1**), only participants affected by such a failure in $\mathbf{G}$ (e.g., they are expecting communication from p_2) or those involved in the corresponding failure handling activity are informed (**P2**). There is at most one f appearing in $\mathbf{H}$ that will be handled/raised (**P3**). Our semantics ensure that there is no central authority (**P4**), meaning, notifications of failures (and absence thereof) are delivered asynchronously from failure sources and processes are typed by local (i.e., endpoint) types achieved by the projection of participants over protocol types.

This design results in a distinguishable type system from the design of Eq. (2):

$$\mathbf{T}[p_1 \to p_2 : \widetilde{S}; \mathbf{G}]\mathbf{H}[\mathbf{G}'] \tag{2}$$

In Eq. (2), only one failure may occur in a try-block and where/when it will occur is not specified; once a failure — no matter which one — occurs somewhere in $p_1 \to p_2 : \widetilde{S}$ or $\mathbf{G}$, the handling activity $\mathbf{G}'$ simply takes over. Previous works on dealing with failures [2,3,4,6] are based on Eq. (2) and/or centralised authorities, and hence do not satisfy all of **P1** to **P4** (see Section 7 for details).

Just as multiparty session types aim to specify the interactions among participants and verify implementations of these participants, protocol types specify the global interplay of failures in interactions among participants and verify the failure-handling activities of these participants. To the best of our knowledge, this is the first work that presents a type system for statically checking fine-grained failure handling activities across asynchronous/concurrent processes for partial failures in practical distributed systems. We define a calculus of processes with the ability to not only raise ("throw") and handle ("catch") failures, but to also automatically notify processes of failures or absence thereof at runtime.

Our framework also gives protocol designers and endpoint application developers simple and intuitive description/programming abstractions, and ensures safe interactions among endpoint applications in concurrent environments.

Paper structure. Section 2 gives motivating examples to introduce the design of protocol types, which capture the properties **P1** to **P4**; then we introduce an operation called *transformation* to transform a protocol type to local (i.e., endpoint) types while preserving the desired properties. Section 3 gives a process calculus with de-centralised multiple-failure-handling capability, including the syntax for programs and networks, and the operational semantics for runtime. Section 4 gives a type system for local processes and Section 5 gives a type system for networks to maintain communication coherent. Section 6 states the property of safety, including subject reduction and communication safety, and the property of progress. Section 7 discusses related works. Finally Section 8 concludes our work.

Detailed formal and auxiliary definitions, lemmas, and proofs are presented in the extended version of this paper [9].

2 Protocol Types, Local Types, and Transformation

This section uses examples to show the design behind protocol types and uses Figure 1 to illustrate an operation called *transformation*, which generates local types from protocol types. All formal definitions can be found in the extended version of this paper [9].

The first example, visualised by Figure 1, shows the properties **P1** and **P2**. Assume that in a network all outgoing traffic passes through a proxy (Proxy), monitoring the traffic and logging general information, e.g., consumed bandwidth. The proxy sends this information to a log server (Log). If the proxy detects suspicious behaviour in the traffic, it raises a SuspiciousB failure and notifies Log and a supervision server (SupServer) to handle the failure by having Log send the traffic logs to SupServer; if the proxy detects that the quota of Client is low, it raises a QuotaWarn failure and notifies Log and Client to handle the failure by having Log send quota information to Client. Then Proxy forwards the traffic from Client to an external server (EServer). We propose the following type to formalise the above scenario:

$$\begin{aligned}\mathbf{G}_{proxy} = \mathbf{T}[&\mathsf{Client} \to \mathsf{Proxy} : \mathsf{str};\\ &\mathsf{Proxy} \to \mathsf{Log} : \mathsf{str} \vee \mathsf{SuspiciousB}, \mathsf{QuotaWarn}; \mathsf{Proxy} \to \mathsf{EServer} : \mathsf{str}]\\ \mathbf{H}[&\mathsf{SuspiciousB} : \mathsf{Log} \to \mathsf{SupServer} : \mathsf{str},\ \mathsf{QuotaWarn} : \mathsf{Log} \to \mathsf{Client} : \mathsf{str}]; \mathsf{end}\end{aligned}$$

It specifies that either SuspiciousB or QuotaWarn may occur at interaction Proxy $\to$ Log (**P1**). Proxy can raise the failure and correspondingly sends *failure/non-failure notifications* to all relevant parties. SuspiciousB affects Log and SupServer since they both handle that failure; it also affects Client because the occurrence of SuspiciousB implies that QuotaWarn will never occur, and thus Client will not yield to the handling activity for QuotaWarn. When QuotaWarn occurs, the situation is similar to SuspiciousB's. Once one of them occurs, Proxy sends failure notifications carrying the occurred failure to Log, SupServer, and Client; when no failures occur, Proxy sends non-failure notifications carrying both failures to the same participants to inform them not to yield to handling activity. No failures will affect Proxy and EServer (**P2**) because Proxy and EServer are not involved in any failure handling activities: Proxy continues sending

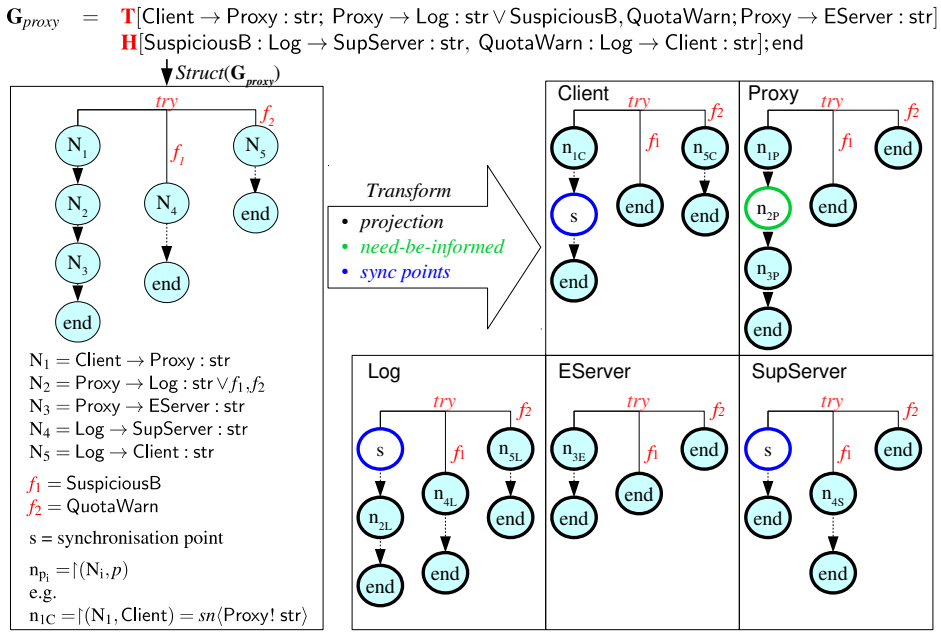

Fig. 1: Overview of *transformation* to obtain local types from a protocol type. The left-hand-side figure visualises a protocol type, $\mathbf{G}_{proxy}$, as a global structure by function *Struct*. The right-hand-side figures are local types for participants in $\mathbf{G}_{proxy}$. They are gained by an operation called *transformation*, which firstly *projects* the global structure onto participants to get *simple* local types, then adds the information of *need-be-informed* participants to the positions with green rings, and finally adds *synchronisation points* to the positions with blue rings. After *transformation*, local types for robust failure handling are reached.

to EServer after it raises a failure and EServer still receives a message from Proxy as expected.

The next example shows **P2** and **P3**. Assume a resource provider (RP) informs a coordinator (Coord) of which resources it can provide. The library (Lib) can get the resource by requesting Coord only if RP sends a list of resources to Coord. If failure NoRes occurs, meaning absence of resource, Coord informs Lib of this; otherwise, Lib places a request to Coord or raises a failure Abort (due to one of many possible local problems) and Coord invokes Record to record this failure:

$$\mathbf{G}_{coord} = \mathbf{T}[\mathsf{RP} \to \mathsf{Coord} : \mathsf{str} \vee \mathsf{NoRes}; \mathsf{Lib} \to \mathsf{Coord} : \mathsf{str} \vee \mathsf{Abort}; \mathsf{Coord} \to \mathsf{Lib} : \mathsf{int}]$$
$$\mathbf{H}[\mathsf{NoRes} : \mathsf{Coord} \to \mathsf{Lib} : \mathsf{bool},\ \mathsf{Abort} : \mathsf{Coord} \to \mathsf{Record} : \mathsf{str}]; \mathsf{end}$$

Although it may seem that interactions RP → Coord and Lib → Coord can run concurrently, this is not the case because, Lib can only get the resource if RP gives Coord a resource list, which implies failures NoRes and Abort are dependent (**P3**). Additionally, we constrain that failure handling should be not be mixed (**P2**). *Synchronisation* of

$$\text{(Sorts) } S ::= \mathsf{bool} \mid \mathsf{unit} \mid \mathsf{int} \mid \mathsf{str} \quad \text{(Failure) } f ::= \mathsf{AuthFail} \mid \mathsf{Abort} \mid \mathsf{ExecFail} \mid \ldots$$
$$\text{(Handlers) } h ::= \emptyset \mid h, f : g \quad \text{(Set of Failures) } F ::= \emptyset \mid F, f$$
$$\text{(Inter. Types) } g ::= \varepsilon \mid p \to p : \widetilde{S} \vee F \mid \mathbf{T}[g]\mathbf{H}[h] \mid g;g \mid t \mid \mu t.g \quad \text{(Protocol Types) } \mathbf{G} ::= g;\mathsf{end}$$

Fig. 2: Syntax of protocol types.

Lib to yield to the completion of RP → Coord is thus needed. This helps programmers tremendously in reasoning about the states that participants are in after failures. Note that, if RP → Coord and Lib → Coord have no failures specified, then RP → Coord and Lib → Coord can run concurrently because there is no failure dependency.

2.1 Protocol Types

To handle *partial failures* in interactions which exhibit the properties **P1** to **P4** , Figure 2 defines protocol types based on the definition of session types given in the work by Bettini *et al.* [1]. Protocol types, denoted by **G**, are composed of interaction types g and terminated by end. We use $(p,\ldots)$ to range over identifiers, $(S,\ldots)$ to range over basic types like bool, unit, int, and str, and $(F,\ldots)$ to range over sets of failures. We highlight the key concepts:

(A) $p_1 \to p_2 : \widetilde{S} \vee F$ is a *failure-raising interaction tagged with* F, or F*-raising interaction* for short. When $\widetilde{S} \neq \emptyset$ and $F \neq \emptyset$, either p_1 sends a content of type $\widetilde{S}$ or raises *one* failure in F to p_2. When one of $\widetilde{S}$ and F is empty, p_1 only makes an output based on the non-empty one. We do not allow both F and $\widetilde{S}$ to be empty.

(B) $\mathbf{T}[g]\mathbf{H}[h]$ defines default interaction in g, which is an interaction type, and a *handling environment*, $h = \{f_i : g_i\}_{i \in I}$, which maps failures to handling activities defined in global types. Our design allows h to deal with different failures, with exactly one handler taking over once failures occur.

(C) In (A), if F is empty, we do not require the interaction to be enclosed in a try-handle term; otherwise, the interaction *must* appear within a try-handle term.

In the remaining syntax, we use ε for idle, and $g_1;g_2$ for sequential composition. A type variable is denoted by t, and a recursive type under an equi-recursive approach [18] is denoted by $\mu t.g$, assuming every t appearing in g is guarded by prefixes.

For brevity, our protocol types presentation omits parallel composition, thus we do not allow session interleaving or multi-threading at a local participant. Note that we can still implement two individual interactions running in parallel by implementing two disjoint groups of interacting participants who execute two respective protocols. We omit branching with multiple options of ongoing interactions, since the term $\mathbf{T}[p_1 \to p_2 : l_1,\ldots,l_n;g]\mathbf{H}[l_1 : g_1,\ldots,l_n : g_n]$ is able to encode the branching in multiparty session types [1,12] by using failures $l_1,\ldots,l_n$ as labels for branches. We leave unaffected participants to continue default actions regardless of the occurrence of failures; we do not inform them of the failure. Moreover, we do not have well-formedness constraints on the shape of interactions in branches (i.e. failure handlers in our syntax) as most multiparty session types and choreographic programming related works require [1,3,4,5,8,12,13,16,17,21].

2.2 Local Types and Transformation

In order to achieve our desired properties we use an operation, called *transformation*, to synthesize a guidance to locally guide which participants need to coordinate with others once a failure occurs, or inversely to assert that none has occurred, before proceeding with the next action. The operation *transformation* includes the following steps:

1. **Generating a global structure** from a given protocol type and alpha renaming it.
2. **Projecting** the above structure to every participants to obtain *simple* local types, which are not yet sufficient for robust failure handling. The projection algorithm is similar to the mechanism in multiparty session types [1,12].
3. **Adding** the information of *need-(to)-be-informed* participants, who are those affected by or involved in handling failures, and synchronisation points to local types.

After these 3 steps, we obtain local types which are sufficient for our type system to ensure *robust failure handling*.

Figure 1 uses the example of $\mathbf{G}_{proxy}$ to demonstrate the operation of *transforming* a protocol type to each participants' local types, which are defined below:

Definition 1 (Local Types T).

$$\text{(Local Types) } \mathbf{T} ::= \mathbf{n} \mid try\{\mathbf{T},\ \mathbf{H}\} \mid \mathbf{n} \dashrightarrow \mathbf{T} \mid t \mid \mu t.\mathbf{T} \quad \text{(Handlers) } \mathbf{H} ::= \emptyset \mid \mathbf{H}, f : \mathbf{T}$$

$$\text{(Action)} \qquad \mathbf{n} ::= \varepsilon \mid \mathsf{end} \mid sn\langle p!\ \widetilde{S} \vee F, \widetilde{p}, \widetilde{p}\rangle \mid rn\langle p?\ \widetilde{S} \vee F\rangle \mid yield\langle F\rangle$$

A local type is either an action ($\mathbf{n}$), a try-handle type ($try\{\mathbf{T},\mathbf{H}\}$), a sequencing type ($\mathbf{n} \dashrightarrow \mathbf{T}$), a local type variable (t), or a recursive type ($\mu t.\mathbf{T}$). We use ε to type an idle action, while end types termination. A sending action, typed by $sn\langle p!\ \widetilde{S} \vee F, \widetilde{p}', \widetilde{p}''\rangle$, specifies a sending of normal content of type $\widetilde{S}$ to p or raising a failure in F. When a failure is raised, the sender also sends failure notifications to participants $\widetilde{p}'$. When normal content is sent, the sender also sends non-failure notifications to participants $\widetilde{p}''$. A receiving action, typed by $rn\langle p?\ \widetilde{S} \vee F\rangle$, specifies the reception of content of type $\widetilde{S}$ from p, who may raise a failure in F instead. An action yielding to the arrival of a non-failure notification informing that no failures in F occurred, is typed by $yield\langle F\rangle$. A handling environment in local types, denoted by $\mathbf{H} = \{f_i : \mathbf{T}_i\}_{i\in I}$, maps failures to corresponding local handling actions defined in local types.

In Figure 1, we firstly create a **global structure** for $\mathbf{G}_{proxy}$ by $Struct(\mathbf{G}_{proxy})$. Global structures, denoted by $\mathtt{T}$, consist of either a single interaction ($\mathtt{N}$), a try-handle structure ($try\{\mathtt{T},\mathtt{H}\}$) where $\mathtt{H}$ has a similar shape to handling environments in local types, a sequence ($\mathtt{N} \dashrightarrow \mathtt{T}$), or a recursive structure ($\mu t.\mathtt{T}$). We define $\mathtt{N}$ as either $p \to p : \widetilde{S} \vee F$ or ε or end. By defining $Struct(\text{single interaction}) = \mathtt{N}$, a try-handle structure is obtained by $Struct(\mathbf{T}[g]\mathbf{H}[\ f_1 : g_1, ..., f_n : g_n];\mathbf{G}) = try\{Struct(g;\mathbf{G}),\ f_1 : Struct(g_1;\mathbf{G}), ...,\ f_n : Struct(g_n;\mathbf{G})\}$, while a recursive structure is obtained by $Struct(\mu t.\mathbf{G}) = \mu t.Struct(\mathbf{G})$.

The *simple* local types are gained by *projecting* $\mathtt{T}$, created by $Struct(\mathbf{G})$, on each participants. The projection rules are defined below:

Definition 2 (Projecting $\mathtt{T}$ onto Endpoint p). Assume $\mathtt{T} = Struct(\mathbf{G})$ and $\mathtt{T}$ is alpha-renamed so that all failures in $\mathtt{T}$ are unique. Define $\upharpoonright(\mathtt{T}, p)$ as generating a local type on

p:

$$(1)\ \upharpoonright(\mathsf{end},p)=\mathsf{end}\quad (2)\ \upharpoonright(p_1\to p_2:\widetilde{S}\vee F,p)=\begin{cases} sn\langle p_2!\,\widetilde{S}\vee F,\ \_\ ,\ \_\rangle & \text{if } p=p_1\neq p_2\\ rn\langle p_1?\,\widetilde{S}\vee F\rangle & \text{if } p=p_2\neq p_1\\ \varepsilon_F & \text{otherwise}\end{cases}$$

$$(3)\ \upharpoonright(try\{\mathtt{T},f_1:\mathtt{T}_1,...,f_n:\mathtt{T}_n\},p)=try\{\upharpoonright(\mathtt{T},p),f_1:\upharpoonright(\mathtt{T}_1,p),...,f_n:\upharpoonright(\mathtt{T}_n,p)\}$$

$$(4)\ \upharpoonright(\mathtt{N}\dashrightarrow\mathtt{T},p)=\upharpoonright(\mathtt{N},p)\dashrightarrow\upharpoonright(\mathtt{T},p)\quad (5)\ \upharpoonright(t,p)=t\qquad (6)\ \upharpoonright(\mu t.\mathtt{T},p)=\mu t.\ \upharpoonright(\mathtt{T},p)$$

Others are undefined.

Rule (2) is for dually interacting participants. It introduces ε_F, which has equivalent meaning to ε (i.e. idle action) but is only used in *transformation* for adding synchronisation points. As we project an interaction $p_1\to p_2:\widetilde{S}\vee F$ with $p_1\neq p_2$ onto p_1 (resp. p_2), we get an action $sn\langle p_2!\,\widetilde{S}\vee F,\ \_\ ,\ \_\rangle$ (resp. $rn\langle p_1?\,\widetilde{S}\vee F\rangle$). Note that the two slots $\_$ in the sending action are preserved for adding the need-be-informed participants as a failure occurs (the first slot), and those as no failures occur (the second slot). As we project the interaction to some participant who is *not* in the interaction, we get ε_F (idle action). The subscript F indicates that if p is affected by some failures in F a synchronisation point will be added at this position. Rule (3) simply projects every sub-structure in the try block and handlers onto the participant. Rule (4) sequences two local types projected from a global structure. Other rules are straightforward.

After projection, we add *need-be-informed participants* into the failure-raiser's sending actions (e.g., the one marked in green ring in Figure 1). We use $C(\mathtt{T},F)$ to get the set of need-be-informed participants regarding a unique F in a global structure $\mathtt{T}$. It is the least fixed point of $c(\mathtt{T},\mathtt{T},F,r)$, which recursively collects the need-be-informed participants regarding F based on $\mathtt{T}$. Since for every protocol the number of participants is finite, function c will converge to a fixed set of participants. The key calculation is done by the rule below

$$c(\mathtt{T},\mathtt{N},F,r)=\begin{cases} r\cup pid(\mathtt{N})\cup C(\mathtt{T},FSet(\mathtt{N})) & \text{if } (F \text{ appears before } \mathtt{N} \text{ in } \mathtt{T})\wedge\\ & ((pid(\mathtt{N})\cap r\neq\emptyset)\vee(FSet(\mathtt{N})\neq\emptyset))\\ r & \text{otherwise}\end{cases}$$

where we require the *initial* r to be the set containing the receiver of F-raising interaction and the participants involved in handling F in $\mathtt{T}$; later r acts as an accumulator collecting the participants causally related to the initial r. $\mathtt{N}$ is an interaction, $pid(\mathtt{N})$ is the set of participants in $\mathtt{N}$, and $FSet(\mathtt{N})$ returns the failures tagged on $\mathtt{N}$. This rule says that if the interaction we are checking appears after the F-raising interaction, and some of its interacting participants are related to r or the interaction itself can raise another failure set (e.g. the interaction $\mathsf{Lib}\to\mathsf{Coord}:\mathsf{str}\vee\mathsf{Abort}$ in $\mathsf{RP}\to\mathsf{Coord}:\mathsf{str}\vee\mathsf{NoRes};\mathsf{Lib}\to\mathsf{Coord}:\mathsf{str}\vee\mathsf{Abort}$ is related to $F=\{\mathsf{NoRes}\}$), then we collect its participants (i.e. $pid(\mathtt{N})$) and the need-be-informed participants with respect to the failures that can be raised by $\mathtt{N}$.

After adding those participants, we add synchronisation points $yield\langle F\rangle$ to the positions where a participant yields to the arrival of non-failure notifications (e.g. those marked in blue rings in Figure 1). The key rule is:

$$Sync(\mathtt{T},\mathbf{n},p) = \begin{cases} yield\langle FSet(\mathtt{N})\rangle \dashrightarrow \mathbf{n} & \text{if } (\mathbf{n} = \restriction(\mathtt{N},p)) \wedge (p \in C(\mathtt{T},FSet(\mathtt{N}))) \wedge p \neq snd(\mathtt{N}) \\ \mathbf{n} \dashrightarrow yield\langle FSet(\mathtt{N})\rangle & \text{if } (\mathbf{n} = \restriction(\mathtt{N},p)) \wedge (p \in C(\mathtt{T},FSet(\mathtt{N}))) \wedge p = snd(\mathtt{N}) \\ \mathbf{n} & \text{otherwise} \end{cases}$$

where $snd(\mathtt{N})$ is the sender for interaction $\mathtt{N}$. This rule says the followings: If p needs to be informed of $F = FSet(\mathtt{N})$ (i.e. $p \in C(\mathtt{T},FSet(\mathtt{N}))$) then it must add a synchronisation point. If p's action (i.e., $\mathbf{n}$) regarding F is ε_F (e.g. in $\mathbf{G}_{coord}$ the participant Lib has $\varepsilon_{\mathsf{NoRes}}$ by Definition 2), $yield\langle F\rangle$ is positioned ahead of p's action (e.g. a sending action of Lib to Coord specified in $\mathbf{G}_{coord}$), because p needs to wait for the notification regarding F before taking any action. If p is the receiver, we have $yield\langle F\rangle$ positioned before the receiving action because $yield\langle F\rangle$ is the point deciding whether the process will handle a failure regarding F or proceed. If p is the sender, we should have $yield\langle F\rangle$ positioned after the sending action, because as p is involved for some failure handling activity regarding F, it needs to first send out failure notifications then go back to execute the handling activity; otherwise the process will get stuck.

In Figure 1, green ring appears at Proxy's second action because, if a failure occurs, Proxy has to inform Log, SupServer, and Client about that failure. Blue rings appear at Client, Log, and SupServer's try blocks because they are involved in handling activity, and they can terminate only after getting the notifications that no failures occurred.

Overall, we define the operation of transformation as $Transform(\mathbf{G},p)$, which transforms $\mathbf{G}$ to a local type for p.

3 Processes for Decentralised Multiple-Failure-Handling

We abstract distributed systems as a finite set of processes communicating by outputting (resp. inputting) messages into (resp. from) the shared global queue asynchronously and concurrently. The semantics of the calculus is in the same style as that of the π-calculus and does not involve any centralised authority for specifying how messages are exchanged (**P4**). The shared queue is only *conceptually* global for convenience, and could be split into individual participant queues.

Syntax. In Figure 3, we define x as value variables, y as channel variables, a as shared names (e.g., names for services or protocol managers), and s as session names (i.e., session IDs), p as participant identifiers, and X as process variables. We use u for names and c for channels, which are either variables or a combination of s and p. The definition for expressions e is standard. We define the syntax of processes $(P,...)$ and that of networks $(N,...)$, which represent interactions of processes at runtime. Process $\mathbf{0}$ is inactive. Process $c!\langle p,\langle\tilde{e}\rangle^F\rangle$ denotes an output, which may alternatively raise a failure f in F, sends a message with content $\tilde{e}$ to p via channel c; while $c?(p,(\tilde{x})^F).P$ denotes an input using c to receive a content from p, which may alternatively raise a failure from F. Every $\tilde{x}$ appearing in P is bound by the input prefix. When $F = \emptyset$, we omit F since the process will not raise/receive a failure. Process $c\ \mathsf{raise}(f)\ \mathsf{to}\ p$ raises f to p via channel c. Process $c \otimes F$ is guarded by $\otimes F$, a synchronisation point, yielding to non-failure notification for F. A try-handle process $\mathsf{try}\{P\}\mathsf{h}\{H\}$ executes P until a handler $f \in dom(H)$ is triggered, then the triggered handler takes over. A handling environment,

$$
\begin{array}{rl}
\text{(Variables) } x,y \quad \text{(Shared Names) } a \quad \text{(Session Names) } s \quad \text{(Process IDs) } p \quad \text{(Process Variables) } X & \\
\text{(Values)} & v ::= \mathsf{unit} \mid \mathsf{false} \mid \mathsf{true} \mid 1 \mid 2 \mid ... \qquad \text{(Names) } u ::= a \mid s \\
\text{(Expressions)} & e ::= x \mid y \mid v \mid u \mid e+e \mid -e \mid e \vee e \mid ... \qquad \text{(Channels) } c ::= y \mid s[p] \\
\text{(Processes)} & P ::= \mathbf{0} \mid c!(p,\langle \tilde{e} \rangle^F) \mid c?(p,(\tilde{x})^F).P \mid c\ \mathsf{raise}(f)\ \mathsf{to}\ p \mid c \otimes F \\
& \quad \mid \mathsf{try}\{P\}\mathsf{h}\{H\} \mid P;P \mid X\langle \tilde{e}\ c \rangle \mid \mathsf{def}\ D\ \mathsf{in}\ P \mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P \\
\text{(Handlers)} & H ::= \emptyset \mid H, f : P \qquad \text{(Declaration) } D ::= X(\tilde{x}\ c) = P \\
\text{(Messages)} & m ::= \varepsilon \mid \langle p,p,\langle \tilde{v} \rangle \rangle^F \mid \langle p,f \rangle \mid \langle\langle p,F \rangle\rangle \qquad \text{(Context) } \mathscr{E} ::= [\,] \mid \mathsf{try}\{\mathscr{E}\}\mathsf{h}\{H\} \mid \mathscr{E};P \\
\text{(Network)} & N ::= a[p](y).P \mid [P]_{\mathbf{T}} \mid s : q \mid N \| N \mid (\mathsf{new}\ s)N \quad \text{(Queues) } q ::= m \mid q \cdot m
\end{array}
$$

Fig. 3: Syntax for processes and networks.

denoted by H, maps failure names to handling processes. We write $P_1;P_2$ to represent a sequential composition where P_2 follows P_1. Process $\mathsf{def}\ D\ \mathsf{in}\ P$ defines a recursion, where declaration $D = (X(\tilde{x}\ c) = P)$ is a recursive call. The term $\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P$ is standard. We define evaluation context $\mathscr{E}$ over processes. It is either a hole, a context in a try-handle term, or a context sequencing next processes.

A network N is composed by linking points, denoted by $a[p](y).P$, and runtime processes, denoted by $[P]_{\mathbf{T}}$ with global transports (i.e., $s : q$) for proceeding communications in a private session (i.e., $(\mathsf{new}\ s)N$). Our framework asks a process to join one session at a time. A linking point $a[p](y).P$ is guarded by $a[p](y)$ for session initiation, where shared name a associates a service to a protocol type. $[P]_{\mathbf{T}}$ represents a runtime process which is guided by $\mathbf{T}$ for notifying need-be-informed participants.

A session queue, denoted by $s : q$, is a queue for messages floating in session s. Message $\langle p_1,p_2,\langle \tilde{v} \rangle \rangle^F$ carries content $\tilde{v}$, sent from p_1 to p_2 prone to failure $f \in F$. Message $\langle p,f \rangle$ (resp. $\langle\langle p,F \rangle\rangle$) carries a failure name f to indicate that *failure f occurred* (resp. a set of failures F to indicate that *no failures in F occurred*) to p. Conventionally we say $\langle\langle p,\emptyset \rangle\rangle = \varepsilon$. When session s is initiated for a network, a private (i.e., hidden) session is created, in which activities cannot be witnessed from the outside. We use structural congruence rules, defined by $\equiv$, which are standard according to the works of multiparty session types [1,12].

Operational semantics. Figure 4 gives the operational semantics for networks (i.e., runtime processes) through the reduction relation $N \to N$. We have added boxes to those rules which differ from standard session type definitions. In rule **[link]**, a session is generated with a fresh name s through shared name a obeying protocol type $\mathbf{G}$. This indicates that all processes in the new session s will obey to the behaviours defined in $\mathbf{G}$. At the same time, a global queue $s : \varepsilon$ is generated, and the local process associated with p replaces y_p with $s[p]$; a local type $\mathbf{T}_p$ is generated by $\mathit{Transform}(\mathbf{G},p)$ to guide the local process associated with p for propagating notifications. Note that, as we enclose a local process with $\mathbf{T}$, together they become an element of a network. $\mathbf{T}$ is merely a local type and the reduction of the network does not change $\mathbf{T}$.

Rule **[rcv]** states that, in s, a process associated with p_1 is able to receive a value $\tilde{v}$ from participant associated with p_2 and message $\langle p_2,p_1,\langle \tilde{v} \rangle \rangle^F$ is on the top of q. Then $\tilde{v}$ will replace the free occurrences of $\tilde{x}$ in P. The shape of $s[p_1]?(p_2,(\tilde{x})^F)$ indicates that

$$\frac{a:\langle \mathbf{G}\rangle \quad \forall p\in\{1..n\}.\mathbf{T}_p = \mathit{Transform}(\mathbf{G},p) \quad s \text{ fresh}}{a[1](y_1).P_1 \parallel ... \parallel a[n](y_n).P_n \to (\mathsf{new}\ s)([P_1[s[1]/y_1]]_{\mathbf{T}_1} \parallel ... \parallel [P_n[s[n]/y_n]]_{\mathbf{T}_n} \parallel s:\varepsilon)}\ \textbf{[link]}$$

$$[s[p_1]?(p_2,(\tilde{x})^F).P]_{\mathbf{T}} \parallel s:\langle p_2,p_1,\langle \tilde{v}\rangle\rangle^F\cdot q \to [P[\tilde{v}/\tilde{x}]]_{\mathbf{T}} \parallel s:q\ \textbf{[rcv]}$$

$$[s[p_1]!(p_2,\langle \tilde{e}\rangle);P]_{\mathbf{T}} \parallel s:q \to [P]_{\mathbf{T}} \parallel s:q\cdot\langle p_1,p_2,\langle \tilde{v}\rangle\rangle \quad \tilde{e}\Downarrow\tilde{v}\ \textbf{[snd]}$$

$$\frac{\tilde{e}\Downarrow\tilde{v} \quad \mathit{node}(\mathbf{T},F) = sn\langle p_2!\,\widetilde{S}\vee F,\widetilde{p},\{p_1',...,p_n'\}\rangle \quad F\neq\emptyset}{\begin{array}{l}[\mathsf{try}\{s[p_1]!(p_2,\langle\tilde{e}\rangle^F);P\}\mathsf{h}\{H\}]_{\mathbf{T}} \parallel s:q \\ \quad\to [\mathsf{try}\{P\}\mathsf{h}\{H\}]_{\mathbf{T}} \parallel s:q\cdot\langle p_1,p_2,\langle\tilde{v}\rangle\rangle^F\cdot\langle\langle p_1',F\rangle\rangle...\langle\langle p_n',F\rangle\rangle\end{array}}\ \textbf{[sndF]}$$

$$\frac{\mathit{node}(\mathbf{T},F) = sn\langle p_2!\,\widetilde{S}\vee F,\{p_2,p_1',...,p_n'\},\widetilde{p}\rangle \quad f\in F}{\begin{array}{l}[\mathsf{try}\{s[p_1]\ \mathsf{raise}(f)\ \mathsf{to}\ p_2;P\}\mathsf{h}\{H\}]_{\mathbf{T}} \parallel s:q \\ \quad\to [\mathsf{try}\{P\}\mathsf{h}\{H\}]_{\mathbf{T}} \parallel s:q\cdot\langle p_2,f\rangle\cdot\langle p_1',f\rangle...\langle p_n',f\rangle\end{array}}\ \textbf{[thwf]}$$

$$\frac{\begin{array}{c}\mathit{act}(P) = s[p] \quad [P]_{\mathbf{T}} \parallel s:q \to [P']_{\mathbf{T}} \parallel s:q' \\ q = \langle p',f'\rangle\cdot q' \Rightarrow (p'\neq p)\vee(f'\notin \mathit{dom}(H))\end{array}}{[\mathsf{try}\{P\}\mathsf{h}\{H\};P'']_{\mathbf{T}} \parallel s:q \to [\mathsf{try}\{P'\}\mathsf{h}\{H\};P'']_{\mathbf{T}} \parallel s:q'}\ \textbf{[try]}$$

$$\frac{f\in \mathit{dom}(H)\cap F}{[\mathsf{try}\{\mathscr{E}[s[p]\otimes F]\}\mathsf{h}\{H\};P']_{\mathbf{T}} \parallel s:\langle p,f\rangle\cdot q \to [H(f);P']_{\mathbf{T}} \parallel s:q}\ \textbf{[hdl]}$$

$$\frac{F'\subseteq F}{[s[p]\otimes F';P]_{\mathbf{T}} \parallel s:\langle\langle p,F\rangle\rangle\cdot q \to [P]_{\mathbf{T}} \parallel s:q}\ \textbf{[sync-done]}$$

$$\frac{F''=F'\setminus F\neq\emptyset}{[s[p]\otimes F';P]_{\mathbf{T}} \parallel s:\langle\langle p,F\rangle\rangle\cdot q \to [s[p]\otimes F'';P]_{\mathbf{T}} \parallel s:q}\ \textbf{[sync]}$$

$$[\mathsf{try}\{v\}\mathsf{h}\{H\}]_{\mathbf{T}} \to [\mathbf{0}]_{\mathbf{T}}\ \textbf{[try-end]}$$

$$[\mathsf{if\ true\ then}\ P_1\ \mathsf{else}\ P_2]_{\mathbf{T}} \to [P_1]_{\mathbf{T}} \quad [\mathsf{if\ false\ then}\ P_1\ \mathsf{else}\ P_2]_{\mathbf{T}} \to [P_2]_{\mathbf{T}}\ \textbf{[if]}$$

$$\frac{[P_1]_{\mathbf{T}} \parallel s:q \to [P_2]_{\mathbf{T}} \parallel s:q'}{[P_1;P]_{\mathbf{T}} \parallel s:q \to [P_2;P]_{\mathbf{T}} \parallel s:q'}\ \textbf{[seq]} \qquad \frac{\tilde{e}\Downarrow\tilde{v} \quad X(\tilde{x}\ c) = P\in D}{[\mathsf{def}\ D\ \mathsf{in}\ X\langle\tilde{e}\ c\rangle]_{\mathbf{T}} \to [\mathsf{def}\ D\ \mathsf{in}\ (P[\tilde{v}/\tilde{x}])]_{\mathbf{T}}}\ \textbf{[call]}$$

$$\frac{N_1\to N_2}{N_1 \parallel N\to N_2 \parallel N}\ \textbf{[net]} \qquad \frac{[P_1]_{\mathbf{T}} \parallel s:q \to [P_2]_{\mathbf{T}} \parallel s:q'}{[\mathsf{def}\ D\ \mathsf{in}\ P_1]_{\mathbf{T}} \parallel s:q \to [\mathsf{def}\ D\ \mathsf{in}\ P_2]_{\mathbf{T}} \parallel s:q'}\ \textbf{[defin]}$$

$$\frac{N_1\equiv N_2\to N_3\equiv N_4}{N_1\to N_4}\ \textbf{[str]} \qquad \frac{N_1\to N_2}{(\mathsf{new}\ s)N_1\to(\mathsf{new}\ s)N_2}\ \textbf{[new]}$$

Fig. 4: Reduction rules for networks (i.e., runtime processes).

its dual action may send it a failure from F; in other words, if $F\neq\emptyset$, a process should be structured by a try-handle term for possible failure handling. Rule **[snd]** is dual to **[rcv]**. We define $\mathit{node}(\mathbf{T},F)$ as a function returning an action tagged with F in a local type $\mathbf{T}$. Rule **[sndF]** states that, if there is an action in $\mathbf{T}$ matching $s[p_1]!(p_2,\langle\tilde{e}\rangle^F)$ and $\tilde{e}\Downarrow\tilde{v}$, then the process associated with p_1 in s is allowed to send a message with content $\tilde{v}$ to p_2 and non-failure notifications $\langle\langle p_1',F\rangle\rangle...\langle\langle p_n',F\rangle\rangle$. Note that, non failure notifications are automatically generated at runtime. If a process follows the guidance of the attached $\mathbf{T}$, since $\mathbf{T}$ is alpha-renamed, every failure raised by the process is unique. Similarly, **[thwf]** states for a process associated with p_1 in s, to raise f to p_2 and other affected ones, $p_1',..,p_n'$. Very importantly, in **[sndF]** and **[thwf]**, q has no failure notification to trigger H

because, as a failure-raising interaction is ready to fire (i.e. its sender is about to send), it implies that, globally, either this interaction is the first failure-raising interaction in s (thus no failure yet occurs in s), or its previous interactions did not raise a failure in $dom(H)$ (thus by **P2**, this interaction is able to raise a failure in $dom(H)$, and no failures in $dom(H)$ yet occurs in s). For convenience, we use act to extract the channel that a process or the set of handlers is acting on, i.e. $act(P) = s[p]$ says P is acting on channel $s[p]$, and $act(H) = act(H(f))$ for every $f \in dom(H)$.

In [try], if the H in a try-handle process associated with p in s will not be triggered by the top message in $s : q$, then the process in the try block will take action according to the process's interaction with the queue. In [hdl], as f arrives to a try-handle process associated with p in s whose try block is yielding to non-failure notification for F and H is able to handle f, the handling process $H(f)$ takes over. Due to asynchrony, other processes' handlers for f may become active before this process. Thus some messages in q may be sent from other processes' handlers of f for P. Note that none of the messages in q are for $\mathscr{E}$ because, all default sending actions in other processes are also guarded by synchronisation points.

Synchronisation either proceeds with [sync-done], where F is sufficient to remove F', or with [sync], where F is included in F' carried in the notification. For the former, some processes in the failure-handling activity only take care of partial failures in F, i.e. F', when they receive F, to ensure that no failures in F occurred. For the latter, further synchronisation is required by $F'' = (F' \setminus F) \neq \emptyset$. In [try-end], since we have added sufficient synchronisation points to guard processes who must yield to non-failure notifications, when a network reaches $[\mathsf{try}\{v\}\mathsf{h}\{H\}]_{\mathbf{T}}$, it is safe to be inactive because no more failure notifications will occur. In other rules, the operations enclosed in $\mathbf{T}$ are standard according to the works of multiparty session types [1,12].

4 Typing Local Processes

This section introduces rules to type user-defined processes. Based on the multiparty session types [1,12], type environments and typing rules for processes are given in Figure 5. A shared environment Γ is a finite mapping from variables to sorts and from process variables to local types; a session environment Δ is a finite mapping from session channels to local types. $\Gamma, x : S$ means that x does not occur in Γ, so does $\Gamma, X : (\tilde{x}\ \mathbf{T})$ and $\Delta, c : \mathbf{T}$. We assume that expressions are typed by sorts. $\Gamma \vdash e : S$ is the typing judgment for expressions, whose typing rules are standard. The typing judgment $\Gamma \vdash P \triangleright \Delta$ for local processes reads as "Γ proves that P complies with abstract specification Δ".

Rule [T-0] states that an idle process is typed by end-only Δ, which means $\forall c \in dom(\Delta)$, $\Delta(c) = \mathsf{end}$. Rule [T-seq] types sequential composition by sequencing P_2's action in Δ_2 after P_1's action in Δ_1 as long as P_1 and P_2 are acting on the same channel. We define $\Delta_1 \circ \Delta_2$ as the one defined in the multiparty session types extended with failure-handling ability [3]. Rule [T-rcv] specifies that $s[p_1]?(p_2, (\widetilde{x})^F).P$ is valid as it corresponds to local type $rn\langle p_2?\ \widetilde{S} \vee F\rangle \dashrightarrow \mathbf{T}$ as long as P, associated with p_1 in session s, is well-typed by $\mathbf{T}$ under an environment which knows $\tilde{x} : \widetilde{S}$. In [T-snd] and [T-thwf], since the slots are not related to typing, their contents are omitted. Rules [T-snd] and [T-thwf] share the

Γ (Shared Environments) $::= \emptyset \mid \Gamma, x : S \mid \Gamma, X : (\tilde{x}\ \mathbf{T})$ $\quad \Delta$ (Session Environments) $::= \emptyset \mid \Delta, c : \mathbf{T}$

$$\frac{\Delta \text{ end only}}{\Gamma \vdash \mathbf{0} \triangleright \Delta}\ \text{[T-0]} \qquad \frac{\forall i \in \{1,2\}.\Gamma \vdash P_i \triangleright \Delta_i}{\Gamma \vdash P_1; P_2 \triangleright \Delta_1 \circ \Delta_2}\ \text{[T-seq]}$$

$$\frac{\Gamma, \widetilde{x} : \widetilde{S} \vdash P \triangleright s[p_1] : \mathbf{T}}{\Gamma \vdash s[p_1]?(p_2, (\widetilde{x})^F).P \triangleright s[p_1] : rn\langle p_2 ?\ \widetilde{S} \vee F\rangle \dashrightarrow \mathbf{T}}\ \text{[T-rcv]}$$

$$\frac{\widetilde{S} \neq \emptyset \quad \Gamma \vdash \tilde{e} : \widetilde{S} \quad \Gamma \vdash P \triangleright s[p_1] : \mathbf{T}}{\Gamma \vdash s[p_1]!(p_2, \langle \tilde{e} \rangle^F); P \triangleright s[p_1] : sn\langle p_2 !\ \widetilde{S} \vee F, \_, \_\rangle \dashrightarrow \mathbf{T}}\ \boxed{\text{[T-snd]}}$$

$$\frac{f \in F \quad \Gamma \vdash P \triangleright s[p_1] : \mathbf{T}}{\Gamma \vdash s[p_1]\ \mathsf{raise}(f)\ \mathsf{to}\ p_2; P \triangleright s[p_1] : sn\langle p_2 !\ \widetilde{S} \vee F, \_, \_\rangle \dashrightarrow \mathbf{T}}\ \boxed{\text{[T-thwf]}}$$

$$\frac{\begin{array}{c} dom(H) = dom(\mathbf{H}) \quad \forall f \in dom(H).f \notin P' \\ \Gamma \vdash P; P' \triangleright act(H) : \mathbf{T} \quad \forall f \in dom(H).\Gamma \vdash H(f); P' \triangleright act(H) : H(f) \end{array}}{\Gamma \vdash \mathsf{try}\{P\}\mathsf{h}\{H\}; P' \triangleright act(H) : try\{\mathbf{T}, \mathbf{H}\}}\ \boxed{\text{[T-try]}}$$

$$\frac{\Gamma \vdash P \triangleright c : \mathbf{T}}{\Gamma \vdash c \otimes F; P \triangleright c : yield\langle F\rangle \dashrightarrow \mathbf{T}}\ \boxed{\text{[T-sync]}} \qquad \frac{\Gamma \vdash e : \mathsf{bool} \quad i = 1,2.\ \Gamma \vdash P_i; P \triangleright \Delta}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2; P \triangleright \Delta}\ \text{[T-if]}$$

$$\Gamma, X : (\widetilde{S}\ \mathbf{T}) \vdash X\langle \tilde{e}\ c\rangle \triangleright c : \mathbf{T}\ \text{[T-var]} \qquad \frac{\Gamma, \tilde{x} : \widetilde{S}, X : (\widetilde{S}\ \mathbf{T}) \vdash P \triangleright c : \mathbf{T} \quad \Gamma, X : (\widetilde{S}\ \mathbf{T}) \vdash P' \triangleright \Delta}{\Gamma \vdash \mathsf{def}\ X(\tilde{x}\ c) = P\ \mathsf{in}\ P' \triangleright \Delta}\ \text{[T-rec]}$$

Fig. 5: Typing rules for processes.

same action for typing because $sn\langle p_2 !\ \widetilde{S} \vee F, \_, \_\rangle$ specifies two possible actions: a sending action $s[p_1]!(p_2, \langle \tilde{e} \rangle^F)$ in which $\tilde{e}$ must have type $\widetilde{S}$, and action $s[p_1]\ \mathsf{raise}(f)\ \mathsf{to}\ p_2$ in which f must be in F. Then the continuing process P is typed by the following $\mathbf{T}$.

For typing handling activities, rule [T-try] types a try-handle term if its default action (i.e., P) with its following process is well-typed, and those in handlers with their follow-up processes are all well-typed. We require the following process P' should not contain any failure appearing in H. Since P and any processes in H are acting on the same channel and $act(H)$ represents the channel that every processes in H is acting on, we use $act(H)$ to get the channel in order to type $\mathsf{try}\{\mathbf{0}\}\mathsf{h}\{H\}$. Recall Figure 1 and projection rules defined in Definition 2, for local types the sequencing action is linked at every leaf in a try-handle term; in other words, the type of P' is attached to the type of P and also to every handler in $\mathbf{H}$. Therefore, as we type a try-handle term, we also consider its following process.

Rule [T-sync] specifies that process $c \otimes F; P$ is well-typed if the local type for c has synchronisation point $yield\langle F\rangle$ and P is well-typed w.r.t. $\mathbf{T}$. The algorithm for adding synchronisation points (introduced in Section 2) automatically places the synchronisation points in local types and ensures that once a failure is raised, other possible failure-raising actions must not fire. Since the operational semantics defined in Figure 4 only deliver notifications regarding F to need-be-informed participants and only one failure in F can be raised, our type system ensures only one failure in a try-handle term is handled and all participants affected by F have consistent failure handling activities.

$$\begin{array}{rl} \mathbf{M}\text{ (Message Types)} & ::= \varepsilon \mid \langle p_1, p_2, \langle \widetilde{S} \rangle \rangle^F \mid \langle p, f \rangle \mid \langle\langle p, F \rangle\rangle \\ \mathbf{q}\text{ (Queue Type)} & ::= \mathbf{M} \mid \mathbf{q} \cdot \mathbf{M} \\ \Delta\text{ (Extended Session Environments)} & ::= \ldots \mid \Delta, s : \mathbf{q} \end{array}$$

$$\Gamma \vdash s : \varepsilon \triangleright \{s : \varepsilon\} \ \text{[T-m}\varepsilon\text{]} \qquad \frac{\Gamma \vdash \widetilde{v} : \widetilde{S} \quad \Gamma \vdash s : q \triangleright \{s : \mathbf{q}\}}{\Gamma \vdash s : q \cdot \langle p_1, p_2, \langle \widetilde{v} \rangle \rangle^F \triangleright \{s : \mathbf{q} \cdot \langle p_1, p_2, \langle \widetilde{S} \rangle \rangle^F\}} \ \text{[T-m]}$$

$$\frac{\Gamma \vdash s : q \triangleright \{s : \mathbf{q}\}}{\Gamma \vdash s : q \cdot \langle p, f \rangle \triangleright \{s : \mathbf{q} \cdot \langle p, f \rangle\}} \ \text{[T-mf]} \qquad \frac{\Gamma \vdash s : q \triangleright \{s : \mathbf{q}\}}{\Gamma \vdash s : q \cdot \langle\langle p, F \rangle\rangle \triangleright \{s : \mathbf{q} \cdot \langle\langle p, F \rangle\rangle\}} \ \text{[T-mF]}$$

$$\frac{\Gamma \vdash a : \langle \mathbf{G} \rangle \qquad \Gamma \vdash P \triangleright y : \mathit{Transform}(\mathbf{G}, p)}{\Gamma \vdash a[p](y).P \triangleright \emptyset} \ \boxed{\text{[T-link]}}$$

$$\frac{\Gamma \vdash P \triangleright \Delta \quad \mathit{act}(P) = s[p] \quad \exists \mathbf{G}.\ \text{s.t. } \mathbf{T} = \mathit{Transform}(\mathbf{G}, p) \quad \mathbf{T} \text{ contains } \Delta(s[p])}{\Gamma \vdash [P]_{\mathbf{T}} \triangleright \Delta} \ \boxed{\text{[T-guide]}}$$

$$\frac{\forall i \in 1,2. \Gamma \vdash N_i \triangleright \Delta_i \quad \mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_2) = \emptyset}{\Gamma \vdash N_1 \parallel N_2 \triangleright \Delta_1, \Delta_2} \ \text{[T-net]} \qquad \frac{\Gamma \vdash N \triangleright \Delta \quad \Delta\langle s \rangle \text{ coherent}}{\Gamma \vdash (\mathsf{new}\ s) N \triangleright \Delta \setminus \Delta\langle s \rangle} \ \text{[T-new]}$$

Fig. 6: Typing rules for networks.

Rule [T-var] types a local process variable, and rule [T-rec] types a recursion with Δ, where the recursive call $X(\tilde{x}, c) = P$ is typed by $c : \mathbf{T}$, indicating that P follows behaviour $\mathbf{T}$ at c. Others are standard according to the works of multiparty session types [1,12].

5 Typing the Network

Ultimately our framework needs to ensure that the network is coherent. *Coherence*, according to the works of multiparty session types [1,12], describes an environment where all *interactions* are complying with the guidance of some $\mathbf{G}$, such that the behaviour of every participant in Δ, say $s[p]$, obeys to $\mathit{Transform}(\mathbf{G}, p)$, which denotes a local type. To reason about coherence of default and handling interactions in a session, we statically type check the interactions by modeling the outputs and inputs among local processes and the shared global queue.

The typing rules for networks are defined in Figure 6 by extending the session environments such as to map queues to queue types. A queue type, denoted by $\mathbf{q}$, is composed by message types, which are typed by their contents or shapes: Rule [T-mε] types an empty queue, while rule [T-m] types a message carrying a value under the assumption that $\Gamma \vdash \widetilde{v} : \widetilde{S}$ and the following queue is well-typed; rule [T-mf] types $\langle p, f \rangle$ by message type $\langle p, f \rangle$, while rule [T-mF] types $\langle\langle p, F \rangle\rangle$ by message type $\langle\langle p, F \rangle\rangle$. Rule [T-link] types a linking point $a[p](y).P$ by assuming that a provides a behaviour pattern defined in $\mathbf{G}$. For guiding P associated with p, [T-link] uses local type $\mathbf{T}$ generated by $\mathit{Transform}(\mathbf{G}, p)$ to type P acting on channel y. Rule [T-guide] states that $[P]_{\mathbf{T}}$ is well-typed by Δ if P is well-typed by Δ, and $\mathbf{T}$, gained by some $\mathbf{G}$, contains the type which types P acting on channel $s[p]$. Note that, by rule [link] (see Figure 4), $[P]_{\mathbf{T}}$ is created after linking and $\mathbf{T}$ is not changed after any reduction; thus $\mathbf{G}$ in rule [T-guide] comes from rule [T-link]. Rule [T-net] ensures the parallel composition of two networks if each of them is well-typed and

they do not share a common channel (i.e., $dom(\Delta_1) \cap dom(\Delta_2) = \emptyset$). The composed network exhibits the union of the session environments. Rule [T-new] types hiding (i.e., $(\mathsf{new}\ s)N$) when the session environment of networks under s, denoted by

$$\Delta\langle s\rangle \stackrel{\text{def}}{=} \{s'[p] : \mathbf{T} \mid s'[p] \in dom(\Delta), s' = s\} \cup \{s : \mathbf{q}\},$$

is coherent:

Definition 3 (Coherence). We say $\Delta\langle s\rangle$ is coherent if there exists $\mathbf{G}$ such that $pid(\mathbf{G}) = \{p \mid s'[p] \in dom(\Delta), s' = s\}$ and either (1) $\forall s[p] \in dom(\Delta\langle s\rangle)$ we have $Transform(\mathbf{G}, p)$ is equal to the type of $\Delta(s[p])$ after $s[p]$ absorbs all messages heading to it; or (2) there exists $\Delta' \subset \Delta$ such that $\forall s[p] \in dom(\Delta\langle s\rangle \setminus \Delta'\langle s\rangle)$ we have that $Transform(\mathbf{G}, p)$ is equal to the type of $\Delta(s[p])$ after $s[p]$ absorbs all messages heading to it, and $\Delta'\langle s\rangle$ is coherent.

Note that due to asynchrony, after a sender takes action, the type of the sender and its receiver may be temporarily incoherent if the sender has moved forward and the output is still in the global queue. Therefore, coherence holds only after a receiver has absorbed all messages heading to it.

As we aim to handle partial failure(s), either (1) no failures occurred such that there exists $\mathbf{G}$ defining interactions for every $s[p]$ in $\Delta\langle s\rangle$, or (2) a failure occurs such that the need-be-informed participants, who are in $\Delta'\langle s\rangle$, are handling that failure in a coherent way, and other unaffected ones, who are in $\Delta\langle s\rangle \setminus \Delta'\langle s\rangle$, still follow the behaviour defined in $\mathbf{G}$.

6 Properties

We prove that our typing discipline ensures the properties of *safety* and *progress*. The property of safety is defined by *subject reduction* and *communication safety*. Firstly we define $\Delta \rightharpoonup \Delta'$ as reductions of session environments. Intuitively, the reductions correspond closely to the operational semantics defined in Figure 4. Subject reduction states that a well-typed network (resp. coherent session environment) is always well-typed (resp. coherent) after reduction:

Theorem 1 (Subject Congruence and Reduction).

1. (subject congruence) $\Gamma \vdash N \triangleright \Delta$ and $N \equiv N'$ imply that $\Gamma \vdash N' \triangleright \Delta$.
2. (subject reduction) $\Gamma \vdash N \triangleright \Delta$ with Δ coherent and $N \to N'$ imply that $\Gamma \vdash N' \triangleright \Delta'$ such that $\Delta \rightharpoonup \Delta'$ or $\Delta \equiv \Delta'$ and Δ' is coherent.

According to the definition of communication safety in the works of multiparty session types [1,12], it is a corollary of Theorem 1. Note that, since our calculus is based on the work of Bettini *et al.* [1], global linearity-check is not needed. For convenience, we define here contexts on networks:

$$\mathscr{C} ::= [\,] \mid \mathscr{C} \parallel N \mid N \parallel \mathscr{C} \mid (\mathsf{new}\ s)\mathscr{C}$$

Corollary 1 (Communication Safety). Suppose $\Gamma \vdash N \triangleright \Delta$ and Δ is coherent. Let $N_1 = \mathscr{C}_1[s : q \cdot \langle p_2, p_1, \langle \vec{v}\rangle\rangle^F \cdot q']$ and $N_2 = \mathscr{C}_2[s : q \cdot \langle p_1, f\rangle \cdot q']$ and $N_3 = \mathscr{C}_3[s : q \cdot \langle\langle p_1, F\rangle\rangle \cdot q']$ and no messages in q is sending to p_1.

1. If $N = \mathscr{C}[\mathscr{E}[s[p_1]?(p_2,(\tilde{x})^F).P]_{\mathbf{T}}]$, then $N \equiv N_1$ or $N \rightarrow^* N_1$.
2. If $N = \mathscr{C}[\mathscr{E}[\mathsf{try}\{s[p_1] \otimes F'; P\}\mathsf{h}\{H\}]_{\mathbf{T}}]$ and $F' \subseteq F \neq \emptyset$, then either (a) $N \equiv N_2$ or $N \rightarrow^* N_2$ or (b) $N \equiv N_3$ or $N \rightarrow^* N_3$.
3. If $N = \mathscr{C}[\mathscr{E}[\mathsf{try}\{v\}\mathsf{h}\{H\}]_{\mathbf{T}}]$ and $f \in dom(H)$ and process $H(f)$ is acting on $s[p_1]$, then $N \not\equiv N_2$ and $N \not\rightarrow^* N_2$.

This corollary states that our system is *free from deadlock* and *starvation*: if there is a receiving action in N, then N is either structurally congruent to the network which contains the message for input, or N will reduce to such a network. We state that $[\mathsf{try}\{v\}\mathsf{h}\{H\}]_{\mathbf{T}}$ is safe to become idle by proving that no $f \in dom(H)$ is heading to it (Case 3).

Corollary 1 provides the means to prove that our system *never gets stuck and is free from orphan messages* (*property of progress*):

Theorem 2 (Progress). $\Gamma \vdash N \triangleright \Delta$ with Δ coherent and $N \rightarrow N'$ imply that N' is communication safe or $N' = \mathbf{0} \parallel s : \varepsilon$.

This theorem states that every interaction in a well-typed network is a safe interaction and reducible until the whole network terminates without any message left.

7 Related Works

Failure handling has been addressed in several process calculi and communication-centered programming languages. For instance, the conversation calculus [20] models exception handling in abstract service-based systems with message-passing based communication. It studies expressiveness and behaviour theory of bisimilarity rather than theory of types. Colombo and Pace [7] investigate several different process calculi for failure-recovery within long-running transactions. They give insight regarding the application of these failure-recovery formalisms in practice via comparing the design choices and formal notions of correctness properties. Both works do not provide a type system to statically type check local implementations.

Previous works for failure handling with type systems [3,4,5,13] extend the theory of session types to specify error handling under asynchronous interactions. These works do not capture handling of partial failures and the scenarios which exhibit the properties **P1** to **P4**. They may be able to encode multiple possible failures at the interaction level (**P1**), for example, by (i) explicitly using a labeled branching inside the failure handler, or (ii) piggybacking a label with the failure notification ("multiplexing"). However, (i) implies double communication and synchronisation (once for the failure notification, then for the branch) and (ii) implies that either the well-formedness constraints on the shape of interactions in handlers are needed or any participants related to any failure handling activity should be informed as a failure occurs in order to know how to proceed. Our approach is different since we do not have such constrains and we do not inform the unaffected participants. Moreover, while the termination of try-handle terms in those works demands an agreement of all participants, ours allows local try-handle terms to terminate since we have locally added synchronisation points by *transformation* (see Section 2.2). Our approach can encode the global types for exception-handling

proposed in the work by Capecchi *et al.* [3], which is the closest related work (and other related ones have similar try-handle syntax). The formal encoding can be found in the long version of this paper [9].

Collet and Van Roy [6] informally present a distributed programming model of Oz for asynchronous failure handling and focus on programming applications in a distributed manner. Jakšić and Padovani [14] study a type theory for error handling for copy-less messaging and memory sharing to prevent memory leaks/faults through typing of exchange heaps. Lanese *et al.* [11,15] formalise a feature which can dynamically install fault and compensation handlers at execution time in an orchestration programming style. They investigate the interplay between fault handling and the request-response pattern. In contrast, our framework statically defines the handlers for non-trivial failure handling, which can only be done with a global perspective.

8 Concluding Remarks

Protocol types enable the design of protocols in an intuitive manner, and statically type check multiple failure-handling processes in a transparent way. Our type discipline exhibits the desirable properties of **P1(alignment)**, **P2(precision)**, **P3(causality)**, and **P4(decentralisation)** for robust failure handling, and ensures fundamental properties of safety and progress. We are currently implementing the proposed framework and are extending it to support system-induced failures as opposed to application-specific ones focused on in this paper, in addition to parameterisation and dynamic multiroles.

References

1. L. Bettini, M. Coppo, L. D'Antoni, M. D. Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR'08*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
2. L. Caires and H. T. Vieira. Conversation types. In *ESOP '09*, volume 5502 of *LNCS*, pages 285–300. Springer, 2009.
3. S. Capecchi, E. Giachino, and N. Yoshida. Global escape in multiparty sessions. *MSCS*, 29:1–50, 2015.
4. M. Carbone, K. Honda, and N. Yoshida. Structured interactional exceptions in session types. In *CONCUR'08*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.
5. M. Carbone, N. Yoshida, and K. Honda. Asynchronous session types: Exceptions and multiparty interactions. In *SFM'09*, volume 5569 of *LNCS*, pages 187–212. Springer, 2009.
6. R. Collet and P. V. Roy. Failure handling in a network-transparent distributed programming language. In *Advanced Topics in Exception Handling Techniques*, pages 121–140, 2006.
7. C. Colombo and G. J. Pace. Recovery within long-running transactions. *ACM Comput. Surv.*, 45(3):28:1–28:35, July 2013.
8. P.-M. Deniélou and N. Yoshida. Dynamic multirole session types. In *POPL'11*, pages 435–446, 2011.
9. Technical Report. Long version of this paper. `https://github.com/Distributed-Systems-Programming-Group/paper/blob/master/2016/forte16_long_dsp.pdf`.
10. F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, Mar. 1999.

11. C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro. On the interplay between fault handling and request-response service invocations. In *Application of Concurrency to System Design, 2008. ACSD 2008. 8th International Conference on*, pages 190–198, June 2008.
12. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL '08*, pages 273–284. ACM, 2008.
13. R. Hu, R. Neykova, N. Yoshida, R. Demangeon, and K. Honda. Practical interruptible conversations - distributed dynamic verification with session types and Python. In *RV '13*, volume 8174 of *LNCS*, pages 130–148. Springer, 2013.
14. S. Jakšić and L. Padovani. Exception handling for copyless messaging. *Science of Computer Programming*, 84:22–51, 2014.
15. I. Lanese and F. Montesi. Error handling: From theory to practice. In *ISoLA'10*, volume 6416 of *LNCS*, pages 66–81. Springer, 2010.
16. I. Lanese, F. Montesi, and G. Zavattaro. Amending choreographies. In *WWV'13*, volume 123 of *EPTCS*, pages 34–48, 2013.
17. D. Mostrous. *Session Types in Concurrent Calculi: Higher-Order Processes and Objects*. PhD thesis, Imperial College London, 2009.
18. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
19. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413, 1994.
20. H. T. Vieira, L. Caires, and J. C. Seco. The conversation calculus: A model of service-oriented computation. In *ESOP '08*, volume 4960 of *LNCS*, pages 269–283. Springer, 2008.
21. N. Yoshida and V. T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.