A Multiparty Session Typing Discipline for Fault-Tolerant Event-Driven Distributed Programming∗

MALTE VIERING, Technische Universität Darmstadt, Germany
RAYMOND HU, Queen Mary University of London, UK
PATRICK EUGSTER, Universität della Svizzera italiana (USI), Switzerland and Purdue University, USA
LUKASZ ZIAREK, University at Buffalo, USA

This paper presents a formulation of multiparty session types (MPSTs) for practical fault-tolerant distributed programming. We tackle the challenges faced by session types in the context of distributed systems involving asynchronous and concurrent partial failures – such as supporting dynamic replacement of failed parties and retrying failed protocol segments in an ongoing multiparty session – in the presence of unreliable failure detection. Key to our approach is that we develop a novel model of event-driven concurrency for multiparty sessions. Inspired by real-world practices, it enables us to unify the session-typed handling of regular I/O events with failure handling and the combination of features needed to express practical fault-tolerant protocols. Moreover, the characteristics of our model allow us to prove a global progress property for well-typed processes engaged in multiple concurrent sessions, which does not hold in traditional MPST systems.

To demonstrate its practicality, we implement our framework as a toolchain and runtime for Scala, and use it to specify and implement a session-typed version of the cluster management system of the industrial-strength Apache Spark data analytics framework. Our session-typed cluster manager composes with other vanilla Spark components to give a functioning Spark runtime; e.g., it can execute existing third-party Spark applications without code modification. A performance evaluation using the TPC-H benchmark shows our prototype implementation incurs an average overhead below 10%.

CCS Concepts: • Theory of computation → Type structures; • Software and its engineering → Distributed programming languages; Software fault tolerance.

Additional Key Words and Phrases: session types, type systems, asynchronous multiparty session types, fault-tolerant distributed programming, event-driven programming

ACM Reference Format:
Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. 2021. A Multiparty Session Typing Discipline for Fault-Tolerant Event-Driven Distributed Programming. Proc. ACM Program. Lang. 5, OOPSLA, Article 124 (October 2021), 30 pages. https://doi.org/10.1145/3485501

1 INTRODUCTION

Background: distributed systems, partial failures, and MPSTs. Correctly designing and implementing distributed systems is notoriously difficult. Even an idealized setting requires reasoning about independent program components concurrently executing on a set of physically remote processes.

∗This project is supported in part by ERC grant FP7-617805, the Hasler Foundation, SNSF grant 197353, BMBF grant 01IS17050, and NSF Awards CNS-1823230 and SHF-1749539.

Authors’ addresses: Malte Viering, viering@dsp.tu-darmstadt.de, Technische Universität Darmstadt, Germany; Raymond Hu, r.hu@qmul.ac.uk, Queen Mary University of London, UK; Patrick Eugster, eugstp@usi.ch, Universität della Svizzera italiana (USI), Switzerland and and Purdue University, USA; Lukasz Ziarek, lziarek@buffalo.edu, University at Buffalo, USA.
separated by a network that offers communication only via asynchronous message passing. More realistically, the distributed system will also be subject to partial failures where some processes may fail while others continue operating.

Multiparty session types (MPSTs) [Carbone et al. 2008; Honda et al. 2016] is an active area of research on typing disciplines for concurrent processes; see Ancona et al. [2016]; Gay and Ravara [2017]; Hüttel et al. [2016] for surveys of session types and the broader area of behavioral types. The idea is to apply type systems to message passing programs to statically ensure properties such as absence of message reception errors and deadlocks. MPSTs is a promising technique for distributed systems: a key notion in MPST theory is projection, which derives a decoupled (i.e., distributed) view of a protocol from the perspective of each participant. Existing approaches in MPSTs are, however, not applicable to a significant class of real-world distributed systems, as they do not support the specification and verification of practical protocols dealing with partial failures.

Reasoning about failures is a long-standing challenge for MPSTs and related techniques. First, the chaotic nature of communication patterns involving partial failures among concurrent, asynchronously interacting participants is at odds with the traditional ethos of session types, which seeks to ensure safety by strictly regulating all potential interactions. Second, protocols for fault-tolerant distributed systems often require a combination of relatively advanced features for handling failures, e.g., dynamically replacing failed participants and retrying failed protocol segments in an application, often in the presence of imperfect failure detection. These challenges must be overcome for techniques like session types to be applicable to practical distributed systems.

In the existing literature on MPSTs, there are two main works on the topic of failure handling [Adameit et al. 2017; Viering et al. 2018]. We elaborate on related work in Sec. 8, but to summarize, both are theoretical works and limited in practical applicability. The system of Adameit et al. [2017] is essentially based on a synchronous communication model. Viering et al. [2018] assume perfect failure detection (unimplementable in an asynchronous system) and propose a new programming model based on an asynchronous variant of try-catch that is remote from existing practices. Crucially, neither support the range of features required to express dynamic participant replacement and retrying in an on-going fault-tolerant application.

This paper. This paper develops the first MPST-based theory for practical programming of fault-tolerant distributed applications. Our aim is to support MPST-based specification and verification of multiparty asynchronous protocols that deal with crash-stop process failures, to ensure the key properties of communication safety and progress. We present our approach through the use case of a substantial system from an industry-strength data analytics framework: the distributed cluster management (CM) system of Apache Spark1 [Zaharia et al. 2012].

As mentioned, a crucial challenge is to not only model failure-prone sessions, but to support the combination of features needed in fault-tolerant applications, like Apache Spark. To this end, our theory draws inspiration from one of the major paradigms for distributed programming in practice: event-driven programming (EDP). Event-driven concurrency allows for highly concurrent and asynchronous programming commensurate with the nature of the network; for instance, EDP is an integral part of Spark’s runtime. We develop the first model of event-driven concurrency for multiparty sessions, and exploit the characteristics of EDP to facilitate the integration of “regular” session I/O handling with features for failure detection and failure notifications via asynchronous messaging, dynamic role assignment based on a notion of role sets (for replacing failed roles), and subsession spawning (for attempting/retrying failure-prone protocol segments) – EDP allows us to handle all these kinds of behaviors as events under a uniform session-typeable programming interface. The Spark CM use case depends on all of these features. Moreover, the characteristics

---

1 Apache Spark. http://spark.apache.org.
of EDP allow us to prove a *global progress* property for well-typed processes engaged in multiple concurrent (sub)sessions, which does not hold in traditional MPST systems.

To demonstrate the practicality of our approach, we implement our system as a toolchain for fault-tolerant distributed programming in Scala\(^2\), and use it to specify and implement a session-typed version of Spark’s CM\(^3\). Our session-typed CM is compatible with other vanilla Spark components, providing a functioning Spark runtime that can execute existing third-party Spark applications *without* any code modification. This is enabled by the design of our theory, in (1) targeting EDP as a practical, established programming model for asynchronous I/O, (2) supporting the above combination of features needed for practical fault-tolerant protocols, and (3) appropriately abstracting failure detection mechanisms, including catering for unreliable failure detection (i.e., *false suspicions* of remote process failures). We use a Spark implementation of the industry-standard TPC-H benchmark suite\(^4\) to test our session-typed CM and evaluate its runtime performance. Measuring the time it takes from submitting a Spark application to its completion, the results show our prototype implementation incurs an average overhead below 10%.

The roadmap and the contributions of this paper are as follows.

- We present an MPST-based framework that integrates the features needed to specify practical fault-tolerant protocols. Sec. 2 illustrates these features, including *failure-aware subsessions and rolesets* (participant parameterization and dynamic role assignment), using the Spark CM use case.
- We develop the first model of *event-driven concurrency* for multiparty sessions. Sec. 3 demonstrates endpoint projection and implementation of distributed, event-driven processes using our Scala toolchain. The central abstractions are *session-typed event handlers* and *event loops*.
- Sec. 4 and Sec. 5 formalize our framework. Sec. 6 presents the key properties of *communication safety and global progress* for an entire well-typed system of *multiple, concurrent subsessions*.
- Sec. 7 presents a practical evaluation of our framework. We use our Scala toolchain to specify and implement examples from MPSTs literature in addition to the Spark CM use case, and present the runtime performance of using the latter to execute the TPC-H benchmark.

Sec. 8 discusses related work, and Sec. 9 concludes with future work. The accompanying online supplement\(^5\) provides additional examples, auxiliary definitions, and proof details.

### 2 MPSTS FOR FAULT-TOLERANT SESSIONS

#### 2.1 Running Example: Apache Spark’s Standalone Cluster Manager

Apache Spark is a big data analytics framework for distributed clusters. Spark supports several types of *cluster managers* – e.g., Spark’s standalone cluster manager (Spark-CM), Mesos [Hindman et al. 2011] or YARN [Vavilapalli et al. 2013] – for managing the distributed resources in running Spark applications\(^6\). We use the Spark-CM as a concrete use case (Sec. 7) and a running example.

Fig. 1 depicts the execution of a Spark application as managed by the Spark-CM, including recovery from a process failure. An application involves the *manager* (called *master* in Spark’s source code) and a pool of distributed *worker* processes. The Spark-CM launches a new application by (a) first *assigning* a worker process as the application *driver*, and then (b) assigning multiple workers as *executors* (we depict just one for simplicity). The driver hosts the main user program while the executors perform their allocated computations. If a driver or executor process fails, as in (c), the Spark-CM preserves the running application by assigning another worker to replace the

---

\(^2\)Toolchain implementation. [https://github.com/viering/MPST_toolchain_for_fault-tolerant_distributed_EDP](https://github.com/viering/MPST_toolchain_for_fault-tolerant_distributed_EDP)

\(^3\)Spark use case implementation. [https://github.com/viering/spark_cm_use_case](https://github.com/viering/spark_cm_use_case)

\(^4\)TPC-H benchmark. [http://www.tpc.org/tpch/](http://www.tpc.org/tpch/)

\(^5\)Online supplement. [https://viering.github.io/page/papers/viering21.pdf](https://viering.github.io/page/papers/viering21.pdf)

\(^6\)Apache Spark cluster mode overview. [https://spark.apache.org/docs/latest/cluster-overview.html](https://spark.apache.org/docs/latest/cluster-overview.html)
failed one. The cause of a process failure may be the failure of its host machine (the latter can be represented by the former).

The application as a whole thus comprises a set of concurrent, collaborative subtasks that these distributed processes carry out via asynchronous message passing over the network. E.g., in (b) the master and driver may continue the task of spawning further executors, while concurrently engaging with the existing executor in a separate computation subtask. This use case demonstrates a general class of distributed systems and the combination of features they depend on, which existing MPSTs do not support. Systems such as the Spark-CM depend on:

- dynamically initiating new concurrent tasks between subsets of the existing participants (e.g., allocating and performing work across different workers in parallel).
- dynamically assigning the participants for the above (e.g., when assigning executors);
- handling of and recovery from participant failures (e.g., replacing a failed executor);
- accounting for the fact that, in general, failure detection is asynchronous and potentially unreliable, i.e., a participant may be falsely suspected of failure even as it continues operating.

2.2 Limitations of Standard MPSTs
The standard workflow of MPSTs [Coppo et al. 2016; Honda et al. 2016] starts from the user specification of a global type, describing all possible interactions between participants from a global perspective. Consider this global type (written in a practical notation for later comparisons):

\[
\text{RunEx}(\text{roles master, driver, exec}) \{ \ldots \text{master} \rightarrow \text{driver}: \text{InitDr} \ldots \}
\]

This protocol named RunEx (short for “run executor”) is between three participants, abstracted as roles master (the manager), driver, and exec. The above fragment specifies that master sends an InitDr message to driver, which in turn then sends an InitEx to exec.

We can use this minimal example to clarify some limitations of the session abstraction developed in standard MPSTs. First, a session involves a fixed set of participants, one participant playing each role, that are all fixed when the session is first initiated: there is no notion of dynamically (i.e., during an ongoing session) assigning a participant to a role, nor losing or replacing assigned participants. Second, there is no notion of a failure event nor how the protocol should proceed in such an event – standard MPSTs support protocol branching (choice) and recursion, but their strictly regulated nature (to conservatively ensure safety) cannot capture the asynchronous nature of failures. Standard MPSTs thus cannot express protocols involving failure handling and recovery, nor verify processes that engage in such behaviors.

2.3 MPSTs for Fault-tolerant Sessions (Initial Overview)
This paper develops a generalized notion of session to address the limitations of standard MPSTs and better support real-world distributed systems such as the Spark-CM. In our setting, a session
is generalized as a system of failure-aware, concurrent subsessions. We first explain individual subsessions, then inter-subsession concurrency.

**Role sets.** We introduce an abstraction of session participants called role sets. In standard MPSTs, a role represents a participant process (participant, for short) with a certain behavior. By contrast, a role set in our model (written with an initial uppercase letter, e.g., Masters, Workers) represents a non-empty set of participant processes that are capable of the same behavior. In this setting, a role (initial lowercase letter, e.g., a master m, a driver worker wD, or an executor worker wE) represents a single participant dynamically assigned from its role set (as explained shortly). In addition to the standard role-to-role message passing, e.g., master → driver, a role can multisend a message to a whole role set, e.g., master → Workers.

Role sets provide a practical, lightweight form of participant parametricity. Our framework abstracts from the run-time size of a role set (i.e., the number of participants) and how processes are spawned; it assumes only that a sufficient number of participants are available at run-time.

**Subsessions.** The following subprotocol illustrates our general notion of failure-aware subsession:

```plaintext
RunEx(roles m: Masters, wD: Workers; assign wE: Workers; rosets Workers) {  // a (sub)protocol
  ... m → wE: InitEx ...  // the "normal activity"
  with wE@m.            // wE is monitored by m for potential failure
  ... m → Workers: FailEx ... }  // the "failure handling activity", if wE is _suspected_ by m
```

It specifies a communication (sub)session whose participants are the individual processes playing roles m, wD, and wE, combined with all member processes of the role set Workers. Each role is annotated by its role set (e.g., wD: Workers). The assign declaration specifies to dynamically assign a participant from role set Workers to play role wE when this subsession is initiated. roles specifies participants already assigned roles prior to this subsession, and rosets specifies role sets. Our framework is agnostic of how the assign mechanism is implemented, provided that the assigned participant is not also playing one of the roles in this subsession.

The main subprotocol definition has two parts: the normal activity written before the with, and the failure handling activity after. The wE@m clause specifies that, in this subsession, m is responsible for monitoring wE for failure. (In general, any one of roles may be the monitor.) By default, the subsession proceeds following the interactions in the normal activity. However, if m suspects wE of failure, m will switch to the failure handling; it may do so at any point during the subsession. Global type well-formedness requires all other participants (i.e., except wE) to be notified of wE’s failure via explicit interactions in the failure handling; these participants also switch to the failure handling upon receiving such a notification (i.e, FailEx), and (barring other failures) the remaining participants in this subsession will proceed following the failure handling.

To simplify the presentation of our theory, we restrict a subsession to one assign; multiple role assignments can be expressed via nested subsessions (see below) or supported as a syntactic sugar.

Our framework is agnostic of any particular failure detection mechanism. Our theory simply models failure suspicions asynchronously and non-deterministically – this is a key design choice for broader practical applicability, including settings with unreliable failure detection.

**Concurrent subsessions.** In our framework, a session is a system of one or more concurrent subsessions. Any one participant process may be involved in any number of concurrent subsessions, playing an individual role and/or as a generic role set member in each.

A session starts from a root (or entry) subsession, i.e., an instance of the root subprotocol, which collects together every participant that may be involved at some point in the overall session.

```plaintext
root RunDr(roles m: Masters; assign wD: Workers; rosets Workers) { ... with wD@m ... }  // root
```
Fig. 2. Global type for the Session-CM: a session-typed Spark-CM. RunDr is the root subprotocol.

We refer to the roles in the root subsession as robust roles. A global type does not explicitly specify failure monitoring for robust roles; we assume they are monitored for failure by external mechanisms, or their failure simply causes the entire session (i.e., all participants) to fail.

More generally, a subsession is a child subsession spawned by an initiation interaction between the relevant subset of participants in its parent subsession. E.g., from within the root RunDr, we can spawn a new subsession instance of RunEx (from above) by:

```plaintext
... spawn RunEx(m, wD; Workers; Workers) ... // from inside RunDr; RunEx is as above
```

The arguments correspond to the parameter declarations of RunEx. The roles arguments may be any existing role from the specified role set, and the assign argument is the role set from which to assign a participant. The spawned subsession runs concurrently with its parent (i.e., with the parent’s interactions following spawn), meaning that any processes participating in both do so concurrently.

A running session thus forms a tree of parent-child subsessions, which we refer to as the monitoring tree. Excluding the robust role, every individual role that occurs in some subsession has a monitor, in that subsession or an ancestor (i.e., the subsession in which the role was first assigned). In other words, every case of (non-robust) participant failure is safely covered within the session as a whole. This gives some intuition of how our framework incorporates subsessions, dynamic role assignment, and failure handling while retaining the desired MPST safety and progress properties (Sec. 3.3 and 6.2). We explain this further using the Spark-CM example below.

Note, spawn actions may occur within both normal and failure handling activities. The latter are crucial to express fault-tolerant protocols that replace failed participants and retry failed subsessions.

### 2.4 Global Type for the Component Interactions in the Spark-CM

We can now express the message passing behavior of fault-tolerant protocols as MPST global types. Fig. 2 specifies a global type for a simplified but representative version of the Spark-CM, which we refer to as the Session-CM; our full version is compatible with the actual Spark framework and supports existing third-party Spark applications. The Session-CM has two subprotocols, RunDr and RunEx, for handling the lifecycle of a driver and an executor in a Spark application, respectively.

The root subprotocol, RunDr, involves these participants: a (robust) manager role m, a driver role wD assigned from role set Workers, and the role set Workers as a whole. To launch a Spark application, RunDr dynamically assigns a driver process (as part of session initiation) and initializes it (lines 3–4). RunDr then spawns a number of concurrent child subsessions that assign executors to carry out the subtasks of the application in parallel (lines 5–8).

Line 3 is the standard role-to-role message passing of MPSTs [Coppo et al. 2016; Honda et al. 2016]: role m sends an InitDr message (carrying a DrInfo payload) to role wD to initialize the driver;
Global subprotocols (e.g., RunDr, RunEx) → Local subprotocols (e.g., for Masters) → Scala API (for Masters) → Fault-tolerant event-driven Scala program (for Masters)

**Fig. 3.** Scala toolchain for MPST-based fault-tolerant distributed programming. Gray is written by the user.

\[
\text{wD then replies to m with an acknowledgment on Line 4. As standard in MPSTs, and crucial for real-world applications, messaging in our framework is asynchronous, meaning output actions do not block (but input actions do block until a message is received). Lines 5–8 specify a standard branch type m → Workers (\text{AddEx}(): \ldots, \text{Ok}(): \ldots): role m makes an internal choice to, in this instance, \text{multisend} either \text{AddEx} or \text{Ok} messages to the whole of Workers, who will follow the received message case as an external choice. Note that wD belongs to role set Workers, so the participant playing wD receives the branch message this way.}
\]

On line 5, \text{mut ...} is a standard recursive type: the recursion is enacted by the \text{t} in the \text{AddEx} case (causing the subsession to “loop back” to line 5), while \text{Ok} leads to subsession termination \text{end}. As part of the recursive branch case, line 7 may be repeated to \text{spawn} multiple \text{RunEx} subsessions, all running concurrently with the \text{RunDr} subsession. Each child subsession involves: m and wD from \text{RunDr} as \text{roles}, an assigned Workers participant to play the new role \text{wE}, and the role set Workers. Our \text{spawn} is novel to MPSTs in supporting (i) unrestricted spawning of child tasks, where participants may be involved in multiple such tasks, that are (ii) fully concurrent with the parent task. Notably, our theory ensures \text{global progress} for the entire system of all subsessions: prior work on nested protocols [Demangeon and Honda 2012] in MPSTs ensures progress only for a single session.

Lastly, lines 9–11 specify the failure handling behavior for the \text{RunDr} subsession. If the monitor m suspects the participant playing wD has failed (by whatever failure detector it employs), it switches from the normal to the failure handling activity in this subsession \text{and} stops its activity in all descendent subsessions. Our framework requires the monitor to explicitly notify all (potentially) remaining participants as part of the failure handling: m does so by a multisend to Workers (this implicitly excludes the suspected participant), who switch to failure handling and stop any descendent activities upon receiving the notification. A new \text{RunDr} subsession is then spawned between m and the remaining Workers, with some \text{other} participant assigned to wD (i.e., a replacement).

The structure of \text{RunEx} is similar. It dynamically assigns and initializes an executor to carry out some of the actual application work. By default, our theory permits the same participant to play \text{wE} in multiple \text{RunEx} subsessions. Note, a failure of wD during \text{RunEx} is handled by the parent \text{RunDr} subsession (as mentioned, remaining participants stop activity in all descendent subsessions when failure handling is activated); to reiterate, all \text{roles} (except robust roles) are \text{necessarily} monitored in the subsession where they were first assigned.

### 3 EVENT-DRIVEN PROGRAMMING FOR FAULT-TOLERANT SESSIONS

#### 3.1 Scala Toolchain Overview

Fig. 3 shows the main stages and artifacts in our MPST-based toolchain for fault-tolerant distributed programming in Scala. It is designed to support concurrent subsessions, failure handling, and \textit{event-driven programming} (EDP) following a standard MPST top-down workflow. It starts from a user specification of the fault-tolerant application protocol as a global type, i.e., a set of interdependent subprotocols. We continue the Session-CM example (Fig. 2) to illustrate the subsequent stages.

**Local types and CFSMs.** A global type is \textit{projected} to a local type describing the view of the protocol from each kind of participant. In our setting, each participant kind (e.g., a master) must support its behavior as a \textit{generic} role set member (i.e., Masters) \textit{and} as all its (potential) roles (i.e., m) – note, in
Sec. 4.2 presents projection in detail; here, we first illustrate the concept by showing the communicating finite state machine (CFSM) representation for local types used internally by our toolchain implementation. In a nutshell, our toolchain represents each localised subprotocol as a pair of CFSMs, for the normal and failure handling activities; e.g., Fig. 4 (top) depicts the CFSM pair for role m in RunDr. Each state represents a point in the subprotocol where this participant performs one of the localized interaction types: select (output choice, subsuming the basic send), branch (input choice, subsuming receive), failure suspicion (by a monitor), spawn (subsession initiation), or end (subsession termination). By the design of our framework, spawn and suspicion states have exactly one outgoing transition, and end is a terminal state. Each transition represents an I/O action from that state to the next. Output ! is a non-blocking action. The potentially blocking actions are input ? and spawn (denoted by the subprotocol name). Following session initiation in standard MPSTs, spawn is a synchronization between the relevant participants; we also treat subsession termination as a similar synchronization.

A participant behavior starts from the initial state of its normal activity CFSM. The suspicion action @ may switch the monitor (e.g., m) to its failure handling CFSM from any state of the normal activity, including end (i.e., state 5) if the subsession is not yet terminated. The other subsession participants (e.g., wD) switch to their failure handling CFSMs on receiving the initial message (i.e., a failure notification) in those CFSMs; the one exception is the monitored participant, whose failure handling is empty. See the supplement for all local types and CFSMs for Session-CM.

Fig. 4 (top) thus represents the behavior of m. In the normal activity (a), m first performs a send to, then a receive from, wD. It then selects (by some internal decision) between the spawn loop (i.e., sending Ack to Workers and spawning RunEx) or the terminal case. In the failure handling (b), m multisends the failure notification, then joins the spawning of the replacement RunDr subsession. This instance, Masters has no explicit generic behavior (unlike Workers).
**Scala API generation.** For practical programming, we implement our theory as a toolchain in Scala. Our toolchain uses the CFSM representation of each local type to generate a *type-directed API* [Hu and Yoshida 2016] for MPST-based EDP. The basic idea is: for each state, it generates a state-specific *channel type*; and for each transition, it generates a transition-specific I/O method on the associated channel type. Each I/O method takes the arguments, if any, required by the action (e.g., a message), and returns the channel type for its successor state in the CFSM; input methods additionally return the received message. For each transition, it also generates a transition-specific *singleton type* to serve as a type-level identifier for that event, which we refer to as *event types*.

Fig. 4 (c) summarizes the key API elements generated for $m$ in RunDr, i.e., from the CFSM pair in Fig. 4 (top). Select and branch states (including the unary cases) have methods named $!$ and $?$, respectively. The single method of a spawn or suspicion state is named init and failure, respectively. The corresponding event type is given next to each I/O method signature. To simplify this presentation, we assume message labels are unique across the subprotocol, and event type values are referred to by same name as their type. We illustrate how a programmer may use this API next.

### 3.2 MPST-based Event-driven Programming in Scala

MPST theory has primarily been developed in the context of the $\pi$-calculus, where the model of concurrency centers on parallel composition of sequential processes. From a practical perspective, it is akin to *multithreaded* programming with message passing, where each user process (or thread) calls the runtime system to perform its I/O (and may be blocked until the action is completed).

This paper proposes a novel *event-driven* model of concurrency for multiparty sessions. It is characterized by an *inversion of control*: the user relies on the runtime system to monitor a set of concurrent event sources (e.g., channels), and dispatch the processing of each event occurrence (e.g., message arrivals) by *calling back* the appropriate routines in the user code; user code is never blocked because it is only called when the event is ready. Our approach is inspired by the fact that EDP is a major paradigm for distributed programming in practice; it is used to implement applications featuring all the features we have discussed, including the Spark CM. As we shall see, event-driven concurrency is key to realizing our framework by unifying the handling of all the “regular” session I/O and failure events under a uniform session-typeable programming interface.

The central abstractions in EDP are the *event loop* and *event handlers*. In our setting, each participant process is a session event loop that monitors a set of (sub)session channels for I/O events and dispatches each by firing its corresponding session-typed event handler.

**Session-typed event handlers.** Our framework provides a library function for creating handlers:

$$\lambda(event1, event2) \{ \text{case } (s, c: ChanType) \Rightarrow /* \text{ handler body: return the channel after event2 */ } \}$$

The above is a function call that takes three arguments. We have named it $\lambda$ here to assist comparison with our formal notation (Sec. 4.3). In parentheses are values of the singleton event types. The first specifies the immediate event type to be handled (i.e., the event that triggers the firing of this handler). The second specifies the last I/O action that this handler will perform before handing the resulting channel back to the event loop for monitoring. We refer to these as the *initial* and *return* events of this handler. The return event must be reachable from the initial in the CFSM, and every action after the initial must be non-blocking (i.e., output). A handler that performs only an initial I/O action specifies the same event for both.

The third argument is the handler function itself, a partial function written $\{ \text{case } (s, c) \Rightarrow \ldots \}$ in Scala. Its parameters $(s, c)$ are supplied by the framework when the event loop calls this function to handle the specified event: $c$ is the subsession channel on which the event has occurred and is ready to be handled, and $s$ is just a plain data object for storing and retrieving data across separate handlers. Guided by static Scala typing on the channel types, the user implements the handler body...
to handle the initial event and perform any subsequent I/O actions up to and including the return event. The framework and generated APIs are designed such that static type checking ensures: (i) the type of the channel parameter $c$ corresponds to the initial event, and (ii) the handler returns a pair $(s_2, c_2)$ back to the event loop, where $s_2$ is the (possibly updated) data object and $c_2$ is a channel whose type corresponds to the return event, i.e., the channel yielded by the last I/O method performed. As an example of (ii), the $\text{SndInditDr}$ handler in Fig. 4 (d) returns the pair of $s$ unchanged, and the channel returned by performing $c!\text{InitDr}(\ldots)$ which is of type $M_2$. The type annotations on the channel variables $c$ in Fig. 4 (d) are optional.

Fig. 4 (d) demonstrates a set of minimal handler implementations for $\text{Masters}$ in $\text{RunDr}$ (Fig. 2); in general, handlers may also contain other arbitrary Scala code. The first handler performs the single output action of its sole event. Its channel parameter type $M_1$ corresponds to its initial event $\text{SndInitDr}$; static typing ensures we perform $!$ with the message $\text{InitDr}(\ldots)$ (payload arguments omitted) and return the resulting channel to satisfy the return event. The second handles its input event similarly. The $(\text{SuswD}, \text{SndFailDr})$ handler is an example of performing multiple I/O actions in one handler. In the $\text{SndAddEx}$ handler, we use this Scala syntax for partial functions:

\[
\begin{cases}
\text{case} \ (s, c) \ \text{if} \ /\!\!/\text{cond} /\Rightarrow \ldots
\end{cases}
\]

Scala checks the additional if clause at run-time before allowing the function to be applied; i.e., this $\text{SndAddEx}$ handler is only fired if $s.\text{workRemaining}()$ is true then.

Session event loops. A user constructs an endpoint program by registering event handlers like the above with the event loop runtime provided by our framework as a library. The handlers must cover all possible events that the endpoint may participate in according to the protocol. Our theory checks such event coverage, and that each handler is non-blocking apart from the initial event, as part of static type checking (Sec. 6.1). Our prototype implementation checks these dynamically when the list of handlers is registered (i.e., when the endpoint is first initialized); however, these are simple to check and would be straightforward to implement statically, e.g., as a compiler plugin.

The event loop library uses a lightweight internal representation of the CFMSs (built by the API generation) to track the current state of the local endpoint in each of its subsessions at run-time, and thereby determine which handlers are eligible for firing. The current state is the only precondition for firing an output handler (i.e., outputs are non-blocking). In addition to the current state, the event loop fires an input handler when a message is available in the input queue. A spawn handler is fired when the sub-session initiation synchronization is complete. Following our theory, our prototype separates the mechanism for failure detection from the event loop itself. The former may be implemented in various ways (e.g., heartbeats); the latter simply reacts to a failure suspicion event by switching to the failure handling CFMS and firing the suspicion event handler (e.g., $\text{SuswD}$).

### 3.3 Failure Model and Properties

We make the following practical assumptions on distributed systems: asynchronous system – there are no upper bounds on processes’ relative speeds or message transmission delays [Fischer et al. 1985]; crash(-stop) failures – processes can fail by halting prematurely, except robust roles – roles played by failure-resilient processes, one being minimally required for our (progress) properties to hold; reliable FIFO communication – messages between non-failed processes are received eventually, in the order sent [Basu et al. 1996]; monitors – failures of non-robust roles are detected by designated (possibly failure-prone) peers, with the possibility of false suspicions – inevitable in the above model [Chandra and Toueg 1996]. Note that suspicions are not reverted.

The next sections present a formal model of our framework as a novel event-driven formulation of a multiparty session $\pi$-calculus, and establish the properties that it guarantees. Formally, a well-typed system – i.e., a system of multiple, dynamically spawned concurrent subsessions – is guaranteed to enjoy communication safety and global progress. In short: a participant will never
receive a message for which it does not have a handler, and the system as a whole will never be stuck (some subsession can always progress), despite processes concurrently engaged in multiple subsessions. Prior session type systems offer progress only when individual participants engage in a single session; e.g., the “simple” condition of Denielou et al. [2012, p. 30, item 1] and Demangeon and Honda [2012, Prop. 3] (e.g., consider two well-typed processes \(p_1, p_2\) in two sessions \(s_1, s_2\), where \(p_1\) first receives on \(s_1\) but \(p_2\) on \(s_2\) – deadlock). (We note Coppo et al. [2016] study global progress using an additional interaction type system on top of MPSTs.) The intuition for global progress in our framework is that event-driven processes by definition never engage in an I/O action unless the event is ready for processing, and this characteristic is independent of the number of active (sub)sessions. Second, our framework ensures that a protocol is safely structured (Sec. 4.1) in both “regular” interactions and handling of all potential participant failures, and every participant process safely provides compliant handlers for every potential event it may engage in (Sec. 6.1).

Using our Scala toolchain, the generated API types ensure communication safety via the native Scala type system modulo linear usage of each channel instance, i.e., the user should call exactly one I/O method on each channel instance in a handler until the return event is reached. Our theory integrates linearity into the static type system, as standard in session types. Our implementation checks linearity at runtime [Castro-Perez et al. 2019; Padovani 2017] and can safely treat violations as failures. Note, this is not a fundamental limitation (it is straightforward to adjust the API design so channels are never directly exposed to the user, e.g., [Zhou et al. 2020]), but we opt for the current style to correspond with our formal presentation. Regarding progress: since a user may include arbitrary Scala code within handlers and the if-clause of partial functions, the practical progress guarantee is modulo termination of handler code and exhaustiveness of the relevant handler if-conditions. Regardless of linearity and progress, however, our toolchain guarantees a well-typed Scala endpoint program will never perform session I/O that is not compliant with the source global protocol, other than premature termination due to failure.

4 GLOBAL TYPES, LOCAL TYPES AND EVENT-DRIVEN SESSION PROCESSES

4.1 Global Types for Specifying Fault-Tolerant, Multiparty Protocols

Fig. 5 defines the syntax of global types. Message labels range over \(l, l_1, l_2\),... Type variables range over \(t, t_1, t_2\),... Subprotocol names range over \(g, g_1, g_2\),... A role set represents a set of participant processes that support a common behavior, e.g., a set of worker processes. A role represents a participant drawn from a role set to fulfill some additional individual behavior, e.g., a worker that runs a Spark driver. Role set names range over \(R, R_1, R_2,\ldots\), and role names range over \(r, r_1, r_2,\ldots\). We may write \(r \in R\) to clarify that \(R\) is the role set of \(r\). A role set \(R\), always refers to all participant processes in the set, whereas a corresponding role \(r\) refers to one of these participants. An overbar, e.g., \(\overline{R}\) denotes a sequence, and may be empty unless explicitly stated otherwise. In examples, role set names start with an uppercase letter and roles with a lowercase.

A top-level global type \(G\) describes a complete protocol comprising a set of distinctly named subprotocols. \(g(F, r; \overline{R}) = G_1\) with \(r @ r’\). \(G_2\) defines a subprotocol, named \(g\), between the participants given by the three parameter segments: (1) a non-empty sequence of distinct roles \(F\); (2) a role \(r\)
dynamically assigned from \( R (r \in R) \) when \( g \) is spawned; and (3) a sequence of distinct role sets \( \bar{R} \). Note, a role \( r \) and its role set \( R \) may both be present, meaning the participant assigned to \( r \) fulfills both behaviors concurrently; e.g., the participant assigned to \( w \in RunDr \) (see Fig. 2) behaves as \( w \in \text{Workers} \) and a member of \( \text{Workers} \). Multiple roles \( r_1, r_2 \) may be from the same role set \( R \). As mentioned, restricting a subsession to one role assignment simplifies the presentation; multiple assignments can be expressed via nested subsession spawning. \( G_1 \) is the normal activity of \( g \), whereas the shaded parts relate to failure handling. \( r \oplus r' \) specifies that \( r' \) is the monitor of the assigned \( r \), and \( G_2 \) is the corresponding failure handling activity (or simply, failure handling); \( r \) is not permitted to occur in its own failure handling. Note: our formalism assumes the message labels (resp. subprotocol names) in the normal activity are disjoint from those in the failure handling.

A spawn type \( g(\bar{r}, R; \bar{R}) \). \( G \) specifies spawning a new subsession instance of subprotocol \( g \) that runs concurrently with the continuation \( G \). The spawn arguments are in three segments corresponding to the parameters of \( g \): non-empty, distinct roles \( \bar{r} \); a role set \( R \) from which to assign a participant; and distinct role sets \( \bar{R} \). Spawns in failure handling and recursive spawns are permitted. A branch type \( r \rightarrow z \{ l_i : G_i \}_{i \in I} \) specifies that \( r \) decides which case the protocol will follow by sending a message labeled \( l_i \) to another role \( z = r' \) or by multisending messages labeled \( l_i \) to a role set \( z = R' \); the protocol then proceeds according to \( G_i \). Following MPSTs tradition, for the former we assume \( r \neq r' \) (note, no two roles in one subsession are ever played by the same participant); for the latter, we assume \( R' \neq R \), where \( r \in R \). We may write \( r \rightarrow z l \cdot G \) for short when \( l \) is a singleton. \( \mu t \cdot G \) and \( t \) allow recursive types (our model adopts an iso-recursive approach). We assume type variable occurrences are bound and guarded in the usual way (e.g., \( \mu t \cdot t \) is not permitted). Lastly, \( \end \) denotes subprotocol termination (often elided for brevity).

Example 4.1. The top-level global type \( G_{\text{CM}} \) corresponding to Fig. 2 comprises two subprotocols:  

\[
\begin{align*}
g_{\text{RunDr}}(m; w_D; W) &= m \rightarrow w_D l_{\text{InitDr}} \cdot w_D \rightarrow m l_{\text{Ack}} \cdot \mu t \cdot m \rightarrow W \{ l_{\text{AddEx}} : g_{\text{RunEx}}(m, w_D; W; W) \cdot t \}, l_{\text{Ok}} : \text{end} \} \quad \text{with} \quad w_D @ m \cdot m \rightarrow W l_{\text{FailDr}} \cdot g_{\text{RunDr}}(m; w_D; W) \cdot \text{end} \\
g_{\text{RunEx}}(m; w_D; w_E; W) &= m \rightarrow w_E l_{\text{InitEx}} \cdot w_E \rightarrow m l_{\text{ExDone}} \cdot m \rightarrow w_D l_{\text{ExFinished}} \cdot \text{end} \\
&\text{with} \quad w_E @ m \cdot m \rightarrow W l_{\text{FailEx}} \cdot g_{\text{RunEx}}(m, w_D; W; W) \cdot \text{end}
\end{align*}
\]

Well-formedness. Following the standard approach in MPSTs, we impose basic constraints to prevent global types that are not safely realizable as distributed protocols. (1) As standard, our system considers only global types for which projection (Sec. 4.2) is well-defined for every participant kind: a key point is to ensure consistency of multiparty branching, which leads to the next condition. (2) We require the \( G \) in a monitor term \( r_1 @ r_2 \cdot G \) to start with the monitor \( r_2 \) sending some label (i.e., failure notification) to all other roles (except the monitored role \( r_1 \) and role sets in the subprotocol. This ensures that the monitor soundly informs all other participants if it suspects a failure. E.g., in both failure handlings in Ex. 4.1/Fig. 2, the monitor \( m \) informs all workers \( W \) about the failure (note, \( w_D \in W \)). The supplement provides the straightforward formal definition of well-formedness.

4.2 Local Types and Endpoint Projection

Fig. 5 defines the syntax of local types, which represent the view of a protocol localized to one kind of participant, i.e., the combined view of a role set and its role assignments. Local types are used to statically ensure that a participant process (i.e., an event loop) safely implements the full behavior of its role set and potential role assignments. Parts related to failure handling have darker shading, and runtime terms (i.e., only occurring during execution) are lightly shaded; some terms are both. As explained in Sec. 4.3 and 5, our formalism also reuses local type syntax as an artifact for recording the current state of subprotocols at runtime, to drive event loops and handler firing.

Participant names (i.e., process identifiers) range over \( p, p_1, p_2, ..., q, q_1, q_2, ... \). Meta symbol \( u \) stands for a role \( r \) or role set \( R \); and \( y \) for a role \( r \), or the additional runtime annotation \( p : r \) specifying that
participant $p$ is playing role $r$ in a given subsession. E.g., assume participants $\{p_1, \ldots, p_n\}$ constitute a worker role set $W$ and the protocol involves a role $w \in W$. Although any of $\{p_1, \ldots, p_n\}$ may play $w$, the specific $p_j$ picked to play $w$ at runtime is recorded by the annotation $p_j:w$.

A top-level local type $L$ describes a complete localized protocol comprising a set of local subprotocols, as yielded by projection (explained shortly). $g_r(\bar{r}; r; \bar{R}) = L$ defines the local subprotocol view of $g$ from the annotated $z$, i.e., a local role or role set. $g_r(y; R; \bar{R}).L$ is a local spawn type for subprotocol $g$. The three argument/parameter segments are as explained for their global type counterparts: non-empty, distinct roles $y$ to play $r$; a role set $R$ from which to assign a participant to play $r$; and distinct role sets $\bar{R}$. The send/select type $u!\{l_i : L_i\}_{i \in I}$ specifies: select and (multi)send $l_i$ to $u$, then proceed as $L_i$. Receive/branch $y?\{l_i : L_i\}_{i \in I}$ is the dual: receive an $l_i$, then proceed as $L_i$; note, we receive only from a role (not a role set). $L_i$ with $L_2$ represents a normal activity $L_1$ under execution and the as yet inactive failure handling $L_2$. For the monitor, $L_2$ is (initially) a suspicion type $y\downarrow L$. $L$ specifying: monitor $y$ and switch to $L$ (i.e., switch from the normal to the failure activity) upon the suspected failure of $y$. There is no dual to $y\downarrow L$ because failing is an implicit action that can occur at any time. $-\text{with} L$ represents an active failure handling, and only occurs at runtime. $\mu t.L$ and $t$ are for recursive types. end (often elided) represents the termination of a normal or failure handling activity. A stopped type $\text{end}_E$ represents a behavior that is (prematurely) halted due to failure handling being activated in some ancestor subsession, causing all child subsession activities to be halted. The $L$ annotation specifies the point at which the behavior was halted.

We use a simple context notation $\mathcal{E}[^\cdot]$ for local types (defined: $\mathcal{E} ::= [] \mid \mathcal{E}$ with $L \mid -\text{with} \mathcal{E}$) to access a normal or active failure handling activity in with terms, e.g., in reduction rules (Sec. 5).

**Projection.** Our typing statically associates every participant with a role set. However, a safe implementation of a participant process must also be capable of all potential role assignments. Our notion of endpoint projection [Coppen et al. 2016; Honda et al. 2016] is thus the projection of a top-level global type onto a role set together with the projections onto each of its role assignments. E.g., for the subprotocol $g_{\text{RunEx}}$ on Ex. 4.1/Fig. 2, the top-level local type of a workers participant includes multiple local subprotocols: $g_{\text{RunEx}_w} \cdot g_{\text{RunEx}_{w'}} \cdot g_{\text{RunEx}_{w''}}$.

**Definition 4.2** (Projection $G \uparrow R = L$). The projection of a top-level global type $G$ onto a role set $R$ is

$G \uparrow R = \{g \uparrow R \mid g(\bar{r}; r; \bar{R}) \in G \land R \in \bar{R}\} \cup \{g \uparrow r \mid g(\bar{r}; r; \bar{R}) \in G \land r \in \bar{R} \land \bar{R} \cdot r \cdot \{\}\}$

The LHS of the union projects subprotocol onto role set $R (g \uparrow R)$ if $g$ uses $R$. The RHS projects $g$ onto role $r_2 (g \uparrow r_2)$ if $g$ uses $r_2$. We define $g \uparrow r$ below. We leave the definition of $g \uparrow R$ to the supplement; it is very similar but simpler than $g \uparrow r$.

**Definition 4.3** (Projection $g \uparrow r = g_r$). The projection of a subprotocol $g$ onto a role $r$, and the projection of a global type $G$ onto a role $r$ (i.e., $G \uparrow r = L$), are respectively defined:

\[
g \uparrow r = g_r,
\]

\[
(g(\bar{r}; r_1; \bar{R}) = G_1 \uparrow r_1 @ r_2 G_2) \uparrow r = \begin{cases} g_r(\bar{r}; r_1; \bar{R}) = G_1 \uparrow r \text{ with } r_1 \downarrow G_2 \uparrow r & \text{if } r = r_2, \\
\mu g(\bar{r}; r_1; \bar{R}) = G_1 \uparrow r \text{ with } G_2 \uparrow r & \text{if } r = r_1, \\
\mu g(\bar{r}; r_1; \bar{R}) = G_1 \uparrow r \text{ with } \text{end} & \text{if } r = r_1, \\
\end{cases}
\]

\[
G \uparrow r = L,
\]

\[
r_1 \downarrow z \{l_i : G_i\}_{i \in I} \uparrow r = \begin{cases} z(\{l_i : G_i \mid r\}_{i \in I}) & \text{if } r = r_1, \\
\text{else if } z = r \lor r \in z, \\
\text{else if } \forall i, j \in I. G_i \uparrow r = G_j \uparrow r
\end{cases}
\]

\[
g(\bar{r}; R; \bar{R}) \cdot G \uparrow r = \begin{cases} g(\bar{r}; R; \bar{R}) \cdot G \uparrow r & \text{if } \bar{r} \in \bar{R} \land \exists R' \in \{\bar{R} \cdot r \in R', G \uparrow r \} \text{ otherwise } \\
\mu \cdot G \uparrow r & \text{if } G \uparrow r \neq t \text{ end otherwise }
\end{cases}
\]

The top rule projects the normal activity of a subprotocol onto $r$, and projects the failure handling according to $r$’s involvement in monitoring. If $r$ is the monitor, the projection of $G_2$ is prefixed by a monitor type $r_1 \downarrow$. If $r$ is neither the monitor nor the monitored role, then we just project $G_2$. If $r$
is the monitored role, then failure handling is set to end − r is not involved in the handling of its own failure! We explicitly project the failure handling of the monitored role r as end, even if the role set R of r occurs in the global failure handling. The rule below the dashed line is the standard projection (e.g., Coppo et al. [2016]) of an interaction according to the role’s involvement, with the additional clause for a multisend to the role set R of r (via the condition r ∈ z) – the latter projects to a receive because the participant playing an assigned role r is still a member of R. The next rule projects a spawn term and its continuation onto r: the spawn is retained if r or the role set of r occurs in the arguments. The recursion and termination rules are standard.

Example 4.4. The projection G_{CM} ⊢ M = \{ g_{RunDr} \mid m, g_{RunEx} \mid m \} (i.e., both as role m), yielding:

\[ g_{RunDr}(m; w_D; W) = w_D!Ι_{InitDr} \cdot w_D!Ι_{Ack} \cdot m \cdot W' \] with \[ w_D!Ι_{AddEx} : g_{RunEx}(m, w_D; W; W') \cdot m, w \cdot \] end

\[ g_{RunEx}(m, w_D; W) = w_D!Ι_{InitEx} \cdot w_E!Ι_{ExDone} \cdot w_D!Ι_{ExFinished} \cdot end \] with \[ w_E!Ι_{ExFail} : g_{RunEx}(m, w_D; W; W) \cdot end \]

Our toolchain represents local types as CFSMs as explained in Sec. 3.1; simply put, each action in the local type is a CFSM transition, and the normal and failure handling activities are represented as separate CFSMs. E.g., \( g_{RunDr} \) is represented by the CFSM pair in Fig. 4.

4.3 Event-Driven Session Processes and Networks

Fig. 6 (top) defines the syntax of participant processes (or simply processes), denoted by P, P_1, P_2, ..., and networks, denoted by N, N_1, N_2, .... Failure handling terms have darker shading, and runtime terms (i.e., that arise only during execution) are lightly shaded; some are both. (Sub)session names range over s, s_1, s_2, .... Meta symbol c denotes a session channel (or simply, channel) s[p], or a channel variable ranging over x, x_1, .... We write ˜x for a potentially empty set \( \{x_i\}_{i \in I} \).

The heart of a participant process is an event loop and its event handlers (or simply, handlers). (\( H, p \)) is an event loop for participant p with handlers H. As discussed in Sec. 5, an event loop encapsulates and monitors a set of channels for I/O events, and handles each event occurrence by firing a handler, which is executed to completion before the event loop resumes. The handler is dynamically selected according to the event, e.g., the message received, and the current state of the participant in the subprotocol for this subsession. As mentioned, every participant p is statically associated with a role set R: we write R_{ids} for the set of all participants associated with R.

A handler \( [L] \lambda x . P \) defines a session computation. Channel variable x is substituted for the event-ready channel when the handler is fired, P is the handler body, and guard type L specifies the initial event type and I/O behavior to perform on x in P. For comparison, a handler is written using our toolchain (Sec. 3.2): \( λ(/{*guard types/*})(case \ldots x \Rightarrow {/*p*/})\). (The “initial” and “return” events there correspond to our single guard type L here.) The idea is that a handler is only fired if it will not block – P may contain a blocking action like receive, but is only fired if the expected message is present. To this end, our type system (Sec. 6.1) restricts the guard type to use: (a) at most one blocking action, which must be the initial action; (b) no recursion; (c) no branching or selection, i.e., be flat. These restrictions are natural to EDP and help ensure progress without limiting expressiveness: branching and selection are expressed in our system via separate handlers (see Ex. 4.5), and recursion manifests as recurring events and repeat handler firings (see Ex. 5.4).
The other $P$ terms occur inside handler bodies. $c[u]!\tau.P$ performs a \texttt{(multi)send} of label $l$ on channel $c$ to a role or role set $u$, followed by $P$; $c[r]!\tau.P$ is the \texttt{receive} counterpart. For simplicity, we do not model message payload types/values. A \texttt{spawn} $c[R;R;\bar{R}](g)$ uses channel $c$ to initiate a new subsession for $g$. The participants joining the new subsession from the current subsession are: roles $\bar{R}$, a participant assigned from $R$, and role sets $\bar{R}$. The action by a monitor signifying it suspects $r$ has failed and will switch to $P$. loop denotes the completion of a handler execution and resumption of the event loop: loop is substituted for the original event loop term when a handler is fired, hence we distinguish it from the typical 0 term. We may omit loop in examples; e.g., we write $\lambda x \cdot x[D]!\text{InitDr}$ with the trailing .loop omitted.

Example 4.5 (Event-driven selection). Let $\bar{H}, p$ be an event loop for a participant of role set $M$ from Ex. 4.1/Fig. 2 with participant identifier $p$. $\bar{H}$ contains, e.g., the handlers $[W!\lambda x. x[W]!\lambda \bar{D}!\lambda \bar{D}!\lambda \bar{D}]$ and $[W!\lambda x. x[W]!\lambda \bar{D}!\lambda \bar{D}]$ – cf. the $\text{SndAddEx}$ and $\text{SndOk}$ handlers in Fig. 4 (d); similarly for the others (see the supplement). These two handlers implement the \textit{selection} from $\text{runDr}$, i.e., the selection $\text{W'} \{ \text{AddEx} : \ldots, \text{Ok} : \ldots \}$; cf. $g_{\text{runDr}}(m; w; D; W)$ in Ex. 4.4. The first handler covers the $l_{\text{AddEx}}$ choice, and the second covers $l_{\text{Ok}}$. Compared to Fig. 4, our formalism omits the $if$-clause – we abstract from the internal decision procedure and simply model it non-deterministically.

Network $N || N$ is a standard parallel composition, for composing participant event loops and configurations into a complete session system. We write $\parallel p \in p (N)$ for a composition of $N$ terms for each $p$. For each process $P$ in a network, we assume all terms within $P$ relate to a single participant $p$. In our system, restriction $(\text{vs} : G) N$ is applied only to the root session, to annotate the $G$; for simplicity, we assume subsessions are always created with fresh names. An endpoint configuration (or simply, \textit{configuration}) $s[p] : (L, b)$ records the runtime information used by participant $p$ to execute its part in subsession $s$: a local type $L$, called the current subprotocol state, and an asynchronous input buffer $b$, which holds a FIFO per subsession peer for incoming messages. An empty buffer is written $\epsilon$. The subprotocol state serves two purposes: to designate the next set of events expected in subsession $s$ and thereby direct (in conjunction with the event occurrence) which handler to fire, and to record the binding of (peer) roles to participant names. We opt to represent the notion of subprotocol state by reusing local type syntax; however, this mechanism may be implemented in various ways (e.g., we use an FSM), and we emphasize our system is fully \textit{statically} typed.

Example 4.6. Take the event loop from Ex. 4.5, for an $M$ participant $p$, and two configurations $s[p] : (E[p_1 ; w_D]^l_{\text{InitDr}} \cdot p_1 ; w_D? l_{\text{ok}} . L'], b_1)$ and $s[p] : (E[p_1 ; w_D]^l_{\text{InitDr}} \cdot L', b_2)$ for some $L'$. The event loop can fire the handler $[w_D]!l_{\text{InitDr}} \lambda x . x[w_D]!l_{\text{InitDr}}$ under the first configuration, but not the second. The subprotocol state in the first (resp. second) expects to send $l_{\text{InitDr}}$ (resp. to receive $l_{\text{ok}}$).

The remaining items are runtime environments for process reduction: $\Theta$ is the monitoring tree, i.e., the parent-child subsession relation. $\mathcal{F}$ is the fail set, which records all suspected participants. We can model these environments in a global fashion since their decomposition into local views (as implemented in our prototype) is straightforward and has no significant impact on the overall design. Regarding $\Theta$, a participant is concerned only with the subtree pertaining to the subsessions itself is involved in: a lookup in a local or a global view is no different. Regarding $\mathcal{F}$, as we shall see, participants are only ever added to $\mathcal{F}$ (i.e., crash-stop failures): it is straightforward (even in an asynchronous system) to have participants converge on a consistent view, which is all we require.

5 OPERATIONAL SEMANTICS FOR EVENT-DRIVEN CONCURRENT SUBSESSIONS

5.1 Overview
We define a \textit{reduction} relation on systems of the form $(\Theta, \mathcal{F}, N)$. Relation $(\Theta_1, \mathcal{F}_1, N_1) \rightarrow (\Theta_2, \mathcal{F}_2, N_2)$ reduces a network $N_1$ under environments $\Theta_1$ and $\mathcal{F}_1$ to $N_2$ and updated environments $\Theta_2$ and $\mathcal{F}_2$. 
Failure handling.

Failure handling.

We streamline our formalism by assuming the root subsession to be already established in an initial system: this allows us to omit the usual "shared channel" prefixes used to bootstrap a session – that aside, our model freely supports concurrent subsession spawning, based on the standard session initiation in MPSTs. We present the reduction rules in three parts: event loops and handler firing, subsession spawning, and failure handling. On notation: we (1) omit $\Theta$ and/or $\mathcal{F}$, for brevity, from a rule definition if they are neither used nor modified; (2) write ‘.’ to denote an irrelevant element.

We continue with the session-typed CM (Session-CM) as running example. Fig. 7 illustrates the system over a few reduction steps in a scenario where the driver participant fails after a $g_{\text{RunEx}}$ subsession has already been spawned: Fig. 7 (a) shows the initial system from Ex. 5.1; Ex. 5.5 demonstrates spawning the $g_{\text{RunEx}}$ protocol, leading to (b); and in Ex. 5.6, the participant playing $w_D$ fails and $g_{\text{RunDr}}$ is restarted, corresponding to (c) and (d).

**Example 5.1.** Fig. 7 (a) depicts an initial system for Session-CM. This root subsession $s$ involves participants $p_1$ in role set $M$ and $p_{1,3}$ in $W$: the monitoring tree is $\Theta_0 = \{s \mapsto \{p_1,p_2,p_3,p_4\}\}$. No failures have occurred yet: the failure set is $\mathcal{F}_0 = \emptyset$. This system is thus $(\Theta_0, \mathcal{F}_0, N_0)$, where $L = \{1,4\}$, $N_0 = (vs: G_{\text{CM}}) (\Pi_{i\in L} (H_i, p_i) \Pi_{i\in L} s[p_i]: (L_i, e))$ and

\[
L_1 = p_2: w_D! \text{InitDr} \cdot p_2: w_D? l_{\text{Ack}}, \mu t. w! \{l_{\text{AddEx} : g_{\text{RunEx}} (p_1 : m, p_2 : w_D; W ; W ); \tau, l_{\text{Ok}} : \text{end}\} \text{ with } ...
\]

\[
L_2 = p_1 : m? l_{\text{InitDr}} \cdot p_1 : m! l_{\text{Ack}}, \mu t, t \cdot p_1 : m? \{l_{\text{AddEx} : g_{\text{RunEx}} (p_1 : m, p_2 : w_D; W ; W ); \tau, l_{\text{Ok}} : \text{end}\} \text{ with } ...
\]

\[
L_k = \mu t, t \cdot p_1 : m? \{l_{\text{AddEx} : g_{\text{RunEx}} (p_1 : m, p_2 : w_D; W ; W ); \tau, l_{\text{Ok}} : \text{end}\} \text{ with } ...
\]

$k \in \{3,4\}$

The $(vs: G_{\text{CM}})$ specifies that $s$ is an instance of the root subprotocol $g_{\text{RunDr}}$. Ex. 4.5 gave an example of $H_1$, i.e., a handler list for an $M$ participant; $H_{2,4}$ may be implemented similarly. Every configuration has an empty buffer. The initial subprotocol state $L_1$ of $p_1$ is essentially the projection of $g_{\text{RunDr}}$ onto $m$ annotated with the runtime bindings of roles to participants, i.e., $p_1$ is playing $m$ (i.e., $p_1 : m$) and $p_2$ is playing $w_D$ (i.e., $p_2 : w_D$); similarly for $L_{3,4}$. Subprotocol state $L_2$ differs from $L_{3,4}$ because $p_2$ has been assigned to $w_D$ whereas $p_{3,4}$ are still generic members of role set $W$. That is, $L_2$ describes the combined behavior of $w_D$ and $W$, whereas $L_{3,4}$ describes $W$ only.

### 5.2 Event Loops and Handler Firing

The main reduction rules are defined in Fig. 8–9. We leave the standard structural rules (for parallel composition, restriction and structural congruence) to the supplement. Fig. 8 gives the core rules for event loop and handler execution. Fire is the key rule for the event loop $(H, p)$ of participant $p$ to fire a handler $[L_2] \lambda x. P$. Premise $L_1 \vdash L_2$ checks that the current subprotocol state $L_1$ of $s$ matches the handler’s guard type $L_2$, and $\text{fire}(L_1, L_2, b, \mathcal{F})$ checks that for the required runtime information (e.g., the expected message) to ensure the handler will not block. The event loop term is then replaced by the handler body $P$ with the session channel substituted for $x$ (written $\{s[p]/x\}$), and the event loop itself substituted for loop, to return control back to the event loop after the handler is completed. When in this form, we say the event loop has an active handler.

---

Proc. ACM Program. Lang., Vol. 5, No. OOPSLA, Article 124. Publication date: October 2021.
An MPST Discipline for Fault-Tolerant Event-Driven Distributed Programming 124:17

(\Theta, \mathcal{T}, N) \rightarrow (\Theta', \mathcal{T}', N')

\text{Fire} \quad \begin{aligned}
\frac{[L_2]!x. P \in H}{\mathcal{L} \times L_2} & \quad \text{fire}(L_1, L_2, b, \mathcal{F}) \quad N = s[p]: (L_1, b) \\
& \quad (\mathcal{F}, (H, p) \parallel N) \rightarrow (\mathcal{F}, P(s[p]/x)((H, p)/\text{loop}) \parallel N)
\end{aligned}

j \in I \quad b'_q = b_q[p \mapsto b_q(p) \cdot l_j]

s[p][r]!l_j.P \parallel s[p] : (\mathcal{E}[q;r/l_i : \{l_i\}_{i \in I}]_\ldots) \quad || s[q] : (_, b_q) \rightarrow P \parallel s[p] : (\mathcal{E}[L_j]_\ldots) \quad || s[q] : (_, b'_q)

(q \rightarrow l_j, l_i) \in b \quad L = \mathcal{E}[R!\{l_i : L_i\}_{i \in I}] \quad \bar{p} = R_{\text{ids}} \setminus \mathcal{F}

j \in I \quad b' = b[q \mapsto \bar{l}]

s[p][r]!l_j.P \parallel s[p] : (\mathcal{E}[q;r/l_i : \{l_i\}_{i \in I}]_b) \rightarrow P \parallel s[p] : (\mathcal{E}[L_j], b')

\text{SEND}

L = \mathcal{E}[R!\{l_i : L_i\}_{i \in I}] \quad \bar{p} = R_{\text{ids}} \setminus \mathcal{F}

j \in I \quad \forall q \in \bar{p} \quad b'_q = b_q[p \mapsto b_q(p) \cdot l_j]

\text{Recv}

\frac{(\mathcal{F}, s[p][R]!l_j.P \parallel s[p] : (\mathcal{E}[L_j]_\ldots) || q \notin \bar{p} s[q] : (_, b_q))}{(\mathcal{F}, P \parallel s[p] : (\mathcal{E}[L_j], _) || p \notin \bar{p} s[q] : (_, b'_q)}

\text{MSND}

L - L' = L''

with \( L' - L_g = L' \) with \( L' - L = L' \)

\( L - L_g = L' \) with \( L - L' = L' \)

\( L - L'' = L' \)

\( L' - L = L'' \)

\( L' - L'' = \)

\( I \in \{l_i\}_{i \in I} \quad L_j - L_g = \ L_j - L'' \quad I \in \{l_i\}_{i \in I} \quad y \equiv r \quad L_j - L_g = \ L_j - L'' \)

\( Y \in \{l_i\}_{i \in I} \quad r \equiv r \quad L_j - L_g = \ L_j - L'' \)

\( g(\bar{y}, R, \bar{R}).L - g(\bar{r}, \bar{R}, \bar{R}).\text{end} = L \quad L - \text{end} = L \quad (y \equiv r \text{ means } y = p:r \vee y = r)

Fig. 8. Event loops and handler firing. (We omit \( \Theta \) and/or \( \mathcal{F} \) where irrelevant; \( N \) is always present.)

The \textit{match} predicate \( L \times L' \) holds when local type \textit{subtraction} \( L - L' \) is defined for these types. (We define \( L \times L' \) via \( L - L' \) because the latter is anyway needed again later in the type system.)

**Definition 5.2 (Subtraction).** \( L \) subtracted by guard type \( L' \) \((L - L' = L'') \) is defined by Fig. 8 (bottom).

\( L - L' \) is defined when \( L' \) is a “prefix” of the normal or failure handling activity in \( L \). It yields the “remainder” of \( L \) after subtracting \( L' \), and converts it to an active failure handling if an initial failure notification is consumed. Two actions are matched if they are of the same kind and use the same roles; participant names (i.e., \( p \) in \( p:r \)) are ignored. As an example: \( p_1;r_1?\{l_1 : p_2;r_2!l_1'. L_1', L_2 : L_2'\} - r_1?l_1. r_2!l_2 = L_1' \), by consuming the \( l_1 \) and then the \( l \) within that branch case.

Predicate \textit{fire}(L1, L2, b, \mathcal{F}) \) intuitively checks whether any potentially blocking action in guard type \( L_2 \) (at the most), is executable \textit{without} blocking under the given \( b \) and \( \mathcal{F} \). The subprotocol state \( L_1 \) provides the binding of roles to participants. E.g., consider \textit{fire}(p1;r1?l1, L, b, \mathcal{F}) \) with \( L = r_1?l_1. r_2!l_2 \). The predicate holds iff \( b(p_1) = l_1 \cdot \bar{l} \); i.e., it is false if \( b(p_1) = \epsilon \) or \( b(p_1) = l' \cdot \bar{l} \land l' \neq l_1 \).

The definition of fire is straightforward and left to the supplement.

**Example 5.3 (Handler firing).** Let \( N = (H, p) \parallel s[p](L, b) \) where \( L = L_1 \) with \ldots and \( [L']!x. [r_1] !l_1. x[r_2] !l_2'. \text{loop} \in H \quad L' = r_1?l_1. r_2!l_2'. L_1 = p_1;r_1?\{l_1 : p_2;r_2!l_1'. L_2 : L_2'\}. \quad \text{Then} \ L \times L' \), as \( L' \) is a “prefix” of the normal activity \( L_1 \). Given \( b(p_1) = l_1 \cdot \bar{l} \) and some \( \mathcal{F} \), then \textit{fire}(L, L', b, \mathcal{F}) \) holds. Hence, \( N \) may be reduced by \textit{Fire} to activate this handler, i.e., \( (\mathcal{F}, N_1) \rightarrow (\mathcal{F}, s[p][r_1] !l_1. s[p][r_2] !l_2. (H, p) \parallel s[p] : (L, b)). \)

I/O actions are performed by event loops with active handlers. \textit{Send} dispatches label \( l_j \) from \( p \) to the participant playing \( r \). The subprotocol state of \( p \) must have a send type with an \( l_j \) case as its normal or active failure handling activity, and provide the \( q \) bound to \( r \); then \( l_j \) is appended to the buffer for \( p \) at \( q \). \textit{Recv} consumes a label \( l_j \) from the participant playing \( r \), if the subprotocol state of \( p \) has a corresponding receive type, and there is an available \( l_j \) from the \( q \) bound to \( r \).
\[(\Theta, F, N) \rightarrow (\Theta', F', N')\]

**SPAWN**

\[p_1 = \tilde{q} \cup ((R_{db} \cup \tilde{R}_{db}) \setminus F) \quad p_0 \in R_{db}' \setminus \tilde{q} \quad \tilde{p}_2 = \tilde{q} \cup p_0 \cup (\tilde{R}_{db} \setminus F) \quad \tilde{p}_2 \cap F = \emptyset \]

\[s_2\text{ fresh} \quad \Theta' = \Theta[s_2 \mapsto (\tilde{p}_2, 0), s_1 \mapsto (\Theta(s_1) \cup s_2)] \quad L = g(\tilde{F}, R', \tilde{R}, L_p) \quad L' = \tilde{q} R_{db} p_1 p \]

**SeSS-GC**

\[(\Theta, F, ||_{p \in \tilde{F}}((\tilde{H}_p, p) || s_1[p]:(E_p[L_p, \ldots])) \rightarrow (\Theta', F, ||_{p \in \tilde{F}}((\tilde{H}_p, p) || s_1[p]:(E_p[L_p, \ldots])) ||_{p \in \tilde{F}} s_2[p] : (L', t))\]

\[s\text{ is the root} \quad \Theta(s) = (\tilde{p}_1, 0) \quad \tilde{p} = \tilde{p}_1 \setminus F \quad s_2 \in \tilde{s} \quad \forall p \in \tilde{p}. \text{done}(L_p)\]

**ROOT-GC**

\[(\Theta, F, ||_{p \in \tilde{F}}((\tilde{H}_p, p) || s_2[p] : (L_p, \ldots)) \rightarrow (\Theta \setminus s, F, 0) \quad (\Theta, F, N) \rightarrow (\Theta, F \cup p, N)\]

**Mon**

\[q \in F \quad s \rightsquigarrow_{\Theta}(s_i)_{i \in I} \quad L_p = L' \text{ with } q r \downarrow L\]

**RcvFN**

\[(\Theta, F, s[p] \downarrow \downarrow L \cdot p \mid s[p] : (L_p, \ldots)) \quad (\Theta, F, p || s[p] : (L_p, \ldots)) \rightarrow (\Theta, F, p || s[p] : (L_p, \ldots)) \quad \text{with } L' \text{ } (\Theta, F, p || s[p] : (L_p, \ldots)) \rightarrow (\Theta, F, p || s[p] : (L_p, \ldots))\]

**Clean**

\[b(q) = l_1 \quad \tilde{l} \quad b' = b[q \mapsto \tilde{l}] \quad (L = \text{with } L' \text{ and } s \text{ in } L') \quad (\Theta, F, ((\tilde{H}, p) || s[p] : (L, b))) \rightarrow (\Theta, F, ((\tilde{H}, p) || s[p] : (L, b')))\]

Fig. 9. Subsession spawning, failure suspension and failure handling activation.

**Example 5.4 (Recursion).** Let \(N' = (\tilde{H}', p) || s[p] : (E[\mu. q r \downarrow l]. t, \ldots)\) with appropriate handlers \(\tilde{H}'\).

By unfolding the recursive subprotocol state, \(N'\) allows repeated handler firing like: \(N' \xrightarrow{\text{UNFOLD}} N'_1 \xrightarrow{\text{FIRE}} N'_2 \xrightarrow{\text{SEND}} N'' \xrightarrow{\text{UNFOLD}} N'_1 \xrightarrow{\text{FIRE}} N'_2 \xrightarrow{\text{SEND}} \ldots\) (showing only the main rules).

### 5.3 Subsessions and Spawn

In Fig. 9, **SPAWN** performs a **subsession initiation** to spawn a new concurrent instance of a subprotocol \(g\). We base it on session initiation in standard asynchronous MPSTs (e.g., [Coppo et al. 2016]), i.e., a synchronization (e.g., a multiparty handshake) that collects all the participants needed to conduct the (sub)session. **SPAWN** is straightforward but involves several aspects: (1) ensure that all participants involved in the spawn have reached their corresponding subprotocol states and perform the spawn together; (2) determine which participants join the new subsession; and (3) create the new subsession and update/create the configurations. We first illustrate (1)–(3).

**Example 5.5.** Assume a step \((F_0, \Theta_0, (vs : G_{CM}) N_1) \xrightarrow{\text{SPAWN}} (F_0, \Theta', (vs : G_{CM}) N_2)\) that spawns a new \(g_{RunEx}\) subsession. The LHS system (reached from Ex. 5.1 by some steps) may have

\[N_1 = ||_{i \in I} (\tilde{H}_i, p_i) ||_{i \in I} s[p_i] : (L_i, \epsilon) \quad L_i = g_{RunEx}(p_m, m, p_2, W_d; W; W'). L' \text{ with } \ldots\]

with \(I = \{1, 2\}\), \(F_0 = \emptyset\) and \(\Theta_0 = \{s \mapsto (p_1, p_2, p_3, p_4)\}\). For (1), **SPAWN** requires in \(N_1\) that no event loop of any spawn participant has an active handler, and all subprotocol states \((L_i)\) are at the corresponding spawn type. For (2), **SPAWN** determines the participants of the new subsession, say \(s'\), based on the above spawn arguments (inherited from \(G_{CM}\)) and target subprotocol parameters, i.e.,

\[g_{RunEx}(m, W_d; w; W).\]

Thus: (i) \(p_1\) playing \(m\) in \(s\) will play \(m\) in \(s'\), and \(p_2\) playing \(W_d\) in \(s\) will play...
\( w_D \) in \( s' \); (ii) \texttt{Spawn} assigns a participant from \( W \), say \( p_3 \), to play \( W_E \) in \( s' \); (iii) all \( W \) participants that do not play named roles in \( s' \) (i.e., \( p_A \)) will behave as plain \( W \) in \( s' \).

Figure 7 (b) depicts the RHS system (i.e., that containing \( N_2 \)) after the spawn, where:
\[
N_2 = \|\|_\text{Init} (\bar{H}_i, p_i) \|\|_\text{Init} s[p_i] : (L_i' \text{ with } \ldots, e) \|\|_\text{Init} s'[p_i] : (L_i' \text{'}, e) \quad L_i' = p_1: \texttt{InitEx} \cdot p_3: \texttt{ExDone} \cdot \ldots \quad L_i' = p_1: m? \texttt{InitEx} \cdot p_3: m! \texttt{ExDone} \quad \ldots \quad L_i' = \text{end with } \ldots
\]

For (3), \texttt{Spawn} updates the subprotocol states \( L_i'' \) in the \( s \) configurations. It creates new configurations for \( s' \) with these subprotocol states: \( p_1 \) (playing \( m \)) has \( \texttt{gRunEx} \) projected onto \( m (L_i'') \); \( p_2 \) has \( \texttt{gRunEx} \) projected onto \( w_D (L_i'') \); \( p_3 \) has \( \texttt{gRunEx} \) projected onto \( W_E (L_i'') \); and \( p_4 \) has \( \texttt{gRunEx} \) projected onto \( W (L_i'') \) – with role-to-participant bindings \( p_1: m, p_2: w_D, p_3: w_E \) embedded in all these projections. It adds the new subsession \( s' \) to the monitoring tree, i.e., \( \Theta' = \Theta [s' \mapsto (p_1, p_2, p_3, p_4)] \).

We now explain the \texttt{Spawn} rule and (1)–(3) in more detail. For (1), premise \( p_1 = q \cup (R_i' \cup \bar{R}_i) \setminus \mathcal{F} \) determines all of the parent subsession (\( s_1 \) participants involved in the spawn event; i.e., the participants bound to role arguments (\( \bar{q} \mathcal{R} \)), all unsuspected participants in the assigned role’s role set (\( R' \)), and all unsuspected participants in the role set arguments (\( \bar{R} \)). We model participants \( p_1 \) as synchronized when each has no active handler and its subprotocol state designates the spawn event (i.e., its configuration for \( s_1 \) has the form \( (s_1[p] : (E_p[L], \_), L = q(\bar{q} \mathcal{R}, R'; \bar{R}) \cdot L_p) \). For (2), we assign a participant (\( p_0 \in R_i' \setminus \bar{q} \)) that is not already bound to a role in the new subsession. Then \( p_2 = q \cup p_0 \cup (R_i'' \setminus \mathcal{F}) \) is the set of participants joining the subsession; the non-assigned members of \( R' \) do not join (unless \( R' \in R_i'' \)). We also check that all participants bound to roles are non-suspected (\( p_2' \cup \mathcal{F} = \emptyset \)). For (3), we update the subprotocol states of each \( p \in p_1 \) involved in the spawn (i.e., \( s_1[p] : (E_p[L], \_), L = q(\bar{q} \mathcal{R} R', \bar{R}) \cdot L_p \)). A fresh name \( s_2 \) for the new subsession is added to the monitoring tree and recorded as a child of \( s_1 (\Theta' = \Theta [s_2 \mapsto (p_2, 0), s_1 \mapsto (s_1' \cup s_2)]) \), and an initial configuration for \( s_2 \) is created for each \( p \in p_2 \). The initial subprotocol state \( L' \) is given by \( g \mathcal{R} p_0 \mathcal{R} p \). It simply computes \( g \mapsto r = g_r \) if \( p \) is bound to a role \( r \), else \( g \mapsto r = g_R \) where \( R \) is \( p \)'s role set; then yields the subprotocol body \( L' \) of \( g_r \mathcal{R} R \), with all the role bindings of participants \( \bar{q}; p_0 \) embedded as \( \text{pir} \) annotations.

To simplify \texttt{Spawn} without impacting the overall theory, our formalism does not actually fire spawn handlers, as the actions described above are driven solely from the subprotocol states (this is why spawn terms are formalised without a continuation). Our prototype (Sec. 7) does fire spawn handlers, and implements \texttt{Spawn} as a simple handshake – coordinated by a designated role using its local fail set (i.e., our implementation decomposes \( \mathcal{F} \) into local views).

The rules \texttt{Sess-GC} and \texttt{Root-GC} in Fig. 9 perform garbage collection of configurations \( s[p] : (L_p, \_) \) and monitoring tree \( \Theta \) entries for finished subsessions. Predicate done used in these rules holds if the subprotocol state is terminated (completed, i.e., \( L_p = \text{end with } \), or \( L_p = \text{end with } \)) or stopped (halted due to active failure handling by an ancestor subsession, i.e., \( L_p = \text{end}_l \)). \texttt{Sess-GC} removes the configurations (of unsuspected participants) and the monitoring tree entry for subsession \( s_2 \) when all unsuspected participants \( \bar{p} \) are finished in \( s_2 \). \texttt{Root-GC} is a special case for the root subsession that also removes the corresponding event loops. The garbage collection rules allows a tidier progress statement. The direct practical interpretation of these rules is a subsession teardown synchronization (as we employ in our implementation). As for subsession initiation, more elaborate schemes are possible but outside the main topic, hence we opt for a simpler core formalism.

### 5.4 Failure Suspicion and Handling

Fig. 9 also gives the rules for failure suspicion and handling. \texttt{Susp} non-deterministically declares that an arbitrary participant is suspected of failure. For broader applicability, we expressly abstract from specific kinds of failure and failure detection mechanisms: this rule reflects the absence of
control over failures and un reli a bility of failure detection in real-world distributed systems – being suspected has no bearing on whether the participant continues execution or actually stops.

Rule Mon activates the failure handling at the monitor participant. If p is suspected, Mon can switch the monitor’s subprotocol state to the failure handling (i.e., – with L). We write s \sim^+_{\Theta} for the set of subsession names that are transitively reachable from the subsession name s in \Theta (i.e., all descendant subsessions of s). All configurations of p for these s_i are stopped (s_i[p_i : (end_{p_i, ...})), and p will no longer participate in them: a stopped subprotocol state end_L does not match any local type, i.e., end_L \sim L’ is always false. RcvFN activates failure handling at a non-monitor participant, due to receiving a failure notification, i.e., a message communicated from some other failure handling activity to convey the suspected failure; it is similar to Mon. Lastly, Clean models that participants prioritize failure handling over the normal activity. This rule simply discards messages associated with the normal activity from the front of an input FIFO if p “knows” of a suspected failure, either because the failure handling is already active (the left side of the \lor), or a failure notification has arrived in the FIFO (the right side). l \in L means that l syntactically occurs in L. Prioritizing failure handling is crucial in asynchronous multiparty systems, where there is no causality between the arrival of normal messages and failure notifications; e.g., to avoid being stuck on a blocking action in the normal activity that can no longer be discharged. Recall that label sets between a normal and failure activity are disjoint, providing the intuitive condition for safe failure notifications.

Example 5.6. Assume (\mathcal{F}_0, \Theta’, (vs : G_{CM}) N_2) from Ex. 5.5. We illustrate the reductions for the failure scenario in Fig. 7 where p_2 playing \texttt{wd} is suspected of failure. In Fig. 7 (c), monitor p_1 activates failure handling in s, stops its activity in s’, and sends l_{\text{FailDr}} notifications to p_2 and p_4. Let K = \{1, 3, 4\} (i.e., all unsuspected p’s) and J = \{3, 4\} (the unsuspected workers).

\[ (\mathcal{F}_0, \Theta’, (vs : G_{CM}) N_2) \xrightarrow{\text{Susp}} ((p_2), \Theta’, (vs : G_{CM}) N_2) \xrightarrow{\text{Fire, Mon, MSnd}} ((p_2), \Theta’, (vs : G_{CM}) N_3) \]

\[ N_3 = \|_{k \in K} (\overline{H}_k, p_k) \| s[p_1]: (- with g_{\text{RunDr}}(p_1;m; W; W), b_1) \| s’[p_1] : (end_{l_1}; b_1’) \| s’’[p_1] : (l_1’; b_1’’), ... \]

Susp sets p_2 as suspected, i.e., it adds p_2 to \mathcal{F}_0. Then Fire fires the handler of p_1 for the \texttt{wd} suspicion event, i.e., it fires [\texttt{wd}]. W’!l_{\text{FailDr}} \lambda x. x[Wd]. W’!l_{\text{FailDr}}. We then execute this handler. First, Mon sets the subprotocol state of p_1 in s to active failure handling and removes the suspicion type prefix, i.e., – with W’!l_{\text{FailDr}} g_{\text{RunDr}}(p_1;m; W; W); and it sets the subprotocol states of p_1 in all descendant subsessions of s (i.e., s’) to stopped (i.e., end_{l_1}). Next, MSnd multisends l_{\text{FailDr}} to all unsuspected workers, i.e., p_2 and p_4. Fig. 7 (c) depicts ((p_2), \Theta’, (vs : G_{CM}) N_3).

In Fig. 7 (d), p_3 and p_4 first each activate their failure handling in s.

\[ (p_2), \Theta’, (vs : G_{CM}) N_3) \xrightarrow{\text{Fire, RcvFN, RcvFN, RcvFN}} ((p_2), \Theta’, (vs : G_{CM}) N_4) \]

\[ N_4 = \|_{k \in K} (\overline{H}_k, p_k) \| s[p_1]: (- with g_{\text{RunDr}}(p_1;m; W; W), b_1) \| s’[p_1] : (end_{l_1’}; b_1’) \| s’’[p_1] : (l_1’’; b_1’’’), ... \]

p_3 and p_4 each fire their handler on receiving the failure notification l_{\text{FailDr}} in s. Each then reduce their active handler using RcvFN: they activate failure handling in their subprotocol states of s, remove the branch prefix from the failure activity (i.e., consume the l_{\text{FailDr}}), and set all descendant subprotocol states to stopped (end_{l_1}). The above assumes b_j(p_1) = \epsilon; otherwise, Clean would first discard the (now obsolete) normal activity labels from p_1 in p_1’s buffer.

Lastly, p_1, p_3, and p_4 restart g_{\text{RunDr}}. We illustrate the Spawn step by spawning a new subsession s’’. This time, it happens that p_3 is assigned to play \texttt{wd}. Figure 7 (d) depicts ((p_2), \Theta’’, (vs : G_{CM}) N_5) (\Theta’’ = \Theta’[s’’ \mapsto \{p_1, p_3, p_4\}]), where: ((p_2), \Theta’, (vs : G_{CM}) N_4) \xrightarrow{\text{Spawn}} ((p_2), \Theta’’, (vs : G_{CM}) N_5)

\[ N_5 = \|_{k \in K} (\overline{H}_k, p_k) \| s[p_1]: (- with end, b_k) \| s’[p_1] : (end_{l_1}; b_1’’) \| s’’[p_1] : (l_1’’; b_1’’’), ... \]

Proc. ACM Program. Lang., Vol. 5, No. OOPSLA, Article 124. Publication date: October 2021.
6 TYPE SYSTEM AND PROPERTIES

6.1 Typing Rules

The typing judgments used by our system are of the shapes $\Gamma, \Sigma \vdash N \triangleright \Delta$ and $\vdash (\Theta, \mathcal{F}, (vs : \mathcal{G}) N)$. The latter is for the top-level system, i.e., a network under the root session restriction; the former is for all other rules for processes and networks. Standard environment $\Gamma$ maps the root session name to a top-level global type ($s : \mathcal{G}$). Configuration environment $\Sigma$ maps channel values to configuration types ($s[p] : (L, b)$), and participant names to role sets and handler types ($p : R : \{L_i\}_{i \in I}$). Session environment $\Delta$ maps channels to guard types ($c : L$). We write $\Delta_{end}$ for an end-only session environment (i.e., $\Delta_{end}(c) = \text{end}$, for all $c$ in $\Delta_{end}$); $\Sigma_1 \cdot \Sigma_2$ for the union of two configuration environments with disjoint domains; and $\Delta_1 \cdot \Delta_2$ for the union of $\Delta_1$ and $\Delta_2$, provided for $\{i, j\} = \{1, 2\}$, $c : L \in \Delta_i$ implies ($c \notin \Delta_j \lor \Delta_j(c) = \text{end}$). Basically, $\Delta_1 \cdot \Delta_2$ is the union if no $c$ is in both $\Delta_i$ unless one/bot both map to end; else, it is undefined. An append $c : L$, $\Delta$ is defined if $c$ is not in $\Delta$; analogously for $\Gamma$ and $\Sigma$.

The top three rows in Fig. 10 give the rules for event loops and handlers. TELoop types the event loop for participant $p$ of role set $R$ under $\Gamma = s : \mathcal{G}$. It checks each handler is well-typed, and the set of guard types together provides sufficient coverage of all subprotocols projected from $\mathcal{G}$ onto $R$, i.e., $\forall g_{2}(...) = L \in (\mathcal{G} \uparrow R) \cdot \{L_i\}_{i \in I} \vdash L$. Intuitively, the event loop must cover every possible event in the overall protocol that may involve $p$. The coverage judgment $\{L_i\}_{i \in I} \vdash L$ simply asserts that (1) at least one guard type (and thus handler) on the LHS can be matched to $L$’s normal activity, (2) similarly for the failure handling activity, and (3) recursively so for the remaining unmatched portions of each activity, if any. The event loop is then typed with its set of guard types ($p : R : \{L_i\}_{i \in I}$) as $\Sigma$, and $\Delta_{end}$ (the event loop becomes the terminal term in an active handler process, cf. FIRE). We leave the full definition of coverage to the supplement.

Example 6.1 (Coverage). Consider $\{L_i\}_{i \in I} \vdash L$ with $L$’ where $L = p : a?l_1.q:b!l_2$. Coverage requires that some prefix of $L$ matches a guard type (i.e., handler) in $\{L_i\}_{i \in I}$, say, $L_i = a?l_1$. $\{L_i\}_{i \in I}$ is then required to provide coverage of the remaining portion of $L$, i.e., $b!l_2$; and similarly for $L’$. THandler types a handler under an empty $\Sigma$ and empty $\Delta$ – it checks that the handler body uses the channel argument fully according to the guard type. As discussed in Sec. 4.3, it restricts the guard type to a valid handler behavior: we write $wf(L)$ to simply assert that $L$ is (1) flat (all selections/branchings must be unary sends/receives), (2) only the initial action is potentially blocking (receive, spawn or suspicion), and (3) does not contain recursion. TSnd, TMultSnd, TRcv and
We can now present the key properties of our system. See the online supplement for the full details and proofs. We note the previously discussed assumptions of our system, namely that (1) participants playing the role(s) declared in the root subsession are robust, and (2) the runtime infrastructure provides sufficient participants for role assignments during execution. In fault-sensitive distributed systems (1) is commonly achieved by making use of a so-called “application master”; this component is typically made fault-tolerant by managing its critical state with a fault-tolerant coordination service, e.g., ZooKeeper [Hunt 2010], and (2) is a matter of runtime availability of resources.

Subject reduction states that a well-typed system remains well-typed after any reduction step.

**Theorem 6.3 (Subject Reduction).** Let \( \vdash (\Theta_1, \Gamma_1, (v : \mathcal{G}) N_1) \) such that \( (\Theta_1, \Gamma_1, (v : \mathcal{G}) N_1) \rightarrow (\Theta_2, \Gamma_2, (v : \mathcal{G}) N_2) \). Then \( \vdash (\Theta_2, \Gamma_2, (v : \mathcal{G}) N_2) \).

Note that a well-typed system is coherent by TSystem. The proof proceeds as follows. First, as in Honda et al. [2016] and Coppo et al. [2016], we define a typing environment reduction \( ((\Delta, \Sigma, \mathcal{F}, \Theta) \rightarrow (\Delta', \Sigma', \mathcal{F}', \Theta')) \) that describes how the environments are updated in relation to asynchronous network reduction. In other words, it mimics the interaction dynamics of network reduction at the local type level. We then show preservation of coherence, i.e., typing environment reduction preserves coherence. The proof is by enumerating over all cases of typing
environment reduction. Second, we prove type preservation: a well-typed network \((\Gamma, \Sigma \vdash N_1 \triangleright \Delta)\) remains well-typed after a reduction step to \(N_2\). The proof is by induction on a derivation of the reduction step. Type preservation itself does not demand coherent typing environments, as in some cases the proof requires inductively splitting a network into parts that are not necessarily coherent. Note that in our event-driven setting, \(\text{Recv}\) reductions are implicitly safe (they are guarded by \(\text{Fire}\), which ensures a supported message label is present). Subject reduction follows from preservation of coherence and type preservation.

The key session communication safety property in our system, including safe failure handling, follows as a corollary. For a given session \(s\), we say an unsuspected \(p\): (i) has a reception error if it has a receive subprotocol state but the queued label is not accepted by that receive and \(\text{Clean}\) does not apply; (ii) is stuck if it is blocked on a receive or spawn due to waiting for another unsuspected \(p'\), and vice versa; and (iii) has a non-covered failure if it is blocked on a receive or spawn due to waiting for a suspected \(p'\), and no failure handling in the entire monitoring tree applies.

**Corollary 6.4 (Communication Safety).** Let \(\vdash (\Theta, F, (v:G) N)\). For every session \(s\) in \(\Theta\) and unsuspected \(p\) in \(s\), \(p\) has the following properties: (i) \(p\) does not have a reception error; (ii) \(p\) is not stuck; and (iii) \(p\) does not have a non-covered failure.

Communication safety is a direct consequence of the coherence invariant.

The other key property of our system is global progress: progress for an entire (well-typed) top-level system of multiple, concurrent subsessions in the presence of concurrent failures. There are two supporting properties. First, we prove protocol fidelity, which ensures that all potential process interactions in a coherently well-typed system are documented by the top-level global type; i.e., they are protocol-compliant. Our approach adapts that of Coppo et al. [2016]: we define (a) global type reduction, which models interaction dynamics at the global type level, and (b) a mirror relation that relates a global type to typing environments and their configuration types via projection. Fidelity then states that a subsession \(s\) in a (coherently) well-typed network reduces in tandem with its corresponding global type \(G\), and that the \(G'\) at each step mirrors the configuration types of \(s\). Second, we prove subsession progress for every individual subsession that is “live”: i.e., the subsession is not terminated, has no stopped participant (i.e., no participant with a stopped configuration), and has no outer failure (i.e., no failed participant whose failure is handled in an ancestor subsession – its role was assigned by the ancestor subsession). Our subsession progress is a generalization of the standard progress property of MPSTs that is limited to a single session and does not consider any notion of failure (cf. Sec. 3.3).

**Theorem 6.5 (Subsession Progress).** Assume an initial system \(\vdash (\Theta_1, F_1, (v:G) N_1)\) and \((\Theta_1, F_1, (v:G) N_1) \rightarrow^* (\Theta_2, F_2, (v:G) N_2)\). Let \(s'\) in \(\Theta_2\) such that \(s'\) is not terminated, has no stopped participant, and has no outer failure. Then \((\Theta_2, F_2, N_2')\) \(\rightarrow (\Theta_3, F_3, N_3)\) via a reduction in session \(s'\), with either \(N_2' = N_2\) or \((\Theta_2, F_2, N_2) \rightarrow^* (\Theta_2, F_2, N_2')\).

The proof builds on subject reduction and fidelity by adapting the approach of Coppo et al. [2016]. By fidelity, for any “live” subsession \(s'\), we have a global type \(G\) that has reduced in tandem with \(s'\). Let \(p\) be any participant active in the first action described in \(G\) (i.e., in its active failure handling if applicable, else its normal activity). We show that this first action ensures a reduction step in \(s'\) is available. The proof is by enumerating over all possible shapes of the first action, supported by the following. (1) By fidelity, the configuration type of \(p\) contains this first action in their subprotocol state, and in the case of a receive action the label is present in the queue. (2) If \(p\) does not have an active handler for \(s'\), then coverage ensures that \(p\)’s event loop has a handler with a matching guard type for its subprotocol state (i.e., a handler can be fired for the pending event). (3) An active event handler (for any subsession \(s''\)) can always reduce back to the event loop (this corresponds
to the \((\Theta_2, F_2, N_2) \rightarrow^* (\Theta_2, F_2, N_2')\) clause in the theorem). All together, these ensure that \(p\) can always perform a reduction step for this first action.

**Global progress** states that a system can always progress unless it has completely terminated.

**Theorem 6.6 (Global Progress).** Assume an initial system \(\vdash (\Theta_1, F_1, (\nu : G) N_1)\) and a reduction \((\Theta_1, F_1, (\nu : G) N_1) \rightarrow^* (\Theta_2, F_2, (\nu : G) N_2)\). Then either \(\Theta_2\) is empty, or without using \text{Susp} we have \((\Theta_2, F_2, (\nu : G) N_2) \rightarrow (\Theta_3, F_3, (\nu : G) N_3)\).

A completely terminated system is signified by an empty monitoring tree (i.e., when \(\Theta_2\) is empty), implying that every subsession was terminated or stopped, and the configuration terms and monitoring tree entries have been removed via \text{Sess-GC} and \text{Root-GC}. The proof idea is as follows.

Event-driven concurrency is crucial to global progress in our framework: despite concurrent subsessions, progress in any given subsession is independent of all other subsessions (i.e., actions in one subsession are never blocked by those in another). We thus consider systems where every subsession is not "live" (otherwise subsession progress would apply to one) – in such cases, however, we show that some other reduction step is available, e.g., one of the garbage collection rules. Here, intersession coherence ensures every subsession is safely covered by a failure handling in some ancestor subsession. The side condition regarding \text{Susp} ensures that progress does not abuse failure suspicion to bypass actually stuck sessions.

### 7 EVALUATION

**Example applications.** Sec. 2–3 introduced our session-typed CM, which demonstrates how all the various features of our system are needed in a practical fault-tolerant protocol. The table below summarizes further examples from MPST literature that we have specified and implemented (with added failure handling) using our Scala toolchain to demonstrate its expressiveness.

| (1) Core MPSTs (MP interactions, choice, recursion) | (3) Application-level exceptions/interrupts |
|----------------------------------------------------|------------------------------------------|
| 2-Buyers, Streaming [Honda et al. 2016]           | Two Factor [Fowler et al. 2019]          |
| Sutherland-Hodgman [Neykova et al. 2018]          | Resource Control [Demangeon et al. 2015]|
| (2) Dynamic/parameterized participants             | WebCrawler [Neykova and Yoshida 2017]    |
| 3-Buyers [Coppo et al. 2016]                      | Interruptible 3-Buyers [Capecchi et al. 2016] |
| N-stage Pipe [Castro-Perez et al. 2019]           | Basic failure handling (cf. Fig. 12)      |
| N-stage Ring [Castro-Perez et al. 2019]           | Failure-Aware Streaming [Viering et al. 2018] |

The main takeaways, corresponding to parts (1)–(3), are that our system: (1) subsumes the core communication constructs of standard MPSTs; (2) by necessity supports communication patterns previously limited to relatively exotic MPST features; and (3) can express MPST-based application-level exception and interrupt patterns. Further details and the source code are available online.

**Runtime performance (Spark use case).** To demonstrate the practicality of our framework, we compare the performance of the full version of our Session-CM (see supplement), running over our session runtime, to Spark-CM. Our Session-CM is not just a "toy" – it is compatible with the other Spark Core components and supports the execution of existing third-party Spark applications without any code modification, enabling this experiment. Note, we only reimplement the CM, so our session runtime conducts only the Session-CM internal messaging, not the communications of the Spark application running on top.

The aim is to measure the overheads incurred by our toolchain and session runtime prototype implementation. Our theory introduces additional mechanisms, i.e., potential sources of overhead, mainly: (i) the CFSM used to track the session protocol state and selection of event handlers at runtime, (ii) dynamic checking of linear channel usage, and (iii) infrastructure used to implement...
subsessions (e.g., additional message queues). We note our prototype is not optimized in general (e.g., some abstraction layers around our message queues could be eliminated), whereas Spark’s CM is a mature, industrial-strength component.

We use a Spark implementation of the industry-standard TPC-H benchmark as the benchmark application. TPC-H specifies a set of 22 complex queries on a database of 8 tables; we use a database scaling factor of 10 (i.e., database size ∼10GB). Each benchmark run measures the total time (including application startup time) to execute one query as an independent application using (a) our Session-CM and session runtime, and (b) the Spark-CM, for scheduling; all other factors are the same. In both cases each Spark application (query) has three servers with identical hardware (Intel Xeon E-2278G CPUs, 64GB RAM) and running Ubuntu 18.04.3 LTS.

Fig. 11 reports the average (arithmetic mean) of the execution times (y-axis) and standard deviations (error bars) of 10 runs per query. The plot further states the normalized execution time of the Session-CM for each query, i.e., the result of dividing the average execution time of the Session-CM by the average execution time of the Spark-CM. Across all 22 queries, our Session-CM exhibits an average overhead below 10% (and a maximum below 16.5%) compared to Spark-CM. We repeated the experiment allocating two servers to a Spark application, where a server running an executor fails after 20s (by killing it) and is successfully replaced, for a query we picked at random (Q18). The average execution times of 5 runs are: 99.19s for Spark-CM (std. dev. of 2.23s), and 109.4s for our Session-CM (std. dev. of 0.44s), i.e., an overhead of ∼10%.

8 RELATED WORK

Failure handling in MPSTs. In session types literature, the two main related works on failure handling in MPSTs, both without performance evaluation, are by Adameit et al. [2017] and Viering et al. [2018]. Adameit et al. [2017] extend MPSTs by wrapping interactions in optional blocks to model communication link failures. They require default values for input actions that may thusly fail, giving a form of failure masking (mask. in the CSFH column of Fig. 12) where the protocol specification outside of an optional block is agnostic to whether a failure occurs or not. When execution discards the block of some role due to failure, peer roles must also discard their corresponding blocks: such agreement in the presence of failures is only possible with synchronous channels (sync. in the aMP column) or – boiling down to the same – perfect failure detection (no false suspicions). This limits applicability to real-world distributed systems that are asynchronous.

Fig. 11. Comparing mean execution times of our Session-CM to the Spark-CM on TPC-H queries. The numbers on the bars give the execution time of our Session-CM normalized to the Spark-CM.

---

\(\text{Session-CM} \quad \text{Spark-CM} \quad \text{Normalized execution time}\)

| Query | Time (sec.) | Time (sec.) | Normalized |
|-------|-------------|-------------|-------------|
| Q1    | 1.05        | 1.16        | 0.87        |
| Q2    | 1.18        | 1.18        | 0.99        |
| Q3    | 1.27        | 1.23        | 0.97        |
| Q4    | 1.31        | 1.21        | 0.99        |
| Q5    | 1.09        | 1.24        | 0.87        |
| Q6    | 1.12        | 1.10        | 0.87        |
| Q7    | 1.03        | 1.14        | 0.87        |
| Q8    | 1.11        | 1.06        | 0.87        |
| Q9    | 1.06        | 1.09        | 0.87        |
| Q10   | 1.09        | 1.08        | 0.87        |
| Q11   | 1.10        | 1.10        | 0.87        |
| Q12   | 1.11        | 1.10        | 0.87        |
| Q13   | 1.08        | 1.12        | 0.87        |
| Q14   | 1.08        | 1.10        | 0.87        |
| Q15   | 1.05        | 1.07        | 0.87        |
| Q16   | 1.05        | 1.07        | 0.87        |
| Q17   | 1.07        | 1.07        | 0.87        |
| Q18   | 1.11        | 1.06        | 0.87        |
| Q19   | 1.07        | 1.07        | 0.87        |
| Q20   | 1.08        | 1.10        | 0.87        |
| Q21   | 1.16        | 1.06        | 0.87        |
| Q22   | 1.07        | 1.00        | 0.87        |

TPC-H queries implemented in Spark by Savvas Savvides. [https://github.com/ssavvides/tpch-spark]
Session type frameworks

|                  | aMP | CSFH | FS   | SubS | Par | DP | RR | EDP |
|------------------|-----|------|------|------|-----|----|----|-----|
| This paper       | ✓   | ✓    | ✓    | ✓    | ✓   | ✓  | ✓  | ✓   |
| [Adameit et al. 2017] | sync. mask. | ✓ | ✓ | ext. |
| [Carbone et al. 2008] | sync. | app. | ✓ | ✓ |
| [Demangeon et al. 2015] | ✓ | app. | ✓ | ✓ |
| [Capecchi et al. 2016] | sync. | app. | ✓ |
| [Hu et al. 2010] | ✓ | ✓ | ✓ | ✓ |
| [Demangeon and Honda 2012] | ✓ | ✓ | ✓ |
| [Viering et al. 2018] | ✓ | ✓ | ✓ |
| [Capecchi et al. 2016] | ✓ | ✓ | ✓ |
| [Demangeon et al. 2015] | ✓ | ✓ | ✓ |
| [Carbone et al. 2008] | ✓ | ✓ | ✓ |
| [Hu et al. 2010] | ✓ | ✓ | ✓ |

Fig. 12. Comparison to related works; Sec. 8 clarifies where entries are neither “yes” (√) nor “no” (blank).

Their approach includes nested protocols [Demangeon and Honda 2012] for external participant invitations during a nested session call (ext. in the DA column), offering a form of dynamic role assignment.

Viering et al. [2018] extend MPSTs with try-catch handling for process crash failures. They closely follow traditional MPSTs by directly preserving try-catch structures across global types, local types, and “multithreaded” session π-calculus processes. Their approach works by coupling the reduction (i.e., control flow) of the try-catch construct at the process level to a specific mechanism for detecting distributed process failures based on a fail-safe coordinator, e.g., ZooKeeper [Hunt 2010]. Their system assumes perfect failure detection. By contrast, our model makes no assumptions on the accuracy [Chandra and Toueg 1996] of failure detection (our communication safety and progress are ensured despite continued actions by falsely suspected peers), and monitors are peers that may be failure-prone themselves. Our framework thus applies to concrete systems based on (e.g.) ZooKeeper, but is also not coupled to any specific oracle infrastructure.

Binary sessions, exceptions, and events. Fowler et al. [2019] add an exception handling process primitive for failures in binary sessions. Type-level treatment of failure handling behaviors between the remaining (and new) peers, as in this paper, cannot be studied in a binary setting (binary in the aMP/CSFH columns). Besides the above, the existing session types literature (e.g., [Capecchi et al. 2016; Carbone et al. 2008; Demangeon et al. 2015]) has only addressed application-level failures (app. in the CSFH column). These works consider “exceptional” behaviors in the sense of application logic (e.g., an inappropriate payload value), as opposed to actual participant process failures – the former deal with exception-like protocol control flow over a normally functioning session between fixed participants, while the latter necessitates reasoning about sessions with changing participants due to failures and replacements. Our work tackles the latter by modeling protocols with explicit specification of participant failure detection/notification, asynchronous failure handling behaviors, and retrying failed interactions with dynamically replaced participants. None of the works cited above study fault-tolerance in this regard nor event-driven concurrency for MPSTs.

Hu et al. [2010] present a binary session calculus for encoding EDP patterns. They add a non-blocking primitive for polling channels for messages and a session typecase construct based on dynamic typing [Abadi et al. 1991], while we model event loops and handlers as first-class concepts. Their work does not consider failures (as mentioned, specification of fault-tolerant, multiparty communication patterns cannot be studied in a binary setting), and does not consider the challenges that we address for MPSTs: (partial) projections, coherence, or fidelity. Cano et al. [2017] present a related binary calculus (without types) for reactive sessions (react. in EDP). Coppo et al. [2016] develop an additional interaction type system on top of MPSTs to analyze global progress; our system...
An MPST Discipline for Fault-Tolerant Event-Driven Distributed Programming offers global progress for concurrent (sub)sessions within MPSTs. Our notions of subprotocols and subsessions are inspired by the nested protocols of Demangeon and Honda [2012]. We exploit subsessions to (1) incorporate lightweight practical participant parametricity and dynamic role assignment, and (2) reason about the runtime structure of failure monitoring between subsessions. Their work does not consider failures, and their progress property is restricted to a single session.

**Outside of session types.** This paper (and more broadly MPSTs) promotes a methodology for safe development of message passing applications based on top-down protocol specification and distributed endpoint implementation. There exists a wide range of techniques and tools that focus more, in comparison to this paper, on verification than software development, in the context of distributed systems and algorithms with failure handling. They can be considered on a scale from mostly "manual" to automated techniques. In this regard, the verification aspect of our toolchain can be considered as fully automated (for our specific MPST-based properties) after the user supplies the protocol specification as a global type.

At the former end of the scale are verification approaches based on interactive proof tools such as Verdi [Wilcox et al. 2015] and IronFleet [Hawblitzel et al. 2015]. Recent approaches [Sergey et al. 2018; Taube et al. 2018; von Gleissenthal et al. 2019] employ modularization of proofs or transformation of asynchronous programs into synchronous versions to reduce the user effort. These approaches allow establishing user-defined functional correctness properties for distributed systems, but can require substantial proof engineering effort. The formal specifications written by the user can become quite complex [Chong et al. 2021].

Semi-automated verification approaches like PSync [Dragoi et al. 2016] and Ivy [Padon et al. 2016] can automate parts of the verification by restricting the underlying model. PSync is a domain-specific language based on the heard-of model [Charron-Bost and Schiper 2009] and structures programs into sequentially executed rounds. Ivy verifies safety properties for algorithms; it introduces a modeling language based on EPR logic [Piskac et al. 2010] for which the checking of invariants is decidable. It interactively guides a user via counterexamples to a valid invariant.

At the other end of the scale are fully automated verification techniques that trade off the supported class of properties and algorithms/systems, and perform full or partial verification. Model checking tools can transparently check distributed system implementations [Yang et al. 2009], be combined with a tailored programming language [Killian et al. 2007], and automatically verify restricted classes of distributed algorithms [Konnov et al. 2017; Ma et al. 2019]. Applications of these tools to concrete implementations of distributed systems may be best effort (i.e., incomplete).

9 CONCLUSION, LIMITATIONS AND FUTURE WORK

We presented an MPST framework for practical fault-tolerant distributed programming. We formalized our approach, proving communication safety and global progress, and demonstrated its practicality by implementing and evaluating a session-typed CM for Apache Spark.

Our present formulation inherits some of the standard conservative restrictions of MPSTs. For example, our global type branch is a directed choice, meaning the immediate action in every choice case is between the same two roles/role sets (e.g., $m \rightarrow W$). We believe it is possible to port techniques for relaxing such restrictions from recent works into our setting, e.g., by augmenting global type expressiveness using MPST-based behavioral type properties [Scalas and Yoshida 2019], or by incorporating explicit connection actions [Hu and Yoshida 2017] into subsession initiation.

As future work, we are considering further reimplementations of components from related middleware systems in the Apache middleware family (e.g., Storm, Fink, Kafka). We are also considering further performance evaluations, e.g., scalability in relation to data set size and number of worker processes, as part of improving our toolchain and runtime implementation.
REFERENCES

Martin Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. 1991. Dynamic Typing in a Statically Typed Language. *ACM Trans. Program. Lang. Syst.*, 13, 2 (1991), 237–268. https://doi.org/10.1145/103135.103138

Manuel Adamietz, Kirstin Peters, and Uwe Nestmann. 2017. Session Types for Link Failures. In *FORTE ’17*, Vol. 10321. Springer, 1–16. https://doi.org/10.1007/978-3-319-60225-7_1

Davide Ancona et al. 2016. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages*. 3, 2–3 (2016), 95–230. https://doi.org/10.1561/2500000031

Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. 1996. Simulating Reliable Links with Unreliable Links in the Presence of Process Crashes. In *10th International Workshop on Distributed Algorithms (WDAG’96)* (Lecture Notes in Computer Science). Springer, 105–122. https://doi.org/10.1007/3-540-61769-8_8

Mauricio Cano, Jaime Arias, and Jorge A. Pérez. 2017. Session-Based Concurrency, Reactively. In *FORTE ’17* (Lecture Notes in Computer Science, Vol. 10321). Springer, 74–91. https://doi.org/10.1007/978-3-319-60225-7_6

Sara Caperchi, Elisa Giachino, and Nobuko Yoshida. 2016. Global Escape in Multiparty Sessions. *MSCS*, 2, 2 (2016), 156–205. https://doi.org/10.1007/S00446-014-0084-6

Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2008. Structured Interactional Exceptions in Session Types. In *CONCUR ’08* (LNCS, Vol. 5201). Springer, 402–417. https://doi.org/10.1007/978-3-540-85361-9

David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. 2019. Distributed programming using role-parametric session types in go: statically-typed endpoint APIs for dynamically-instantiated communication structures. *Proc. ACM Program. Lang.*, 3, POPL (2019), 29:1–29:30. https://doi.org/10.1145/3290342

Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM*, 43, 2 (1996), 225–267. https://doi.org/10.1145/226643.226647

Bernadette Charron-Bost and André Schiper. 2009. The Heard-Of Model: computing in Distributed Systems with Benign Faults. *Distributed Computing*, 22, 1 (2009), 49–71. https://doi.org/10.1007/s00446-009-0084-6

Nathan Chong, Byron Cook, Jonathan Eidelman, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tachtschgin, and Mark R. Tuttle. 2021. Code-level model checking in the software development workflow at Amazon Web Services. *Softw. Pract. Exp.*, 51, 4 (2021), 772–797. https://doi.org/10.1002/spe.2949

Mario C oppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2016. Global progress for dynamically interleaved multiparty sessions. *MSCS*, 26, 2 (2016), 238–302. https://doi.org/10.1007/S0960129514000188

Romain Demangeon and Kohei Honda. 2012. Nested Protocols in Session Types. In *CONCUR ’12* (Lecture Notes in Computer Science, Vol. 7454). Springer, 272–286. https://doi.org/10.1007/978-3-642-32940-1_20

Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2015. Practical Interruptible Conversations. *Formal Methods in System Design*, 46, 3 (2015), 197–225. https://doi.org/10.1007/s10703-014-0218-8

Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. 2012. Parameterised Multiparty Session Types. *Log. Methods Comput. Sci.*, 8, 4 (2012). https://doi.org/10.2168/LMCS-8(4:6)2012

Cezara Dragoi, Thomas Henzinger, and Damien Zufferey. 2016. PSync: A Partially Synchronous Language for Fault-tolerant Distributed Algorithms. In *POPL ’16*. ACM, 400–415. https://doi.org/10.1145/2837614.2837650

Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32, 2 (1985), 374–382. https://doi.org/10.1145/3149.214121

Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional asynchronous session types: session types without tiers. *PACMPL*, 3, POPL (2019), 28:1–28:29. https://doi.org/10.1145/3290341

Simon Gay and Antonio Ravara (Eds.). 2017. *Behavioural Types: from Theory to Tools*. River Publishers. 287–307 pages. http://eprints.gla.ac.uk/146884/

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Par no, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015*, Monterey, CA, USA, October 4-7, 2015, Ethan L. Miller and Steven Hand (Eds.). ACM, 1–17. https://doi.org/10.1145/2815400.2815428

B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI’11)*.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM*, 63, 1 (2016), 9:1–9:67. https://doi.org/10.1145/2827695

Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. 2010. Type-Safe Eventful Sessions in Java. In *ECOOP ’10* (Lecture Notes in Computer Science, Vol. 6183). Springer, 329–353. https://doi.org/10.1007/978-3-642-14107-2_16

Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification Through Endpoint API Generation . In *FASE ’16* (LNCS, Vol. 9633). Springer, 401–418. https://doi.org/10.1007/978-3-662-49665-7
An MPST Discipline for Fault-Tolerant Event-Driven Distributed Programming

Raymond Hu and Nobuko Yoshida. 2017. Explicit Connection Actions in Multiparty Session Types. In *FASE ’17 (Lecture Notes in Computer Science, Vol. 10202)*. Springer, 116–133. https://doi.org/10.1007/978-3-662-54494-5_7

Patrick Hunt. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *USENIX ’10. USENIX Association.*

Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1 (2016), 3:1–3:36. https://doi.org/10.1145/2873052

Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. 2007. Mace: language support for building distributed systems. In *Proceedings of the 4th ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10–13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 179–188. https://doi.org/10.1145/1250734.1250755

Igor V. Konnov, Marijana Lazic, Helmut Veith, and Josef Widder. 2017. A short countercaseexample for safety and liveness verification of fault-tolerant distributed algorithms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 719–734. http://dl.acm.org/citation.cfm?id=3009860

Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Kareem A. Sakallah. 2019. 14: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) (SOSP ’19). Association for Computing Machinery, New York, NY, USA, 370–384. https://doi.org/10.1145/3341301.3359651

Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. 2018. A session type provider: compile-time API generation of distributed protocols with refinements in F#. In *International Conference on Compiler Construction, CC 2018*, Christophe Dubach and Jingling Xue (Eds.). ACM, 128–138. https://doi.org/10.1145/3178372.3179495

Rumyana Neykova and Nobuko Yoshida. 2017. Let it recover: multiparty protocol-induced recovery. In *Proceedings of the 25th International Conference on Compiler Construction, Austin, TX, USA, February 5-6, 2017*, Peng Wu and Sebastian Hack (Eds.). ACM, 98–108. https://doi.org/10.1145/3033019.3033031

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13–17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 614–630. https://doi.org/10.1145/2908080.2908118

Luca Padovani. 2017. A simple library implementation of binary sessions. *J. Funct. Program.* 27 (2017), e4. https://doi.org/10.1017/S0956796816000289

Ruzica Piskac, Leonardo Mendonça de Moura, and Nikolaj Bjørner. 2010. Deciding Effectively Propositional Logic Using DPLL and Substitution Sets. *J. Autom. Reason.* 44, 4 (2010), 401–424. https://doi.org/10.1007/s10817-009-9161-6

Alceste Scalas and Nobuko Yoshida. 2019. Less is more: multiparty session types revisited. *PACMPL* 3, POPL (2019), 30:1–30:29. https://doi.org/10.1145/3290343

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.* 2, POPL (2018), 28:1–28:30. https://doi.org/10.1145/3158116

Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018. Modularity for decidability of deductive verification with applications to distributed systems. In *PLDI ’18*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 662–677. https://doi.org/10.1145/3192366.3192414

Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O’Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *ACM Symposium on Cloud Computing, SOCC ’13*. ACM, 5:1–5:16. https://doi.org/10.1145/2523616.2523633

Malte Viering, Tzu-Chun Chen, Patrick Eugster, Raymond Hu, and Lukasz Ziarek. 2018. A Typing Discipline for Synchronously Verified Crash Failure Handling in Distributed Systems. In *ESOP ’18 (Lecture Notes in Computer Science, Vol. 10801)*. Springer, 799–826. https://doi.org/10.1007/978-3-319-89884-1_28

Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. 2019. Pretend synchrony: synchronous verification of asynchronous distributed programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 59:1–59:30. https://doi.org/10.1145/3290372

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *PLDI ’15*. ACM, 357–368. https://doi.org/10.1145/2737924.2737958

Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, Jennifer Rexford and Emin Gün Sirer (Eds.). USENIX Association, 213–228. http://www.usenix.org/events/ndsi09/tech/full_papers/yang/yang.pdf

Proc. ACM Program. Lang., Vol. 5, No. OOPSLA, Article 124. Publication date: October 2021.
Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In NSDI ’12. 15–28.

Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2020. Statically verified refinements for multiparty protocols. Proc. ACM Program. Lang. 4, OOPSLA (2020), 148:1–148:30. https://doi.org/10.1145/3428216