A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems

Malte Viering^{1(✉)}, Tzu-Chun Chen¹, Patrick Eugster^{1,2,3}, Raymond Hu⁴,
and Lukasz Ziarek⁵

¹ Department of Computer Science, TU Darmstadt, Darmstadt, Germany
viering@dsp.tu-darmstadt.de

² Faculty of Informatics, Università della Svizzera italiana, Lugano, Switzerland

³ Department of Computer Science, Purdue University, West Lafayette, USA

⁴ Department of Computing, Imperial College London, London, UK

⁵ Department of Computer Science and Engineering, SUNY Buffalo, Buffalo, USA

Abstract. A key requirement for many distributed systems is to be resilient toward partial failures, allowing a system to progress despite the failure of some components. This makes programming of such systems daunting, particularly in regards to avoiding inconsistencies due to failures and asynchrony. This work introduces a formal model for crash failure handling in asynchronous distributed systems featuring a lightweight coordinator, modeled in the image of widely used systems such as ZooKeeper and Chubby. We develop a typing discipline based on multiparty session types for this model that supports the specification and static verification of multiparty protocols with explicit failure handling. We show that our type system ensures subject reduction and progress in the presence of failures. In other words, in a well-typed system even if some participants crash during execution, the system is guaranteed to progress in a consistent manner with the remaining participants.

1 Introduction

Distributed Programs, Partial Failures, and Coordination. Developing programs that execute across a set of physically remote, networked processes is challenging. The correct operation of a *distributed program* requires correctly designed protocols by which concurrent processes interact asynchronously, and correctly implemented processes according to their roles in the protocols. This becomes particularly challenging when distributed programs have to be resilient to *partial failures*, where some processes crashes while others remain operational. Partial failures affect both *safety* and *liveness* of applications. Asynchrony is the key

Financially supported by ERC grant FP7-617805 “LiVeSoft - Lightweight Verification of Software”, NSF grants CNS-1405614 and IIS-1617586, and EPSRC EP/K034413/1 and EP/K011715/1.

© The Author(s) 2018

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 799–826, 2018.

https://doi.org/10.1007/978-3-319-89884-1_28

issue, resulting in the inability to distinguish slow processes from failed ones. In general, this makes it impossible for processes to reach agreement, even when only a single process can crash [19].

In practice, such impasses are overcome by making appropriate assumptions for the considered infrastructure and applications. One common approach is to assume the presence of a highly available *coordination service* [26] – realized using a set of replicated processes large enough to survive common rates of process failures (e.g., 1 out of 3, 2 out of 5) – and delegating critical decisions to this service. While this *coordinator model* has been in widespread use for many years (cf. *consensus service* [22]), the advent of cloud computing has recently brought it further into the mainstream, via instances like Chubby [4] and ZooKeeper [26]. Such systems are used not only by end applications but also by a variety of frameworks and middleware systems across the layers of the protocol stack [11, 20, 31, 40].

Typing Disciplines for Distributed Programs. Typing disciplines for distributed programs is a promising and active research area towards addressing the challenges in the correct development of distributed programs. See Hüttel et al. [27] for a broad survey. *Session types* are one of the established typing disciplines for message passing systems. Originally developed in the π -calculus [23], these have been later successfully applied to a range of practical languages, e.g., Java [25, 41], Scala [39], Haskell [34, 38], and OCaml [28, 37]. *Multiparty session types* (MPSTs) [15, 24] generalize session types beyond two participants. In a nutshell, a standard MPST framework takes (1) a specification of the whole multiparty message protocol as a *global type*; from which (2) *local types*, describing the protocol from the perspective of each participant, are derived; these are in turn used to (3) statically *type check* the I/O actions of endpoint programs implementing the session participants. A well-typed system of session endpoint programs enjoys important safety and liveness properties, such as *no reception errors* (only expected messages are received) and *session progress*. A basic intuition behind MPSTs is that the design (i.e., restrictions) of the type language constitutes a class of distributed protocols for which these properties can be statically guaranteed by the type system.

Unfortunately, *no* MPST work supports protocols for asynchronous distributed programs dealing with *partial failures due to process crashes*, so the aforementioned properties no longer hold in such an event. Several MPST works have treated communication patterns based on *exception messages* (or *interrupts*) [6, 7, 16]. In these works, such messages may convey exceptional states in an *application* sense; from a protocol compliance perspective, however, these messages are the same as any other message communicated during a *normal* execution of the session. This is in contrast to *process* failures, which may invalidate already in-transit (*orphan*) messages, and where the task of agreeing on the concerted handling of a crash failure is itself prone to such failures.

Outside of session types and other type-based approaches, there have been a number of advances on verifying fault tolerant distributed protocols and applications (e.g., based on model checking [29], proof assistants [44]); however, little

work exists on providing direct compile-time support for *programming* such applications in the spirit of MPSTs.

Contributions and Challenges. This paper puts forward a new typing discipline for safe specification and implementation of distributed programs prone to process crash failures based on MPSTs. The following summarizes the key challenges and contributions.

Multiparty session calculus with coordination service. We develop an extended multiparty session calculus as a formal model of processes prone to crash failures in asynchronous message passing systems. Unlike standard session calculi that reflect only “minimal” networking infrastructures, our model introduces a practically-motivated *coordinator* artifact and explicit, asynchronous messages for run-time crash notifications and failure handling.

MPSTs with explicit failure handling. We introduce new global and local type constructs for *explicit failure handling*, designed for specifying protocols tolerating partial failures. Our type system carefully reworks many of the key elements in standard MPSTs to manage the intricacies of handling crash failures. These include the well-formedness of failure-prone global types, and the crucial *coherence* invariant on MPST typing environments to reflect the notion of system consistency in the presence of crash failures and the resulting errors. We show safety and progress for a well-typed MPST session despite potential failures.

To fit our model to practice, we introduce programming constructs similar to well-known and intuitive exception handling mechanisms, for handling concurrent and asynchronous process crash failures in sessions. These constructs serve to integrate user-level session control flow in endpoint processes and the underlying communications with the coordination service, used by the target applications of our work to outsource critical failure management decisions (see Fig. 1). It is important to note that the coordinator does *not* magically solve all problems. Key design challenges are to ensure that communication with it is fully asynchronous as in real-life, and that it is involved only in a “minimal” fashion. Thus we treat the coordinator as a first-class, asynchronous network artifact, as opposed to a convenient but impractical global “oracle” (cf. [6]), and our operational semantics of multiparty sessions remains primarily *choreographic* in the original spirit of distributed MPSTs, unlike works that resort to a centralized *orchestrator* to conduct all actions [5, 8]. As depicted in Fig. 1, application-specific communication does not involve the coordinator. Our model lends itself to common practical scenarios where processes monitor each other in a peer-based fashion to detect failures, and rely on a coordinator only to establish agreement on which processes have failed, and when.

A long version of this paper is available online [43]. The long version contains: full formal definitions, full proofs, and a prototype implementation in Scala.

Example. As a motivating example, Fig. 2 gives a global formal specification for a big data streaming task between a distributed file system (DFS) *dfs*, and two

Fig. 1. Coordinator model for asynchronous distributed systems. The coordinator is implemented by replicated processes (internals omitted).

$$\begin{aligned}
 [dfs]G = & \mathbf{t}(\mu t. \\
 & dfs \rightarrow w_1 \ l_{d_1}(S). dfs \rightarrow w_2 \ l_{d_2}(S). \\
 & w_1 \rightarrow dfs \ l_{r_1}(S'). w_2 \rightarrow dfs \ l_{r_2}(S'). t \\
) & \mathbf{h}(\\
 & \{w_1\} : \mu t'. dfs \rightarrow w_2 \ l'_{d_1}(S). \\
 & w_2 \rightarrow dfs \ l'_{r_1}(S'). t', \\
 & \{w_2\} : \dots, \{w_1, w_2\} : \text{end})
 \end{aligned}$$

Fig. 2. Global type for a big data streaming task with failure handling capabilities.

workers $w_{1,2}$. The DFS streams data to two workers, which process the data and write the result back. Most DFSs have built-in fault tolerance mechanisms [20], so we consider dfs to be *robust*, denoted by the annotation $[dfs]$; the workers, however, may individually fail. In the *try-handle* construct $\mathbf{t}(\dots)\mathbf{h}(\dots)$, the *try-block* $\mathbf{t}(\dots)$ gives the *normal* (i.e., failure-free) flow of the protocol, and $\mathbf{h}(\dots)$ contains the explicit *handlers* for potential crashes. In the try-block, the workers receive data from the DFS ($dfs \rightarrow w_i$), perform local computations, and send back the result ($w_i \rightarrow dfs$). If a worker crashes ($\{w_i\} : \dots$), the other worker will also take over the computation of the crashed worker, allowing the system to still produce a valid result. If both workers crash (by any interleaving of their concurrent crash events), the global type specifies that the DFS should safely terminate its role in the session.

We shall refer to this basic example, that focuses on the new failure handling constructs, in explanations in later sections. We also give many further examples throughout the following sections to illustrate the potential session errors due to failures exposed by our model, and how our framework resolves them to recover MPST safety and progress.

Roadmap. Section 2 describes the adopted system and failure model. Section 3 introduces global types for guiding failure handling. Section 4 introduces our process calculus with failure handling capabilities and a coordinator. Section 5 introduces local types, derived from global types by projection. Section 6 describes typing rules, and defines *coherence* of session environments with respect to end-point crashes. Section 7 states properties of our model. Section 8 discusses related work. Sect. 9 draws conclusions.

2 System and Failure Model

In distributed systems care is required to avoid partial failures affecting liveness (e.g., waiting on messages from crashed processes) or safety (e.g., when processes manage to communicate with some peers but not others before crashing) properties of applications. Based on the nature of the infrastructure and application,

appropriate *system and failure models* are chosen along with judiciously made assumptions to overcome such impasses in practice.

We pinpoint the key characteristics of our model, according to our practical motivations and standard distributed systems literature, that shape the design choices we make later for the process calculus and types. As it is common we augment our system with a *failure detector* (FD) to allow for distinguishing slow and failed processes. The advantage of the FD (1) in terms of reasoning is that it concentrates all assumptions to solve given problems and (2) implementation-wise it yields a single main module where time-outs are set and used.

Concretely we make the following assumptions on failures and the system:

- (1) **Crash-stop failures:** Application processes fail by crashing (halting), and do not recover.
- (2) **Asynchronous system:** Application processes and the network are asynchronous, meaning that there are no upper bounds on processes' relative speeds or message transmission delays.
- (3) **Reliable communication:** Messages transmitted between correct (i.e., non-failed) participants are eventually received.
- (4) **Robust coordinator:** The coordinator (coordination service) is permanently available.
- (5) **Asynchronous reliable failure detection:** Application processes have access to local FDs which eventually detect all failed peers and do not falsely suspect peers.

(1)–(3) are standard in literature on fault-tolerant distributed systems [19].

Note that processes can still recover but will not do so *within* sessions (or will not be re-considered for those). Other failure models, e.g., network partitions [21] or Byzantine failures [32], are subject of future work. The former are not tolerated by ZooKeeper et al., and the latter have often been argued to be a too generic failure model (e.g., [3]).

The assumption on the coordinator (4) implicitly means that the number of concomitant failures among the coordinator replicas is assumed to remain within a minority, and that failed replicas are replaced in time (to tolerate further failures). Without loss of validity, the coordinator internals can be treated as a blackbox. The final assumption (5) on failure detection is backed in practice by the concept of *program-controlled* crash [10], which consists in communicating decisions to disregard supposedly failed processes also to those processes, prompting them to reset themselves upon false suspicion. In practice systems can be configured to minimize the probability of such events, and by a “two-level” membership consisting in evicting processes from *individual* sessions (cf. recovery above) more quickly than from a system as a whole; several authors have also proposed network support to entirely avoid false suspicions (e.g., [33]).

These assumptions do not make handling of failures trivial, let alone mask them. For instance, the network can arbitrarily delay messages and thus reorder them with respect to their real sending times, and (so) different processes can detect failures at different points in time and in different orders.

(Basic type) $S ::= \text{bool} \mid \text{str} \mid \text{int}$
 (Global type) $G ::= p \rightarrow q\{l_i(S_i).G_i\}_{i \in I} \mid \mu t.G \mid t \mid \text{end} \mid \mathbf{t}(G_1)\mathbf{h}(H)^\kappa.G_2$
 (Handling env.) $H ::= F:G \mid H, H$ (Handler sig.) $F ::= \{p_i\}_{i \in I}$

Fig. 3. Syntax of global types with explicit handling of partial failures.

3 Global Types for Explicit Handling of Partial Failures

Based on the foundations of MPSTs, we develop *global types* to formalize specifications of distributed protocols with explicit handling of *partial failures due to role crashes*, simply referred to as *failures*. We present global types before introducing the process calculus to provide a high-level intuition of how failure handling works in our model.

The syntax of *global types* is depicted in Fig. 3. We use the following base notations: $p, q, \dots$ for *role* (i.e., participant) names; $l_1, l_2, \dots$ for message *labels*; and $t, t', \dots$ for type variables. *Base types* S may range over, bool, int , etc.

Global types are denoted by G . We first summarize the constructs from standard MPST [15, 24]. A *branch* type $p \rightarrow q\{l_i(S_i).G_i\}_{i \in I}$ means that p can send to q *one* of the messages of type S_k with label l_k , where k is a member of the non-empty index set I . The protocol then proceeds according to the continuation G_k . When I is a singleton, we may simply write $p \rightarrow q\ l(S).G$. We use t for type variables and take an equi-recursive view, i.e., $\mu t.G$ and its unfolding $[\mu t.G/t]$ are equivalent. We assume type variable occurrences are bound and guarded (e.g., $\mu t.t$ is not permitted). **end** is for termination.

We now introduce our extensions for partial failure handling. A *try-handle* $\mathbf{t}(G_1)\mathbf{h}(H)^\kappa.G_2$ describes a “failure-atomic” protocol unit: all *live* (i.e., non-crashed) roles will eventually reach a consistent protocol state, despite any concurrent and asynchronous role crashes. The try-block G_1 defines the *default* protocol flow, and H is a *handling environment*. Each element of H maps a *handler signature* F , that specifies a set of *failed* roles $\{p_i\}_{i \in I}$, to a *handler body* specified by a G . The handler body G specifies how the live roles should proceed given the failure of roles F . The protocol then proceeds (for live roles) according to the continuation G_2 after the default block G_1 or failure handling defined in H has been completed as appropriate.

To simplify later technical developments, we annotate each try-handle term in a given G by a unique $\kappa \in \mathbb{N}$ that lexically identifies the term within G . These annotations may be assigned mechanically. As a short hand, we refer to the try-block and handling environment of a particular try-handle by its annotation; e.g., we use κ to stand for $\mathbf{t}(G_1)\mathbf{h}(H)^\kappa$. In the running examples (e.g., Fig. 2), if there exists only one try-handle, we omit κ for simplicity.

Top-Level Global Types and Robust Roles. We use the term *top-level* global type to mean the source protocol specified by a user, following a typical top-down interpretation of MPST frameworks [15, 24]. We allow top-level global types to be optionally annotated $[\tilde{p}]G$, where $[\tilde{p}]$ specifies a set of *robust* roles—i.e., roles

that can be assumed to never fail. In practice, a participant may be robust if it is replicated or is made inherently fault tolerant by other means (e.g., the participant that represents the distributed file system in Fig. 2).

Well-Formedness. The first stage of validation in standard MPSTs is to check that the top-level global type satisfies the supporting criteria used to ensure the desired properties of the type system. We first list basic syntactic conditions which we assume on any given G : (i) each F is non-empty; (ii) a role in a F cannot occur in the corresponding handler body (a failed role cannot be involved in the handling of its own failure); and (iii) every occurrence of a non-robust role p must be contained within a, possibly outer, try-handle that has a handler signature $\{p\}$ (the protocol must be able to handle its potential failure). Lastly, to simplify the presentation without loss of generality, we impose that separate branch types *not* defined in the same default block or handler body must have disjoint label sets. This can be implicitly achieved by combining label names with try-handle annotations.

Assuming the above, we define *well-formedness* for our extended global types. We write $G' \in G$ to mean that G' syntactically occurs in G ($\in$ is reflexive); similarly for the variations $\kappa \in G$ and $\kappa \in \kappa'$. Recall κ is shorthand for $\mathbf{t}(G_1)\mathbf{h}(H)^\kappa$. We use a lookup function $outer_G(\kappa)$ for the set of all try-handles in G that enclose a given κ , including κ itself, defined by $outer_G(\kappa) = \{\kappa' \mid \kappa \in \kappa' \wedge \kappa' \in G\}$.

Definition 1 (Well-formedness). Let κ stand for $\mathbf{t}(G_1)\mathbf{h}(H)^\kappa$, and κ' for $\mathbf{t}(G'_1)\mathbf{h}(H')^{\kappa'}$. A global type G is *well-formed* if both of the following conditions hold. For all $\kappa \in G$:

1. $\forall F_1 \in \text{dom}(H). \forall F_2 \in \text{dom}(H). \exists \kappa' \in outer_G(\kappa) \text{ s.t. } F_1 \cup F_2 \in \text{dom}(H')$
2. $\nexists F \in \text{dom}(H). \exists \kappa' \in outer_G(\kappa). \exists F' \in \text{dom}(H') \text{ s.t. } \kappa' \neq \kappa \wedge F' \subseteq F$

The first condition asserts that for any two separate handler signatures of a handling environment of κ , there always exists a handler whose handler signature matches the union of their respective failure sets – this handler is either inside the handling environment of κ itself, or in the handling environment of an outer try-handle. This ensures that if roles are active in different handlers of the same try-handle then there is a handler whose signature corresponds to the union over the signatures of those different handlers. Example 2 together with Example 3 in Sect. 4 illustrate a case where this condition is needed. The second condition asserts that if the handling environment of a try-handle contains a handler for F , then there is no outer try-handle with a handler for F' such that $F' \subseteq F$. The reason for this condition is that in the case of *nested* try-handles, our communication model allows separate try-handles to start failure handling independently (the operational semantics will be detailed in the next section; see (TryHdl) in Fig. 6). The aim is to have the relevant roles eventually converge on performing the handling of the outermost try-handle, possibly by interrupting the handling of an inner try-handle. Consider the following example:

Example 1. $G = \mathbf{t}(\mathbf{t}(G')\mathbf{h}(\{p_1, p_2\} : G_1)^2)\mathbf{h}(\{p_1\} : G'_1)^1$ violates condition 2 because, when p_1 and p_2 both failed, the handler signature $\{p_1\}$ will still be triggered (i.e., the outer try-handle will eventually take over). It is not sensible to run G'_1 instead of G_1 (which is for the crashes of p_1 and p_2).

Fig. 4. Challenges under pure asynchronous interactions with a coordinator. Between time (1) and time (2), the task $\phi = (\kappa, \emptyset)$ is interrupted by the crash of P_a . Between time (3) and time (4), due to asynchrony and multiple crashes, P_c starts handling the crash of $\{P_a, P_d\}$ without handling the crash of $\{P_a\}$. Finally after (4) P_b and P_c finish their common task.

4 A Process Calculus for Coordinator-Based Failure Handling

Figure 4 depicts a scenario that can occur in practical asynchronous systems with coordinator-based failure handling through frameworks such as ZooKeeper (Sect. 2). Using this scenario, we first illustrate challenges, formally define our model, and then develop a safe type system.

The scenario corresponds to a global type of the form $\mathbf{t}(G)\mathbf{h}(\{P_a\} : G_a, \{P_a, P_d\} : G_{ad}, \dots)^\kappa$, with processes $P_{a..d}$ and a coordinator Ψ . We define a *task* to mean a unit of interactions, which includes failure handling behaviors. Initially all processes are collaborating on a task ϕ , which we label $(\kappa, \emptyset)$ (identifying the task context, and the set of failed processes). The shaded boxes signify which task each process is working on. Dotted arrows represent notifications between processes and Ψ related to task completion, and solid arrows for failure notifications from Ψ to processes. During the scenario, P_a first fails, then P_d fails: the execution proceeds through failure handling for $\{P_a\}$ and $\{P_a, P_d\}$.

- (I) When P_b reaches the end of its part in ϕ , the application has P_b notify Ψ . P_b then remains in the context of ϕ (the continuation of the box after notifying) in consideration of other non-robust participants still working on ϕ — P_b may yet need to handle their potential failure(s).

(Expression)	$e ::= v \mid x \mid e + e \mid -e \mid \dots$	(Channel)	$c ::= s[p] \mid y$
(Process)	$P ::= a[p](y).P \mid c : \eta$	(Level)	$\phi ::= (\kappa, F)$
(Statement)	$\eta ::= t(\eta)h(H)^\phi.\eta \mid \underline{0} \mid 0 \mid p! l(e).\eta$ $\quad \mid p?\{l_i(x_i).\eta_i\}_{i \in I} \mid X\langle e \rangle$ $\quad \mid \text{def } D \text{ in } \eta \mid \text{if } e \text{ then } \eta \text{ else } \eta$	(Declaration)	$D ::= X(x) = \eta$
(Application)	$N ::= P \mid N \mid N \mid s : h$	(Handling)	$H ::= F : \eta \mid H, H$
(Message)	$m ::= \langle p, q, l(v) \rangle \mid \langle p, \text{crash } F \rangle \mid dn$	(Queue)	$h ::= \emptyset \mid h \cdot m$
(System)	$S ::= \Psi \blacklozenge N \mid (\nu s)S \mid S \mid S$	(Done)	$dn ::= \langle p, q \rangle^\phi$
(Context)	$E ::= t(E)h(H)^\phi.\eta \mid \text{def } D \text{ in } E \mid []$	(Coordinator)	$\Psi ::= G : (F, d)$
		(Done Queue)	$d ::= \emptyset \mid d \cdot dn$

Fig. 5. Grammar for processes, applications, systems, and evaluation contexts.

- (II) The processes of synchronizing on the completion of a task or performing failure handling *are themselves subject to failures* that may arise concurrently. In Fig. 4, all processes reach the end of ϕ (i.e., four dotted arrows from ϕ), but P_a fails. Ψ determines this failure and it initiates failure handling at time (1), while *done* notifications for ϕ continue to arrive asynchronously at time (2). The failure handling for crash of P_a is itself interrupted by the second failure at time (3).
- (III) Ψ can receive notifications that are no longer relevant. For example, at time (2), Ψ has received all *done* notifications for ϕ , but the failure of P_a has already triggered failure handling from time (1).
- (IV) Due to multiple concurrent failures, interacting participants may end up in different tasks: around time (2), P_b and P_d are in task $\phi' = (\kappa, \{P_a\})$, whereas P_c is still in ϕ (and asynchronously sending or receiving messages with the others). Moreover, P_c never executes ϕ' because of delayed notifications, so it goes from ϕ directly to $(\kappa, \{P_a, P_d\})$.

Processes. Figure 5 defines the grammar of processes and (distributed) applications. Expressions $e, e_i, \dots$ can be values $v, v_i, \dots$, variables $x, x_i, \dots$, and standard operations. (Application) processes are denoted by $P, P_i, \dots$. An initialization $a[p](y).P$ agrees to play role p via shared name a and takes actions defined in P ; actions are executed on a session channel $c : \eta$, where c ranges over $s[p]$ (session name and role name) and session variables y ; η represents action statements.

A try-handle $t(\eta)h(H)^\phi$ attempts to execute the local action η , and can handle failures occurring therein as defined in the handling environment H , analogously to global types. H thus also maps a handler signature F to a handler body η defining how to handle F . Annotation $\phi = (\kappa, F)$ is composed of two elements: an identity κ of a *global* try-handle, and an indication of the *current* handler signature which can be empty. $F = \emptyset$ means that the default try-block is executing, whereas $F \neq \emptyset$ means that the handler body for F is executing. Term $\underline{0}$ only occurs in a try-handle during runtime. It denotes a *yielding* for a *notification* from a *coordinator* (introduced shortly).

Other statements are similar to those defined in [15, 24]. Term 0 represents an *idle* action. For convention, we omit 0 at the end of a statement. Action $p! l(e).\eta$ represents a sending action that sends p a label l with content e , then

it continues as η . Branching $p?\{l_i(x_i).\eta_i\}_{i \in I}$ represents a receiving action from p with several possible branches. When label l_k is selected, the transmitted value v is saved in x_k , and $\eta_k\{v/x_k\}$ continues. For convenience, when there is only one branch, the curly brackets are omitted, e.g., $c : p?l(x).P$ means there is only one branch $l(x)$. $X\langle e \rangle$ is for a statement variable with one parameter e , and $\text{def } D \text{ in } \eta$ is for recursion, where declaration D defines the recursive body that can be called in η . The conditional statement is standard.

The structure of processes ensures that failure handling is not interleaved between different sessions. However, we note that in standard MPSTs [15, 24], session interleaving must anyway be prohibited for the basic progress property. Since our aim will be to show progress, we disallow session interleaving within process bodies. Our model does allow parallel sessions at the top-level, whose actions may be concurrently interleaved during execution.

(Distributed) Systems. A (distributed) *system* in our programming framework is a composition of an application, which contains more than one process, and a coordinator (cf. Fig. 1). A system can be running within a private session $\mathbf{s}$, represented by $(\nu \mathbf{s})\mathcal{S}$, or $\mathcal{S} \mid \mathcal{S}'$ for systems running in different sessions independently and in parallel (i.e., no session interleaving). The job of the coordinator is to ensure that even in the presence of failures there is consensus on whether all participants in a given try-handle completed their local actions, or whether failures need to be handled, and which ones. We use $\Psi = G : (F, d)$ to denote a (robust) coordinator for the global type G , which stores in (F, d) the failures F that occurred in the application, and in d done notifications sent to the coordinator. The coordinator is denoted by ψ when viewed as a role.

A (distributed) *application*¹ is a process P , a parallel composition $N \mid N'$, or a global queue carrying messages $\mathbf{s} : h$. A global queue $\mathbf{s} : h$ carries a sequence of messages m , sent by participants in session $\mathbf{s}$. A message is either a regular message $\langle p, q, l(v) \rangle$ with label l and content v sent from p to q or a *notification*. A notification may contain the role of a coordinator. There are *done* and *failure* notifications with two kinds of done notifications dn used for coordination: $\langle p, \psi \rangle^\phi$ notifies ψ that p has finished its local actions of the try-handle ϕ ; $\langle \psi, p \rangle^\phi$ is sent from ψ to notify p that ψ has received all done notifications for the try-handle ϕ so that p shall end its current try-handle and move to its next task. For example, in Fig. 4 at time (4) the coordinator will inform P_b and P_c via $\langle \psi, P_b \rangle^{\langle \kappa, \{P_a, P_d\} \rangle} . \langle \psi, P_c \rangle^{\langle \kappa, \{P_a, P_d\} \rangle}$ that they can finish the try-handle $(\kappa, \{P_a, P_d\})$. Note that the appearance of $\langle \psi, p \rangle^\phi$ implies that the coordinator has been informed that all participants in ϕ have completed their local actions. We define two kinds of *failure* notifications: $\langle \psi, \text{crash } F \rangle$ notifies ψ that F occurred, e.g., $\{q\}$ means q has failed; $\langle p, \text{crash } F \rangle$ is sent from ψ to notify p about the failure F for possible handling. We write $\langle \tilde{p}, \text{crash } F \rangle$, where $\tilde{p} = p_1, \dots, p_n$ short for $\langle p_1, \text{crash } F \rangle \cdot \dots \cdot \langle p_n, \text{crash } F \rangle$; similarly for $\langle \psi, \tilde{p} \rangle^\phi$.

¹ Other works use the term *network* which is the reason why we use N instead of, e.g., A . We call it application to avoid confusion with the physical network which interconnects all processes as well as the coordinator.

$$\begin{array}{l}
a[p_1](y_1).P_1 \mid \dots \mid a[p_n](y_n).P_n \rightarrow \\
(\nu s)(G : (\emptyset, \emptyset) \blacklozenge P_1\{s[p_1]/y_1\} \mid \dots \mid P_n\{s[p_n]/y_n\} \mid s : \emptyset) \quad a : G \quad \textbf{(Link)} \\
s[p] : E[q! l(e).\eta] \mid s : h \rightarrow s[p] : E[\eta] \mid s : h \cdot \langle p, q, l(v) \rangle \quad e \Downarrow v \quad \textbf{(Snd)} \\
s[p] : E[q?\{l_i(x_i).\eta_i\}_{i \in I}] \mid s : \langle q, p, l_k(v_k) \rangle \cdot h \rightarrow \\
s[p] : E[\eta_k\{v_k/x_k\}] \mid s : h \quad k \in I \quad \textbf{(Rcv)} \\
s[p] : E[\text{def } X(x) = \eta \text{ in } X\langle e \rangle] \rightarrow s[p] : E[\text{def } X(x) = \eta \text{ in } \eta\{v/x\}] \quad e \Downarrow v \quad \textbf{(Rec)} \\
\frac{N_1 \equiv N_3 \rightarrow N_4 \equiv N_2}{N_1 \rightarrow N_2} \quad \frac{N_1 \rightarrow N_2}{N_1 \mid N \rightarrow N_2 \mid N} \quad \textbf{(Str, Par)} \\
\frac{N_1 \rightarrow N_2}{\psi \blacklozenge N_1 \rightarrow \psi \blacklozenge N_2} \quad \frac{\mathcal{S} \rightarrow \mathcal{S}'}{(\nu s)S \rightarrow (\nu s)S'} \quad \textbf{(Sys, New)} \\
N \mid s : h \rightarrow N \setminus s[p] : \eta \mid s : \text{remove}(h, p) \cdot \langle \psi, \text{crash } \{p\} \rangle \\
s[p] : \eta \text{ non-robust} \quad \textbf{(Crash)}
\end{array}$$

Fig. 6. Operational semantics of distributed applications, for local actions.

Following the tradition of other MPST works the global queue provides an abstraction for multiple FIFO queues, each queue being between two endpoints (cf. TCP) with no global ordering. Therefore $m_i \cdot m_j$ can be permuted to $m_j \cdot m_i$ in the global queue if the sender or the receiver differ. For example the following messages are permutable: $\langle p, q, l(v) \rangle \cdot \langle p, q', l(v) \rangle$ if $q \neq q'$ and $\langle p, q, l(v) \rangle \cdot \langle \psi, p \rangle^\phi$ and $\langle p, q, l(v) \rangle \cdot \langle q, \text{crash } F \rangle$. But $\langle \psi, p \rangle^\phi \cdot \langle p, \text{crash } F \rangle$ is not permutable, both have the same sender and receiver (ψ is the sender of $\langle p, \text{crash } F \rangle$).

Basic Dynamic Semantics for Applications. Figure 6 shows the operational semantics of applications. We use evaluation contexts as defined in Fig. 5. Context E is either a hole $[]$, a default context $t(E)h(H)^\phi.\eta$, or a recursion context $\text{def } D \text{ in } E$. We write $E[\eta]$ to denote the action statement obtained by filling the hole in $E[\cdot]$ with η .

Rule **(Link)** says that (local) processes who agree on shared name a , obeying to some protocol (global type), playing certain roles p_i represented by $a[p_i](y_i).P$, together will start a private session s ; this will result in replacing every variable y_i in P_i and, at the same time, creating a new global queue $s : \emptyset$, and appointing a coordinator $G : (\emptyset, \emptyset)$, which is novel in our work.

Rule **(Snd)** in Fig. 6 reduces a sending action $q! l(e)$ by emitting a message $\langle p, q, l(v) \rangle$ to the global queue $s : h$. Rule **(Rcv)** reduces a receiving action if the message arriving at its end is sent from the expected sender with an expected label. Rule **(Rec)** is for recursion. When the recursive body, defined inside η , is called by $X\langle e \rangle$ where e is evaluated to v , it reduces to the statement $\eta\{v/x\}$ which will again implement the recursive body. Rule **(Str)** says that processes which are structurally congruent have the same reduction. Processes, applications, and systems are considered modulo structural congruence, denoted by $\equiv$, along with α -renaming. Rule **(Par)** and **(Str)** together state that a parallel composition has a reduction if its sub-application can reduce. Rule **(Sys)** states that a system has a reduction if its application has a reduction, and **(New)**

says a reduction can proceed under a session. Rule (**Crash**) states that a process on channel $s[p]$ can fail at any point in time. (**Crash**) also adds a notification $\langle\psi, \text{crash } F\rangle$ which is sent to ψ (the coordinator). This is an abstraction for the failure detector described in Sect. 2 (5), the notification $\langle\psi, \text{crash } F\rangle$ is the first such notification issued by a participant based on its local failure detector. Adding the notification into the global queue instead of making the coordinator immediately aware of it models that failures are only detected eventually. Note that a failure is not annotated with a level because failures transcend all levels, and asynchrony makes it impossible to identify “where” exactly they occurred. As a failure is permanent it can affect multiple try-handles. The (**Crash**) rule does not apply to participants which are robust, i.e., that conceptually cannot fail (e.g., *dfs* in Fig. 2). Rule (**Crash**) removes channel $s[p]$ (the failed process) from application N , and removes messages and notifications delivered from, or heading to, the failed p by function $\text{remove}(h, p)$. Function $\text{remove}(h, p)$ returns a new queue after removing all regular messages and notifications that contain p , e.g., let $h = \langle p_2, p_1, l(v) \rangle \cdot \langle p_3, p_2, l'(v') \rangle \cdot \langle p_3, p_4, l'(v') \rangle \cdot \langle p_2, \psi \rangle^\phi \cdot \langle p_2, \text{crash } \{p_3\} \rangle \cdot \langle \psi, p_2 \rangle^\phi$ then $\text{remove}(h, p_2) = \langle p_3, p_4, l'(v') \rangle$. Messages are removed to model that in a real system send/receive does *not* constitute an atomic action.

Handling at Processes. Failure handling, defined in Fig. 7, is based on the observations that (i) a process that fails stays down, and (ii) multiple processes can fail. As a consequence a failure can trigger multiple failure handlers either because these handlers are in different (subsequent) try-handles or because of additional failures. Therefore a process needs to retain the information of *who* failed. For simplicity we do not model state at processes, but instead processes read but do not remove failure notifications from the global queue. We define $Fset(h, p)$ to return the union of failures for which there are notifications heading to p , i.e., $\langle p, \text{crash } F \rangle$, issued by the coordinator in queue h up to the first done notification heading to p :

Definition 2 (Union of Existing Failures $Fset(h, p)$)

$$Fset(\emptyset, p) = \emptyset \quad Fset(h, p) = \begin{cases} F \cup Fset(h', p) & \text{if } h = \langle p, \text{crash } F \rangle \cdot h' \\ \emptyset & \text{if } h = \langle \psi, p \rangle^\phi \cdot h' \\ Fset(h', p) & \text{otherwise} \end{cases}$$

In short, if the global queue is $\emptyset$, then naturally there are no failure notifications. If the global queue contains a failure notification sent from the coordinator, say $\langle p, \text{crash } F \rangle$, we collect the failure. If the global queue contains done notification $\langle \psi, p \rangle^\phi$ sent from the coordinator then *all* participants in ϕ have finished their local actions, which implies that the try-handle ϕ can be completed. Our failure handling semantics, (**TryHdl**), allows a try-handle $\phi = (\kappa, F)$ to handle different failures or sets of failures by allowing a try-handle to switch between different handlers. F thus denotes the current set of handled failures. For simplicity we refer to this as the *current(ly handled) failure set*. This is a slight abuse of terminology, done for brevity, as obviously failures are only detected with a

$$\begin{array}{c}
\frac{F' = \cup\{A \mid A \in \text{dom}(\mathbf{H}) \wedge F \subset A \subseteq \text{Fset}(h, p)\} \quad F': \eta' \in \mathbf{H}}{\text{s}[p] : E[\text{t}(\eta)\mathbf{h}(\mathbf{H})^{(\kappa, F')}. \eta''] \mid \text{s} : h \rightarrow \text{s}[p] : E[\text{t}(\eta')\mathbf{h}(\mathbf{H})^{(\kappa, F')}. \eta''] \mid \text{s} : h} \quad (\text{TryHdl}) \\
\\
\text{s}[p] : E[\text{t}(0)\mathbf{h}(\mathbf{H})^\phi. \eta] \mid \text{s} : h \rightarrow \text{s}[p] : E[\text{t}(\underline{0})\mathbf{h}(\mathbf{H})^\phi. \eta] \mid \text{s} : h \cdot \langle p, \psi \rangle^\phi \quad (\text{SndDone}) \\
\\
\frac{\langle \psi, p \rangle^\phi \in h}{\text{s}[p] : E[\text{t}(\underline{0})\mathbf{h}(\mathbf{H})^\phi. \eta] \mid \text{s} : h \rightarrow \text{s}[p] : E[\eta] \mid \text{s} : h \setminus \{\langle \psi, p \rangle^\phi\}} \quad (\text{RcvDone}) \\
\\
\text{s}[p] : E[\eta] \mid \text{s} : \langle q, p, l(v) \rangle \cdot h \rightarrow \text{s}[p] : E[\eta] \mid \text{s} : h \quad l \notin \text{labels}(E[\eta]) \quad (\text{Cln}) \\
\\
\frac{\langle \psi, p \rangle^\phi \in h \quad \phi \notin E[\eta]}{\text{s}[p] : E[\eta] \mid \text{s} : h \rightarrow \text{s}[p] : E[\eta] \mid \text{s} : h \setminus \langle \psi, p \rangle^\phi} \quad (\text{ClnDone})
\end{array}$$

Fig. 7. Operational semantics of distributed applications, for endpoint handling.

certain lag. The handling strategy for a process is to handle the—currently—largest set of failed processes that this process has been informed of and is able to handle. This largest set is calculated by $\cup\{A \mid A \in \text{dom}(\mathbf{H}) \wedge F \subset A \subseteq \text{Fset}(h, p)\}$, that selects all failure sets which are larger than the current one ($A \in \text{dom}(\mathbf{H}) \wedge F \subset A$) if they are also triggered by known failures ($A \subseteq \text{Fset}(h, p)$). Condition $F' : \eta' \in \mathbf{H}$ in **(TryHdl)** ensures that there exists a handler for F' . The following example shows how **(TryHdl)** is applied to switch handlers.

Example 2. Take h such that $\text{Fset}(h, p) = \{p_1\}$ and $\mathbf{H} = \{p_1\} : \eta_1, \{p_2\} : \eta_2, \{p_1, p_2\} : \eta_{12}$ in process $P = \text{s}[p] : \text{t}(\eta_1)\mathbf{h}(\mathbf{H})^{(\kappa, \{p_1\})}$, which indicates that P is handling failure $\{p_1\}$. Assume now one more failure occurs and results in a new queue h' such that $\text{Fset}(h', p) = \{p_1, p_2\}$. By **(TryHdl)**, the process acting at $\text{s}[p]$ is handling the failure set $\{p_1, p_2\}$ such that $P = \text{s}[p] : \text{t}(\eta_{12})\mathbf{h}(\mathbf{H})^{(\kappa, \{p_1, p_2\})}$ (also notice the η_{12} inside the try-block). A switch to only handling $\{p_2\}$ does not make sense, since, e.g., η_2 can contain p_1 . Figure 2 shows a case where the handling strategy differs according to the number of failures.

In Sect. 3 we formally define well-formedness conditions, which guarantee that if there exist two handlers for two different handler signatures in a try-handle, then a handler exists for their union. The following example demonstrates why such a guarantee is needed.

Example 3. Assume a slightly different P compared to the previous examples (no handler for the union of failures): $P = \text{s}[p] : E[\text{t}(\eta)\mathbf{h}(\mathbf{H})^{(\kappa, \emptyset)}]$ with $\mathbf{H} = \{p_1\} : \eta_1, \{p_2\} : \eta_2$. Assume also that $\text{Fset}(h, p) = \{p_1, p_2\}$. Here **(TryHdl)** will not apply since there is no failure handling for $\{p_1, p_2\}$ in P . If we would allow a handler for either $\{p_1\}$ or $\{p_2\}$ to be triggered we would have no guarantee that other participants involved in this try-handle will all select the same failure set. Even with a deterministic selection, i.e., all participants in that try-handle selecting the same handling activity, there needs to be a handler with handler signature $= \{p_1, p_2\}$ since it is possible that p_1 is involved in η_2 . Therefore the type system will ensure that there is a handler for $\{p_1, p_2\}$ either at this level or at an outer level.

(I) explains that a process finishing its default action (P_b) cannot leave its current try-handle $(\kappa, \emptyset)$ immediately because other participants may fail (P_a failed). Below Eq. 1 also shows this issue from the perspective of semantics:

$$\begin{aligned} s[p] : \mathbf{t}(0)h(F : q!l(10).q?l'(x))^{(\kappa, \emptyset)}. \eta' \mid s[q] : \mathbf{t}(p?l(x').p!l'(x' + 10))h(H)^{(\kappa, F)}. \eta'' \\ \mid s : \langle q, \text{crash } F \rangle \cdot \langle p, \text{crash } F \rangle \cdot h \end{aligned} \quad (1)$$

In Eq. 1 the process acting on $s[p]$ ended its try-handle (i.e., the action is 0 in the try-block), and if $s[p]$ finishes its try-handle the participant acting on $s[q]$ which started handling F would be stuck.

To solve the issue, we use (**SndDone**) and (**RcvDone**) for completing a local try-handle with the help of a coordinator. The rule (**SndDone**) sends out a done notification $\langle p, \psi \rangle^\phi$ if the current action in ϕ is 0 and sets the action to $\underline{0}$, indicating that a done notification from the coordinator is needed for ending the try-handle.

Assume process on channel $s[p]$ finished its local actions in the try-block (i.e., as in Eq. 1 above), then by (**SndDone**), we have

$$\begin{aligned} (1) \rightarrow s : \langle q, \text{crash } F \rangle \cdot \langle p, \text{crash } F \rangle \cdot \langle p, \psi \rangle^{(\kappa, \emptyset)} \cdot h \mid \\ s[p] : \mathbf{t}(\underline{0})h(F : q!l(10).q?l'(x))^{(\kappa, \emptyset)}. \eta' \mid s[q] : \mathbf{t}(p?l(x').p!l'(x' + 10))h(H)^{(\kappa, F)}. \eta'' \end{aligned}$$

where notification $\langle p, \psi \rangle^{(\kappa, \emptyset)}$ is added to inform the coordinator. Now the process on channel $s[p]$ can still handle failures defined in its handling environment. This is similar to the case described in (II).

Rule (**RcvDone**) is the counterpart of (**SndDone**). Once a process receives a done notification for ϕ from the coordinator it can finish the try-handle ϕ and reduces to the continuation η . Consider Eq. 2 below, which is similar to Eq. 1 but we take a case where the try-handle can be reduced with (**RcvDone**). In Eq. 2 (**SndDone**) is applied:

$$\begin{aligned} s[p] : \mathbf{t}(\underline{0})h(F : q!l(10).q?l'(x))^{(\kappa, \emptyset)}. \eta' \mid \\ s[q] : \mathbf{t}(\underline{0})h(F : p?l(x').p!l'(x' + 10))^{(\kappa, \emptyset)}. \eta'' \mid s : h \end{aligned} \quad (2)$$

With $h = \langle \psi, q \rangle^{(\kappa, \emptyset)} \cdot \langle \psi, p \rangle^{(\kappa, \emptyset)} \cdot \langle q, \text{crash } F \rangle \cdot \langle p, \text{crash } F \rangle$ both processes can apply (**RcvDone**) and safely terminate the try-handle $(\kappa, \emptyset)$. Note that $Fset(h, p) = Fset(h, q) = \emptyset$ (by Definition 2), i.e., rule (**TryHdl**) can not be applied since a done notification suppresses the failure notification. Thus Eq. 2 will reduce to:

$$(2) \rightarrow^* s[p] : \eta' \mid s[q] : \eta'' \mid s : \langle q, \text{crash } F \rangle \cdot \langle p, \text{crash } F \rangle$$

It is possible that η' or η'' have handlers for F . Note that once a queue contains $\langle \psi, p \rangle^{(\kappa, \emptyset)}$, all non-failed process in the try-handle $(\kappa, \emptyset)$ have sent done notifications to ψ (i.e. applied rule (**SndDone**)). The coordinator which will be introduced shortly ensures this.

$$\begin{array}{c}
\frac{\tilde{p} = \text{roles}(G) \setminus F' \quad F' = F \cup \{p\} \quad m = \langle \tilde{p}, \text{crash } \{p\} \rangle}{G : (F, d) \blacklozenge N \mid s : \langle \psi, \text{crash } \{p\} \rangle \cdot h \rightarrow G : (F', d) \blacklozenge N \mid s : h \cdot m} \text{ (F)} \\
\\
\frac{d' = d \cdot \langle p, \psi \rangle^\phi}{G : (F, d) \blacklozenge s : \langle p, \psi \rangle^\phi \cdot h \rightarrow G : (F, d') \blacklozenge s : h} \text{ (CollectDone)} \\
\\
\frac{\text{roles}(d, \phi) \supseteq \text{roles}(G, \phi) \setminus F \quad \forall F' \in \text{hdl}(G, \phi). (F' \not\subseteq F)}{G : (F, d) \blacklozenge s : h \rightarrow G : (F, \text{remove}(d, \phi)) \blacklozenge s : h \cdot \langle \psi, \text{roles}(G, \phi) \setminus F \rangle^\phi} \text{ (IssueDone)}
\end{array}$$

Fig. 8. Operational semantics for the coordinator.

Rule **(Cln)** removes a normal message from the queue if the label in the message does not exist in the target process, which can happen when a failure handler was triggered. The function $\text{labels}(\eta)$ returns all labels of receiving actions in η which are able to receive messages now or possible later. This removal based on the syntactic process is safe because in a global type separate branch types *not* defined in the same default block or handler body must have disjoint sets of labels (c.f., Sect. 3). Let $\phi \in P$ if try-handle ϕ appears inside P . Rule **(ClnDone)** removes a done notification of ϕ from the queue if no try-handle ϕ exists, which can happen in case of nesting when a handler of an outer try-handle is triggered.

Handling at Coordinator. Figure 8 defines the semantics of the coordinator. We firstly give the auxiliary definition of $\text{roles}(G)$ which gives the set of *all* roles appearing in G .

In rule **(F)**, F represents the failures that the coordinator is aware of. This rule states that the coordinator collects and removes a failure notification $\langle \psi, \text{crash } p \rangle$ heading to it, retains this notification by $G : (F', d)$, $F' = F \cup \{p\}$, and issues failure notifications to all non-failed participants.

Rules **(CollectDone, IssueDone)**, in short inform all participants in $\phi = (\kappa, F)$ to finish their try-handle ϕ if the coordinator has received sufficient done notifications of ϕ and did not send out failure notifications that interrupt the task (κ, F) (e.g. see (III)). Rule **(CollectDone)** collects done notifications, i.e., $\langle p, \psi \rangle^\phi$, from the queue and retains these notification; they are used in **(IssueDone)**. For introducing **(IssueDone)**, we first introduce $\text{hdl}(G, (\kappa, F))$ to return a set of handler signatures which can be triggered with respect to the current handler:

Definition 3. $\text{hdl}(G, (\kappa, F)) = \text{dom}(H) \setminus \mathcal{P}(F)$ if $\mathbf{t}(G_0)\mathbf{h}(H)^\kappa \in G$ where $\mathcal{P}(F)$ represents a powerset of F .

Also, we abuse the function roles to collect the non-coordinator roles of ϕ in d , written $\text{roles}(d, \phi)$; similarly, we write $\text{roles}(G, \phi)$ where $\phi = (\kappa, F)$ to collect the roles appearing in the handler body F in the try-handle of κ in G . Remember that d only contains done notifications sent by participants.

Rule **(IssueDone)** is applied for some ϕ when conditions $\forall F' \in \text{hdl}(G, \phi). (F' \not\subseteq F)$ and $\text{roles}(d, \phi) \supseteq \text{roles}(G, \phi) \setminus F$ are both satisfied, where F contains all failures the coordinator is aware of. Intuitively, these two conditions ensure that (1) the coordinator only issues done notifications to the participants

in the try-handle ϕ if it did not send failure notifications which will trigger a handler of the try-handle ϕ ; (2) the coordinator has received all done notifications from all non-failed participants of ϕ . We further explain both conditions in the following examples, starting from condition $\forall F' \in hdl(G, \phi). (F' \not\subseteq F)$, which ensures no handler in ϕ can be triggered based on the failure notifications F sent out by the coordinator.

Example 4. Assume a process playing role p_i is $P_i = s[p_i] : t(\eta_i)h(H_i)^{\phi_i}$. Where $i \in \{1, 2, 3\}$ and $H_i = \{p_2\} : \eta_{i2}, \{p_3\} : \eta_{i3}, \{p_2, p_3\} : \eta_{i23}$ and the coordinator is $G : (\{p_2, p_3\}, d)$ where $t(\dots)h(H)^\kappa \in G$ and $dom(H) = dom(H_i)$ for any $i \in \{1, 2, 3\}$ and $d = \langle p_1, \psi \rangle^{(\kappa, \{p_2\})} . \langle p_1, \psi \rangle^{(\kappa, \{p_2, p_3\})} . d'$. For any ϕ in d , the coordinator checks if it has issued any failure notification that can possibly trigger a new handler of ϕ :

1. For $\phi = (\kappa, \{p_2\})$ the coordinator issued failure notifications that can interrupt a handler since

$$hdl(G, (\kappa, \{p_2\})) = dom(H) \setminus \mathcal{P}(\{p_2\}) = \{\{p_3\}, \{p_2, p_3\}\}$$

and $\{p_2, p_3\} \subseteq \{p_2, p_3\}$. That means the failure notifications issued by the coordinator, i.e., $\{p_2, p_3\}$, can trigger the handler with signature $\{p_2, p_3\}$. Thus the coordinator will not issue done notifications for $\phi = (\kappa, \{p_2\})$. A similar case is visualized in Fig. 4 at time (2).

2. For $\phi = (\kappa, \{p_2, p_3\})$ the coordinator did not issue failure notifications that can interrupt a handler since

$$hdl(G, (\kappa, \{p_2, p_3\})) = dom(H) \setminus \mathcal{P}(\{p_2, p_3\}) = \emptyset$$

so that $\forall F' \in hdl(G, (\kappa, \{p_2, p_3\})). (F' \not\subseteq \{p_2, p_3\})$ is true. The coordinator will issue done notifications for $\phi = (\kappa, \{p_2, p_3\})$.

Another condition $roles(d, \phi) \supseteq roles(G, \phi) \setminus F$ states that only when the coordinator sees sufficient done notifications (in d) for ϕ , it issues done notifications to *all* non-failed participants in ϕ , i.e., $\langle \psi, roles(G, \phi) \setminus F \rangle^\phi$. Recall that $roles(d, \phi)$ returns all roles which have sent a done notification for the handling of ϕ and $roles(G, \phi)$ returns all roles involving in the handling of ϕ . Intuitively one might expect the condition to be $roles(d, \phi) = roles(G, \phi)$; the following example shows why this would be wrong.

Example 5. Consider a process P acting on channel $s[p]$ and $\{q\} \notin dom(H)$:

$$P = s[p] : t(\dots t(\dots)h(\{q\} : \eta, H')^{\phi'} . \eta')h(H)^\phi$$

Assume P has already reduced to:

$$P = s[p] : t(\underline{0})h(H)^\phi$$

We show why $roles(d, \phi) \supseteq roles(G, \phi) \setminus F$ is necessary. We start with the simple cases and then move to the more involving ones.

$$\begin{aligned}
T &::= p!\{l_i(S_i).T_i\}_{i \in I} \mid p?\{l_i(S_i).T_i\}_{i \in I} \mid t \mid \mu t.T \mid \text{end} \mid \underline{\text{end}} \mid t(T)\mathbf{h}(\mathcal{H})^\phi.T \\
\mathcal{H} &::= F:T \mid \mathcal{H}, \mathcal{H}
\end{aligned}$$

Fig. 9. The grammar of local types.

- (a) Assume q did not fail, the coordinator is $G : (\emptyset, d)$, and all roles in ϕ issued a done notification. Then $\text{roles}(d, \phi) = \text{roles}(G, \phi)$ and $F = \emptyset$.
- (b) Assume q failed in the try-handle ϕ' , the coordinator is $G : (\{q\}, d)$, and all roles except q in ϕ issued a done notification. $\text{roles}(d, \phi) \neq \text{roles}(G, \phi)$ however $\text{roles}(d, \phi) = \text{roles}(G, \phi) \setminus \{q\}$. Cases like this are the reason why **(IssueDone)** only requires done notifications from non-failed roles.
- (c) Assume q failed after it has issued a done notification for ϕ (i.e., q finished try-handle ϕ') and the coordinator collected it (by **(CollectDone)**), so we have $G : (\{q\}, d)$ and $q \in \text{roles}(d, \phi)$. Then $\text{roles}(d, \phi) \supset \text{roles}(G, \phi) \setminus \{q\}$. i.e. **(IssueDone)** needs to consider done notifications from failed roles.

Thus rule **(IssueDone)** has the condition $\text{roles}(d, \phi) \supseteq \text{roles}(G, \phi) \setminus F$ because of cases like (b) and (c).

The interplay between issuing of done notification (**(IssueDone)**) and issuing of failure notifications (**(F)**) is non-trivial. The following proposition clarifies that the participants in the same try-handle ϕ will never get confused with handling failures or completing the try-handle ϕ .

Proposition 1. *Given $s : h$ with $h = h' \cdot \langle \psi, p \rangle^\phi \cdot h''$ and $F\text{set}(h, p) \neq \emptyset$, the rule **(TryHdl)** is not applicable for the try-handle ϕ at the process playing role p .*

5 Local Types

Figure 9 defines local types for typing behaviors of endpoint processes with failure handling. Type $p!$ is the primitive for a sending type, and $p?$ is the primitive for a receiving type, derived from global type $p \rightarrow q\{l_i(S_i).G_i\}_{i \in I}$ by projection. Others correspond straightforwardly to process terms. Note that type $\underline{\text{end}}$ only appears in *runtime* type checking. Below we define $G \upharpoonright p$ to project a global type G on p , thus generating p 's local type.

Definition 4 (Projection). Consider a well-formed top-level global type $[\tilde{q}]G$. Then $G \upharpoonright p$ is defined as follows:

$$\begin{aligned}
(1) \quad G \upharpoonright p & \text{ where } G = \mathbf{t}(G_0)\mathbf{h}(F_1 : G_1, \dots, F_n : G_n)^\kappa.G' = \\
& \begin{cases} \mathbf{t}(G_0 \upharpoonright p)\mathbf{h}(F_1 : G_1 \upharpoonright p, \dots, F_n : G_n \upharpoonright p)^{(\kappa, \emptyset)}.G' \upharpoonright p & \text{if } p \in \text{roles}(G) \\ G' \upharpoonright p & \text{otherwise} \end{cases} \\
(2) \quad p_1 \rightarrow p_2\{l_i(S_i).G_i\}_{i \in I} \upharpoonright p & = \begin{cases} p_2!\{l_i(S_i).G_i \upharpoonright p\}_{i \in I} & \text{if } p = p_1 \\ p_1?\{l_i(S_i).G_i \upharpoonright p\}_{i \in I} & \text{if } p = p_2 \\ G_1 \upharpoonright p & \text{if } \forall i, j \in I. G_i \upharpoonright p = G_j \upharpoonright p \end{cases}
\end{aligned}$$

- (3) $(\mu t.G)\downarrow p = \mu t.(G\downarrow p)$ if $\exists t(G')h(H) \in G$ and $G\downarrow p \neq t'$ for any t'
 (4) $t\downarrow p = t$ (5) $\text{end}\downarrow p = \text{end}$

Otherwise it is undefined.

The main rule is (1): if p appears somewhere in the target try-handle global type then the endpoint type has a try-handle annotated with κ and the default logic (i.e., $F = \emptyset$). Note that even if $G_0\downarrow p = \text{end}$ the endpoint still gets such a try-handle because it needs to be ready for (possible) failure handling; if p does not appear anywhere in the target try-handle global type, then the projection skips to the continuation.

Rule (2) produces local types for interaction endpoints. If the endpoint is a sender (i.e., $p = p_1$), then its local type abstracts that it will send something from one of the possible internal choices defined in $\{l_i(S_i)\}_{i \in I}$ to p_2 , then continue as $G_k\downarrow p$, gained from the projection, if $k \in I$ is chosen. If the endpoint is a receiver (i.e., $p = p_2$), then its local type abstracts that it will receive something from one of the possible external choices defined in $\{l_i(S_i)\}_{i \in I}$ sent by p_1 ; the rest is similarly as for the sender. However, if p is not in this interaction, then its local type starts from the next interaction which p is in; moreover, because p does not know what choice that p_1 has made, every path $G_i\downarrow p$ lead by branch l_i shall be the same for p to ensure that interactions are consistent. For example, in $G = p_1 \rightarrow p_2\{l_1(S_1).p_3 \rightarrow p_1\ l_3(S), l_2(S_2).p_3 \rightarrow p_1\ l_4(S)\}$, interaction $p_3 \rightarrow p_1$ continues after $p_1 \rightarrow p_2$ takes place. If $l_3 \neq l_4$, then G is not projectable for p_3 because p_3 does not know which branch that p_1 has chosen; if p_1 chose branch l_1 , but p_3 (blindly) sends out label l_4 to p_1 , for p_1 it is a mistake (but it is not a mistake for p_3) because p_1 is expecting to receive label l_3 . To prevent such inconsistencies, we adopt the projection algorithm proposed in [24]. Other session type works [17, 39] provide ways to weaken the classical restriction on projection of branching which we use.

Rule (3) forbids a try-handle to appear in a recursive body, e.g., $\mu t.t(G)h(F : t)^\kappa.G$ is not allowed, but $t(\mu t.G)h(H)^\kappa$ and $t(G)h(F : \mu t.G', H)^\kappa$ are allowed. This is because κ is used to avoid confusion of messages from different try-handles. If a recursive body contains a try-handle, we have to dynamically generate different levels to maintain interaction consistency, so static type checking does not suffice. We are investigating alternative runtime checking mechanisms, but this is beyond the scope of this paper. Other rules are straightforward.

Example 6. Recall the global type G from Fig. 2 in Sect. 1. Applying projection rules defined in Definition 4 to G on every role in G we obtain the following:

$$\begin{aligned}
 T_{dfs} &= G\downarrow dfs = t(\mu t.w_1!l_{d_1}(S).w_2!l_{d_2}(S).w_1?l_{r_1}(S').w_2?l_{r_2}(S').t)h(\mathcal{H}_{dfs})^{(1,\emptyset)} \\
 \mathcal{H}_{dfs} &= \{w_1\} : \mu t'.w_2!l'_{d_1}(S).w_2?l'_{r_1}(S').t', \\
 &\quad \{w_2\} : \mu t''.w_1!l'_{d_2}(S).w_1?l'_{r_2}(S').t'', \{w_1, w_2\} : \text{end} \\
 T_{w_1} &= G\downarrow w_1 = t(\mu t.dfs?l_{d_1}(S).dfs!l_{r_1}(S').t)h(\mathcal{H}_{w_1})^{(1,\emptyset)} \\
 \mathcal{H}_{w_1} &= \{w_1\} : \text{end}, \{w_2\} : \mu t'.dfs?l'_{d_2}(S).dfs!l'_{r_2}(S').t', \{w_1, w_2\} : \text{end} \\
 T_{w_2} &= G\downarrow w_2 = t(\mu t.dfs?l_{d_2}(S).dfs!l_{r_2}(S').t)h(\mathcal{H}_{w_2})^{(1,\emptyset)} \\
 \mathcal{H}_{w_2} &= \{w_1\} : \mu t''.dfs?l'_{d_1}(S).dfs!l'_{r_1}(S').t'', \{w_2\} : \text{end}, \{w_1, w_2\} : \text{end}
 \end{aligned}$$

$$\begin{array}{c}
\frac{\Gamma \vdash a : \langle G \rangle}{\Gamma \vdash P \triangleright \{c : G[p]\}} \quad \frac{k \in I \quad \Gamma \vdash e : S_k}{\Gamma \vdash c : \eta_k \triangleright \{c : T_k\}} \quad \frac{}{\Gamma \vdash a[p].P \triangleright \emptyset} \quad \frac{}{\Gamma \vdash c : p! l_k(e).\eta_k \triangleright \{c : p! \{l_i(S_i).T_i\}_{i \in I}\}} \quad [\text{T-ini/T-snd}] \\
\\
\frac{\forall i \in I. \Gamma, x_i : S_i \vdash c : \eta_i \triangleright \{c : T_i\}}{\Gamma \vdash c : p? \{l_i(x_i).\eta_i\}_{i \in I} \triangleright \{c : p? \{l_i(S_i).T_i\}_{i \in I}\}} \quad [\text{T-rcv}] \\
\\
\frac{\Delta \text{ end-only}}{\Gamma \vdash c : \underline{0} \triangleright \Delta} \quad \frac{\Gamma \vdash c : \eta \triangleright \{c : \text{end}\}}{\Gamma \vdash c : \underline{0}.\eta \triangleright \{c : \underline{\text{end}}.\text{end}\}} \quad [\text{T-0/T-yd}] \\
\\
\frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash c : \text{if } e \eta_1 \text{ else } \eta_2 \triangleright \Delta} \quad \frac{\Gamma \vdash e : S}{\Gamma, X : S \quad T \vdash c : X\langle e \rangle \triangleright \{c : T\}} \quad [\text{T-if/T-var}] \\
\\
\frac{\Gamma, X : S \quad t, x : S \vdash c : \eta_1 \triangleright \{c : T'\} \quad \Gamma, X : S \quad \mu t.T' \vdash c : \eta_2 \triangleright \{c : T\}}{\Gamma \vdash c : \text{def } X(x) = \eta_1 \text{ in } \eta_2 \triangleright \{c : T\}} \quad [\text{T-def}] \\
\\
\frac{\Gamma \vdash c : \eta \triangleright \{c : T\} \quad \Gamma \vdash c : \eta' \triangleright \{c : T'\} \quad \text{dom}(\mathbf{H}) = \text{dom}(\mathcal{H}) \quad \forall F \in \text{dom}(\mathbf{H}). \Gamma \vdash c : \mathbf{H}(F) \triangleright \{c : \mathcal{H}(F)\}}{\Gamma \vdash c : \mathbf{t}(\eta)\mathbf{h}(\mathbf{H})^\phi.\eta' \triangleright \{c : \mathbf{t}(T)\mathbf{h}(\mathcal{H})^\phi.T'\}} \quad [\text{T-th}]
\end{array}$$

Fig. 10. Typing rules for processes

6 Type System

Next we introduce our type system for typing processes. Figures 10 and 11 present typing rules for endpoints processes, and typing judgments for applications and systems respectively.

We define shared environments Γ to keep information on variables and the coordinator, and session environments Δ to keep information on endpoint types:

$$\begin{array}{ll}
\Gamma ::= \emptyset \mid \Gamma, X : S \mid \Gamma, x : S \mid \Gamma, a : G \mid \Gamma, \Psi & \Delta ::= \emptyset \mid \Delta, c : T \mid \Delta, s : \mathbf{h} \\
\mathbf{m} ::= \langle p, q, l(S) \rangle \mid \langle p, \text{crash } F \rangle \mid \langle p, q \rangle^\phi & \mathbf{h} ::= \emptyset \mid \mathbf{h} \cdot \mathbf{m}
\end{array}$$

Γ maps process variables X and content variables x to their types, shared names a to global types G , and a coordinator $\Psi = G : (F, d)$ to failures and done notifications it has observed. Δ maps session channels c to local types and session queues to queue types. We write $\Gamma, \Gamma' = \Gamma \cup \Gamma'$ when $\text{dom}(\Gamma) \cap \text{dom}(\Gamma') = \emptyset$; same for Δ, Δ' . Queue types $\mathbf{h}$ are composed of message types $\mathbf{m}$. Their permutation is defined analogously to the permutation for messages. The typing judgment for local processes $\Gamma \vdash P \triangleright \Delta$ states that process P is well-typed by Δ under Γ .

Since we do not define sequential composition for processes, our type system implicitly forbids session interleaving by $[\text{T-ini}]$. This is different from other session type works [15, 24], where session interleaving is prohibited for the progress property; here the restriction is inherent to the type system.

Figure 10 lists our typing rules for endpoint processes. Rule $[\text{T-ini}]$ says that if a process's set of actions is well-typed by $G[p]$ on some c , this process can play role p in a , which claims to have interactions obeying behaviors defined in G . $\langle G \rangle$ means that G is closed, i.e., devoid of type variables. This rule forbids

$$\begin{array}{c}
\Gamma \vdash s : \emptyset \triangleright \{s : \emptyset\} \quad \frac{\Gamma \vdash s : h \triangleright \{s : h\} \quad \Gamma \vdash e : S}{\Gamma \vdash s : h \cdot \langle p, q, l(e) \rangle \triangleright \{s : h \cdot \langle p, q, l(S) \rangle\}} \quad [\text{T-}\emptyset/\text{T-m}] \\
\\
\frac{(p_1, p_2) \in \{(p, \psi), (\psi, p)\} \quad \Gamma \vdash s : h \triangleright \{s : h\}}{\Gamma \vdash s : h \cdot \langle p_1, p_2 \rangle^\phi \triangleright \{s : h \cdot \langle p_1, p_2 \rangle^\phi\}} \quad [\text{T-D}] \\
\\
\frac{p \in \{q, \psi\} \quad m = \langle p, \text{crash } F \rangle \quad \Gamma \vdash N_1 \triangleright \Delta_1 \quad \Gamma \vdash N_2 \triangleright \Delta_2 \quad \Gamma \vdash s : h \triangleright \{s : h\}}{\Gamma \vdash s : h \cdot \langle p, \text{crash } F \rangle \triangleright \{s : h \cdot m\}} \quad \frac{\text{dom}(\Delta_1) \cap \text{dom}(\Delta_2) = \emptyset}{\Gamma \vdash N_1 \mid N_2 \triangleright \Delta_1, \Delta_2} \quad [\text{T-F/T-pa}] \\
\\
\frac{\Gamma \vdash S \triangleright \Delta \quad \Gamma \vdash \Delta_s \text{ coherent}}{\Gamma \vdash (\nu s)S \triangleright \Delta \setminus \Delta_s} \quad \frac{\Gamma' = \Gamma, \Psi \quad \Gamma \vdash N \triangleright \Delta}{\Gamma' \vdash \Psi \blacklozenge N \triangleright \Delta} \quad [\text{T-s/T-sys}]
\end{array}$$

Fig. 11. Typing rules for applications and systems.

$a[p].b[q].P$ because a process can only use one session channel. Rule $[\text{T-snd}]$ states that an action for sending is well-typed to a sending type if the label and the type of the content are expected; $[\text{T-rcv}]$ states that an action for branching (i.e., for receiving) is well-typed to a branching type if all labels and the types of contents are as expected. Their follow-up actions shall also be well-typed. Rule $[\text{T-0}]$ types an idle process. Predicate end-only Δ is defined as stating whether all endpoints in Δ have type end:

Definition 5 (End-only Δ). We say Δ is end-only if and only if $\forall s[p] \in \text{dom}(\Delta), \Delta(s[p]) = \text{end}$.

Rule $[\text{T-yd}]$ types yielding actions, which only appear at runtime. Rule $[\text{T-if}]$ is standard in the sense that the process is well-typed by Δ if e has boolean type and its sub-processes (i.e., η_1 and η_2) are well-typed by Δ . Rules $[\text{T-var}, \text{T-def}]$ are based on a recent summary of MPSTs [14]. Note that $[\text{T-def}]$ forbids the type $\mu t.t$. Rule $[\text{T-th}]$ states that a try-handle is well-typed if it is annotated with the expected level ϕ , its default statement is well-typed, $\mathcal{H}$ and $\mathcal{H}$ have the same handler signatures, and all handling actions are well-typed.

Figure 11 shows typing rules for applications and systems. Rule $[\text{T-}\emptyset]$ types an empty queue. Rules $[\text{T-m}, \text{T-D}, \text{T-F}]$ simply type messages based on their shapes. Rule $[\text{T-pa}]$ says two applications composed in parallel are well-typed if they do not share any session channel. Rule $[\text{T-s}]$ says a part of a system S can start a private session, say s , if S is well-typed according to a $\Gamma \vdash \Delta_s$ that is *coherent* (defined shortly). The system $(\nu s)S$ with a part becoming private in s is well-typed to $\Delta \setminus \Delta_s$, that is, Δ after removing Δ_s .

Definition 6 (A Session Environment Having s Only: Δ_s)

$$\Delta_s = \{s[p] : T \mid s[p] \in \text{dom}(\Delta)\} \cup \{s : h \mid s \in \text{dom}(\Delta)\}$$

Rule $[\text{T-sys}]$ says that a system $\Psi \blacklozenge N$ is well-typed if application N is well-typed and there exists a coordinator Ψ for handling this application. We say $\Gamma \vdash \Delta$ is coherent under Γ if the local types of all endpoints are dual to each other after their local types are updated because of messages or notifications in $s : h$.

Coherence. We say that a session environment is *coherent* if, at any time, given a session with its latest messages and notifications, every endpoint participating in it is able to find someone to interact with (i.e., its dual party exists) right now or afterwards.

Example 7. Continuing with Example 6 – the session environment $\Gamma \vdash \Delta$ is coherent even if w_2 will not receive any message from dfs at this point. The only possible action to take in Δ is that dfs sends out a message to w_1 . When this action fires, Δ is reduced to Δ' under a coordinator. (The reduction relation $\Gamma \vdash \Delta \rightarrow_T \Gamma' \vdash \Delta'$, where $\Gamma = \Gamma_0, \Psi$ and $\Gamma' = \Gamma_0, \Psi'$, is defined based on the rules of operational semantics of applications in Sect. 4, Figs. 6 and 7). In Δ' , which abstracts the environment when dfs sends a message to w_1 , w_2 will be able to receive this message.

$$\begin{aligned} \Delta &= s[dfs] : T_{dfs}, s[w_1] : T_{w_1}, s[w_2] : T_{w_2}, s : \emptyset \\ \Delta' &= s[dfs] : t(w_2!l_{d_2}(S).w_1?l_{r_1}(S').w_2?l_{r_2}(S').T)h(\mathcal{H})^{(1,\emptyset)}, \\ &\quad s[w_1] : T_{w_1}, s[w_2] : T_{w_2}, s : \langle dfs, w_1, l_{d_1}(S) \rangle \\ &\quad \text{where } T = \mu t. w_1!l_{d_1}(S).w_2!l_{d_2}(S).w_1?l_{r_1}(S').w_2?l_{r_2}(S').t \end{aligned}$$

We write $s[p] : T \bowtie s[q] : T'$ to state that actions of the two types are *dual*:

Definition 7 (Duality). We define $s[p] : T \bowtie s[q] : T'$ as follows:

$$\begin{aligned} s[p] : \text{end} \bowtie s[q] : \text{end} \quad s[p] : \underline{\text{end}} \bowtie s[q] : \underline{\text{end}} \quad s[p] : \text{end} \bowtie s[q] : \underline{\text{end}} \\ s[p] : \underline{\text{end}} \bowtie s[q] : \text{end} \quad s[p] : t \bowtie s[q] : t \quad \frac{s[p] : T \bowtie s[q] : T'}{s[p] : \mu t. T \bowtie s[q] : \mu t. T'} \\ \frac{\forall i \in I. s[p] : T_i \bowtie s[q] : T'_i}{s[p] : q! \{l_i(S_i).T_i\}_{i \in I} \bowtie s[q] : p? \{l_i(S_i).T'_i\}_{i \in I}} \\ \frac{s[p] : T_1 \bowtie s[q] : T_2 \quad s[p] : T'_1 \bowtie s[q] : T'_2 \quad \text{dom}(\mathcal{H}_1) = \text{dom}(\mathcal{H}_2) \quad \forall F \in \text{dom}(\mathcal{H}_1). s[p] : \mathcal{H}_1(F) \bowtie s[q] : \mathcal{H}_2(F)}{s[p] : t(T_1)h(\mathcal{H}_1)^\phi.T'_1 \bowtie s[q] : t(T_2)h(\mathcal{H}_2)^\phi.T'_2} \end{aligned}$$

Operation $T \downarrow p$ is to filter T to get the partial type which only contains actions of p . For example, $p_1!l'(S').p_2!l(S) \downarrow p_2 = p_2!l(S)$ and $p_1!\{T_1, T_2\} \downarrow p_2 = p_2?l(S)$ where $T_1 = l_1(S_1).p_2?l(S)$ and $T_2 = l_2(S_2).p_2?l(S)$. Next we define $(h)_{p \rightarrow q}$ to filter h to generate (1) the normal message types sent from p heading to q , and (2) the notifications heading to q . For example $(\langle p, q, l(S) \rangle \cdot \langle q, \text{crash } F \rangle \cdot \langle \psi, q \rangle^\phi \cdot \langle p, \text{crash } F \rangle)_{p \rightarrow q} = p?l(S) \cdot \langle F \rangle \cdot \langle \psi \rangle^\phi$. The message types are abbreviated to contain only necessary information.

We define $T - \mathbf{ht}$ to mean the effect of $\mathbf{ht}$ on T . Its concept is similar to the *session remainder* defined in [35], which returns new local types of participants after participants consume messages from the global queue. Since failure notifications will not be consumed in our system, and we only have to observe the change of a participant's type after receiving or being triggered by some message types in $\mathbf{ht}$, we say that $T - \mathbf{ht}$ represents the effect of $\mathbf{ht}$ on T . The behaviors follows our operational semantics of applications and systems defined in Figs. 6, 7, and 8.

For example $\mathbf{t}(q?\{l_i(S_i).T_i\}_{i \in I})\mathbf{h}(\mathcal{H})^\phi.T' - q?l_k(S_k).\mathbf{ht} = \mathbf{t}(T_k)\mathbf{h}(\mathcal{H})^\phi.T' - \mathbf{ht}$ where $k \in I$.

Now we define what it means for Δ to be coherent under Γ :

Definition 8 (Coherence). $\Gamma \vdash \Delta$ coherent if the following conditions hold:

1. If $\mathbf{s} : \mathbf{h} \in \Delta$, then $\exists G : (F, d) \in \Gamma$ and $\{p \mid \mathbf{s}[p] \in \text{dom}(\Delta)\} \subseteq \text{roles}(G)$ and G is well-formed and $\forall p \in \text{roles}(G), G \upharpoonright p$ is defined.
2. $\forall \mathbf{s}[p] : T, \mathbf{s}[q] : T' \in \Delta$ we have $\mathbf{s}[p] : T \downarrow q - (\mathbf{h})_{q \rightarrow p} \bowtie \mathbf{s}[q] : T' \downarrow p - (\mathbf{h})_{p \rightarrow q}$.

In condition 1, we require a coordinator for every session so that when a failure occurs, the coordinator can announce failure notifications to ask participants to handle the failure. Condition 2 requires that, for any two endpoints, say $\mathbf{s}[p]$ and $\mathbf{s}[q]$, in Δ , equation $\mathbf{s}[p] : T \downarrow q - (\mathbf{h})_{q \rightarrow p} \bowtie \mathbf{s}[q] : T' \downarrow p - (\mathbf{h})_{p \rightarrow q}$, must hold. This condition asserts that interactions of non-failed endpoints are dual to each other after the effect of $\mathbf{h}$; while failed endpoints are removed from Δ , thus the condition is satisfied immediately.

7 Properties

We show that our type system ensures properties of subject congruence, subject reduction, and progress. All auxiliary definitions and proofs are in the long version [43].

The property of subject congruence states that if $\mathcal{S}$ (a system containing an application and a coordinator) is well-typed by some session environment, then a $\mathcal{S}'$ that is structurally congruent to it is also well-typed by the same session environment:

Theorem 1 (Subject Congruence). $\Gamma \vdash \mathcal{S} \triangleright \Delta$ and $\mathcal{S} \equiv \mathcal{S}'$ imply $\Gamma \vdash \mathcal{S}' \triangleright \Delta$.

Subject reduction states that a well-typed $\mathcal{S}$ (coherent session environment respectively) is always well-typed (coherent respectively) after reduction:

Theorem 2 (Subject Reduction)

- $\Gamma \vdash \mathcal{S} \triangleright \Delta$ with $\Gamma \vdash \Delta$ coherent and $\mathcal{S} \rightarrow^* \mathcal{S}'$ imply that $\exists \Delta', \Gamma'$ such that $\Gamma' \vdash \mathcal{S}' \triangleright \Delta'$ and $\Gamma \vdash \Delta \rightarrow_T^* \Gamma' \vdash \Delta'$ or $\Delta \equiv \Delta'$ and $\Gamma' \vdash \Delta'$ coherent.
- $\Gamma \vdash \mathcal{S} \triangleright \emptyset$ and $\mathcal{S} \rightarrow^* \mathcal{S}'$ imply that $\Gamma' \vdash \mathcal{S}' \triangleright \emptyset$ for some Γ' .

We allow sessions to run in parallel at the top level, e.g., $\mathcal{S} = (\nu s_1)(\Psi_1 \blacklozenge N_1) \mid \dots \mid (\nu s_n)(\Psi_n \blacklozenge N_n)$. Assume we have $\mathcal{S}$ with $a[p].P \in \mathcal{S}$. If we cannot apply rule **(Link)**, $\mathcal{S}$ cannot reduce. To prevent this kind of situation, we require $\mathcal{S}$ to be *initializable* such that, $\forall a[p].P \in \mathcal{S}$, **(Link)** is applicable.

The following property states that $\mathcal{S}$ never gets stuck (property of progress):

Theorem 3 (Progress). If $\Gamma \vdash \mathcal{S} \triangleright \emptyset$ and $\mathcal{S}$ is initializable, then either $\mathcal{S} \rightarrow^* \mathcal{S}'$ and $\mathcal{S}'$ is initializable or $\mathcal{S}' = \Psi \blacklozenge \mathbf{s} : h \mid \dots \mid \Psi' \blacklozenge \mathbf{s}' : h'$ and $h, \dots, h'$ only contain failure notifications sent by coordinators and messages heading to failed participants.

After all processes in $\mathcal{S}$ terminate, failure notifications sent by coordinators are left; thus the final system can be of the form $\Psi \blacklozenge \mathbf{s} : h \mid \dots \mid \Psi' \blacklozenge \mathbf{s}' : h'$, where $h, \dots, h'$ have failure notifications sent by coordinators and thus reduction rules (**CollectDone**), (**IssueDone**), and (**F**) will not be applied.

Minimality. The following proposition points out that, when all roles defined in a global type, say G , are robust, then the application obeying to G will never have interaction with a coordinator (i.e., interactions of the application are equivalent to those without a coordinator). This is an important property, as it states that our model does not incur coordination overhead when all participants are robust, or in failure-agnostic contexts as considered in previous MPST works.

Proposition 2. Assume $\forall p \in \text{roles}(G) = \{p_1, \dots, p_n\}$, p is robust and $P_i = \mathbf{s}[p_i] : \eta_i$ for $i \in \{1..n\}$ and $\mathcal{S} = (\nu \mathbf{s})(\Psi \blacklozenge P_1 \mid \dots \mid P_n \mid \mathbf{s} : h)$ where $P_i, i \in \{1..n\}$ contains no try-handle. Then we have $\Gamma \vdash \mathcal{S} \triangleright \emptyset$ and whenever $\mathcal{S} \rightarrow^* \mathcal{S}'$ we have $\Psi \in \mathcal{S}', \Psi = G : (\emptyset, \emptyset)$.

Proof. Immediately by typing rules $[\mathbf{T}\text{-ini}, \mathbf{T}\text{-s}, \mathbf{T}\text{-sys}]$, Definition 4 (Projection), and the operational semantics defined in Figs. 6, 7, and 8.

8 Related Work

Several session type works study exception handling [7, 9, 16, 30]. However, to the best of our knowledge this is the first theoretical work to develop a formalism and typing discipline for the coordinator-based model of *crash failure* handling in practical asynchronous distributed systems.

Structured interactional exceptions [7] study exception handling for binary sessions. The work extends session types with a *try-catch* construct and a *throw* instruction, allowing participants to raise runtime exceptions. Global escape [6] extends previous works on exception handling in binary session types to MPSTs. It supports nesting and sequencing of try-catch blocks with restrictions. Reduction rules for exception handling are of the form $\Sigma \vdash P \rightarrow \Sigma' \vdash P'$, where Σ is the *exception environment*. This central environment at the core of the semantics is updated synchronously and atomically. Furthermore, the reduction of a try-catch block to its continuation is done in a synchronous reduction step involving all participants in a block. Lastly this work can only handle exceptions, i.e., explicitly raised application-level failures. These do not affect communication channels [6], unlike participant crashes.

Similarly, our previous work [13] only deals with exceptions. An interaction $p \rightarrow q : S \vee F$ defines that p can send a message of type S to q . If F is not empty then instead of sending a message p can throw F . If a failure is thrown only participants that have casual dependencies to that failure are involved in the failure handling. No concurrent failures are allowed therefore all interactions which can raise failures are executed in a lock step fashion. As a consequence, the model can not be used to deal with crash failures.

Adameit et al. [1] propose session types for link failures, which extend session types with an optional block which surrounds a process and contains default values. The default values are used if a link failure occurs. In contrast to our work, the communication model is overall synchronous whereas our model is asynchronous; the optional block returns default values in case of a failure but it is still the task of the developer to do something useful with it.

Demangeon et al. study interrupts in MPSTs [16]. This work introduces an interruptible block $\{|G|\}^c \langle l \text{ by } \mathbf{r}; G' \rangle$ identified by c ; here the protocol G can be interrupted by a message l from $\mathbf{r}$ and is continued by G' after either a normal or an interrupted completion of G . Interrupts are more a control flow instruction like exceptions than an actual failure handling construct, and the semantics can not model participant crashes.

Neykova and Yoshida [36] show that MPSTs can be used to calculate safe global states for a safe recovery in Erlang's *let it crash* model [2]. That work is well suited for recovery of lightweight processes in an actor setting. However, while it allows for elaborate failure handling by connecting (endpoint) processes with runtime monitors, the model does not address the fault tolerance of runtime monitors themselves. As monitors can be interacting in complex manners replication does not seem straightforwardly applicable, at least not without potentially hampering performance (just as with *straightforward* replication of entire applications).

Failure handling is studied in several process calculi and communication-centered programming languages without typing discipline. The conversation calculus [42] models exception behavior in abstract service-based systems with message-passing based communication. The work does not use channel types but studies the behavioral theory of bisimilarity. Error recovery is also studied in a concurrent object setting [45]; interacting objects are grouped into coordinated atomic actions (CAs) which enable safe error recovery. CAs can however not be nested. PSYNC [18] is a domain specific language based on the *heard-of* model of distributed computing [12]. Programs written in PSYNC are structured into rounds which are executed in a lock step manner. PSYNC comes with a state-based verification engine which enables checking of safety and liveness properties; for that programmers have to define non-trivial inductive invariants and ranking functions. In contrast to the coordinator model, the *heard-of* model is not widely deployed in practice. Verdi [44] is a framework for implementing and verifying distributed systems in Coq. It provides the possibility to verify the system against different network models. Verdi enables the verification of properties in an idealized fault model and then transfers the guarantees to more realistic fault models by applying transformation functions. Verdi supports safety properties but no liveness properties.

9 Final Remarks

Implementation. Based on our presented calculus we developed a domain-specific language and corresponding runtime system in Scala, using ZooKeeper as the

coordinator. Specifically our implementation provides mechanisms for (1) interacting with ZooKeeper as coordinator, (2) done and failure notification delivery and routing, (3) practical failure detection and dealing with false suspicions and (4) automatically inferring try-handle levels.

Conclusions. This work introduces a formal model of verified crash failure handling featuring a lightweight coordinator as common in many real-life systems. The model carefully exposes potential problems that may arise in distributed applications due to partial failures, such as inconsistent endpoint behaviors and orphan messages. Our typing discipline addresses these challenges by building on the mechanisms of MPSTs, e.g., global type well-formedness for sound failure handling specifications, modeling asynchronous permutations between regular messages and failure notifications in sessions, and the type-directed mechanisms for determining correct and orphaned messages in the event of failure. We adapt coherence of session typing environments (i.e., endpoint consistency) to consider failed roles and orphan messages, and show that our type system statically ensures subject reduction and progress in the presence of failures.

Future Work. We plan to expand our implementation and develop further applications. We believe dynamic role participation and role parameterization would be valuable for failure handling. Also, we are investigating options to enable addressing the coordinator as part of the protocol so that pertinent runtime information can be persisted by the coordinator. We plan to add support to our language and calculus for solving various explicit agreement tasks (e.g., consensus, atomic commit) via the coordinator.

References

1. Adameit, M., Peters, K., Nestmann, U.: Session types for link failures. In: Bouajjani, A., Silva, A. (eds.) FORTE 2017. LNCS, vol. 10321, pp. 1–16. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60225-7_1
2. Armstrong, J.: Making reliable distributed systems in the presence of software errors. Ph.D. thesis, Royal Institute of Technology, Stockholm, Sweden (2003)
3. Birman, K.P.: Byzantine Clients (2017). <https://goo.gl/1Qbc4r>
4. Burrows, M.: The Chubby lock service for loosely-coupled distributed systems. In: OSDI 2006, pp. 335–350. USENIX Association (2006)
5. Caires, L., Pérez, J.A.: Multiparty session types within a canonical binary theory, and beyond. In: Albert, E., Lanese, I. (eds.) FORTE 2016. LNCS, vol. 9688, pp. 74–95. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39570-8_6
6. Capecchi, S., Giachino, E., Yoshida, N.: Global escape in multiparty sessions. *MSCS* **26**(2), 156–205 (2016)
7. Carbone, M., Honda, K., Yoshida, N.: Structured interactional exceptions in session types. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 402–417. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85361-9_32

8. Carbone, M., Lindley, S., Montesi, F., Schürmann, C., Wadler, P.: Coherence generalises duality: a logical explanation of multiparty session types. In: CONCUR 2016. LIPIcs, vol. 59, pp. 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
9. Carbone, M., Yoshida, N., Honda, K.: Asynchronous session types: exceptions and multiparty interactions. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM 2009. LNCS, vol. 5569, pp. 187–212. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01918-0_5
10. Chandra, T.D., Hadzilacos, V., Toueg, S., Charron-Bost, B.: On the impossibility of group membership. In: PODC 1996, pp. 322–330. ACM (1996)
11. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.: Bigtable: a distributed storage system for structured data. In: OSDI 2006, pp. 205–218. USENIX Association (2006)
12. Charron-Bost, B., Schiper, A.: The Heard-Of model: computing in distributed systems with benign faults. *Distrib. Comput.* **22**(1), 49–71 (2009)
13. Chen, T.-C., Viering, M., Bejleri, A., Ziarek, L., Eugster, P.: A type theory for robust failure handling in distributed systems. In: Albert, E., Lanese, I. (eds.) FORTE 2016. LNCS, vol. 9688, pp. 96–113. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39570-8_7
14. Coppo, M., Dezani-Ciancaglini, M., Padovani, L., Yoshida, N.: A gentle introduction to multiparty asynchronous session types. In: Bernardo, M., Johnsen, E.B. (eds.) SFM 2015. LNCS, vol. 9104, pp. 146–178. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18941-3_4
15. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L.: Global progress for dynamically interleaved multiparty sessions. *MSCS* **26**(2), 238–302 (2016)
16. Demangeon, R., Honda, K., Hu, R., Neykova, R., Yoshida, N.: Practical interruptible conversations. *Formal Methods Syst. Des.* **46**(3), 197–225 (2015)
17. Deniérou, P.M., Yoshida, N.: Dynamic multirole session types. In: POPL 2011, pp. 435–446. ACM (2011)
18. Dragoi, C., Henzinger, T., Zufferey, D.: PSync: a partially synchronous language for fault-tolerant distributed algorithms. In: POPL 2016, pp. 400–415. ACM (2016)
19. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. *J. ACM* **32**(2), 374–382 (1985)
20. Ghemawat, S., Gobiuff, H., Leung, S.T.: The Google file system. In: SOSP 2003, pp. 29–43. ACM (2003)
21. Gilbert, S., Lynch, N.: Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* **33**(2), 51–59 (2002)
22. Guerraoui, R., Schiper, A.: The generic consensus service. *IEEE Trans. Softw. Eng.* **27**(1), 29–41 (2001)
23. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0053567>
24. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. *J. ACM* **63**(1), 9:1–9:67 (2016)
25. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: Stevens, P., Wasowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 401–418. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49665-7_24
26. Hunt, P.: ZooKeeper: wait-free coordination for internet-scale systems. In: USENIX 2010. USENIX Association (2010)

27. Hüttel, H., et al.: Foundations of session types and behavioural contracts. *ACM Comput. Surv.* **49**(1), 3:1–3:36 (2016)
28. Imai, K., Yoshida, N., Yuen, S.: **Session-ocaml**: a session-based library with polarities and lenses. In: Jacquet, J.-M., Massink, M. (eds.) *COORDINATION 2017*. LNCS, vol. 10319, pp. 99–118. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59746-1_6
29. Killian, C.E., Anderson, J.W., Braud, R., Jhala, R., Vahdat, A.M.: Mace: language support for building distributed systems. In: *PLDI 2007*, vol. 42, pp. 179–188. ACM (2007)
30. Kouzapas, D., Yoshida, N.: Globally governed session semantics. *LMCS* **10**(4), 1–45 (2014)
31. Kreps, J., Narkhede, N., Rao, J.: Kafka: a distributed messaging system for log processing. In: *NetDB 2011* (2011)
32. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.* **4**(3), 382–401 (1982)
33. Leners, J.B., Wu, H., Hung, W.L., Aguilera, M.K., Walfish, M.: Detecting failures in distributed systems with the FALCON spy network. In: *SOSP 2011*, pp. 279–294. ACM (2011)
34. Lindley, S., Morris, J.G.: embedding session types in haskell. In: *Haskell 2016*, pp. 133–145. ACM (2016)
35. Mostrous, D., Yoshida, N.: Session typing and asynchronous subtyping for the higher-order π -calculus. *Inf. Comput.* **241**, 227–263 (2015)
36. Neykova, R., Yoshida, N.: Let it recover: multiparty protocol-induced recovery. In: *CC 2017*, pp. 98–108. ACM (2017)
37. Padovani, L.: A simple library implementation of binary sessions. *J. Funct. Program.* **27**, e4 (2017)
38. Pucella, R., Tov, J.A.: Haskell session types with (almost) no class. In: *Haskell 2008*, pp. 25–36. ACM (2008)
39. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: *ECOOP 2017*. *LIPICs*, vol. 74, pp. 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
40. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop distributed file system. In: *MSST 2010*, pp. 1–10. IEEE Computer Society (2010)
41. Sivaramakrishnan, K.C., Qudeisat, M., Ziarek, L., Nagaraj, K., Eugster, P.: Efficient sessions. *Sci. Comput. Program.* **78**(2), 147–167 (2013)
42. Vieira, H.T., Caires, L., Seco, J.C.: The conversation calculus: a model of service-oriented computation. In: Drossopoulou, S. (ed.) *ESOP 2008*. LNCS, vol. 4960, pp. 269–283. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78739-6_21
43. Viering, M., Chen, T.C., Eugster, P., Hu, R., Ziarek, L.: Technical appendix: a typing discipline for statically verified crash failure handling in distributed systems. http://distributed-systems-programming-group.github.io/paper/2018/esop_long.pdf
44. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.E.: Verdi: a framework for implementing and formally verifying distributed systems. In: *PLDI 2015*, pp. 357–368. ACM (2015)
45. Xu, J., Randell, B., Romanovsky, A.B., Rubira, C.M.F., Stroud, R.J., Wu, Z.: Fault tolerance in concurrent object-oriented software through coordinated error recovery. In: *FTCS 1995*, pp. 499–508. IEEE Computer Society (1995)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.