

Behavioural Types for Local-First Software


Roland Kuhn ✉ 

Actyx AG, Kassel, Germany

Hernán Melgratti ✉ 

University of Buenos Aires, Argentina

Conicet, Buenos Aires, Argentina

Emilio Tuosto ✉ 

Gran Sasso Science Institute, L'Aquila, Italy

Abstract

Peer-to-peer systems are the most resilient form of distributed computing, but the design of robust protocols for their coordination is difficult. This makes it hard to specify and reason about global behaviour of such systems.

This paper presents *swarm protocols* to specify such systems from a *global* viewpoint. Swarm protocols are projected to *machines*, that is *local* specifications of peers. We take inspiration from behavioural types with a key difference: peers communicate through an event notification mechanism rather than through point-to-point message passing. Our goal is to adhere to the principles of *local-first software* where network devices collaborate on a common task while retaining full autonomy: every participating device can locally make progress at all times, not encumbered by unavailability of other devices or network connections. This coordination-free approach leads to inconsistencies that may emerge during computations. Our main result shows that under suitable well-formedness conditions for swarm protocols consistency is eventually recovered and the locally observable behaviour of conforming machines will eventually match the global specification.

Our model elaborates on the Actyx industrial platform and provides the basis for tool support: we sketch an implemented prototype which proves this work a viable step towards reasoning about local-first and peer-to-peer software systems.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Software and its engineering → Distributed systems organizing principles; Software and its engineering → Distributed programming languages

Keywords and phrases Distributed coordination, local-first software, behavioural types, publish–subscribe, asynchronous communication

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.15

Related Version The extended version of this paper [27] contains more examples, a comparison between our model and *state machine replication* [32], more details on the Actyx middleware, and a discussion on the limitation of our approach.

Extended Version: <https://arxiv.org/abs/2305.04848>

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.9.2.14>

Funding Research partly supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No 778233, by MIUR project PRIN 2017FTXR7S *IT MATTERS* (Methods and Tools for Trustworthy Smart Systems), by the PRO3 MUR project Software Quality, and by the PNRR MUR project VITALITY (ECS00000041), Spoke 2 ASTRA – Advanced Space Technologies and Research Alliance. *machine-runner* and *machine-check* partly funded by the European Union (TaRDIS, 101093006).

Acknowledgements The authors thank the anonymous reviewers for their useful and insightful comments and Daniela Marottoli for her help in the initial phase of this project.



© Roland Kuhn, Hernán Melgratti, and Emilio Tuosto;
licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 15; pp. 15:1–15:28



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

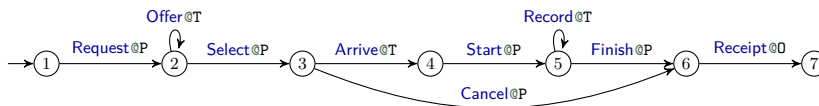


1 Introduction

Fully decentralised systems like peer-to-peer (P2P) networks are notoriously hard to design and analyse. A main challenge is to coordinate components so that the composed system exhibits the expected behaviour. As all decisions in such a system are made locally based on the available partial knowledge, the main problem is to specify which information is transferred to whom and when and how to interpret it, i.e. protocol design. The example we observed with Actyx is implemented in factory shop floor coordination, which is mission-critical: logistics robots compete for transport orders to move goods between machines and warehouses¹. For familiarity, we use taxi rides instead which are of the same shape.

► **Example 1.1** (Our running example). The fleet of a taxi company organises rides within a town using a P2P network. The main goal is to provide any registered passenger who requests a ride with some offers from taxis willing to perform the desired transportation; the passenger may pick one offer based on price and estimated time of arrival – followed by tracking the pickup, ride, and arrival – or cancel the ride. At the end of the trip the accounting office provides a receipt for the journey or cancellation. Note how there is some level of trust between the parties involved; we will treat this as a non-adversarial setting.

The structure of the expected interaction between the different peers (classified by *roles*) can be informally illustrated with the following diagram involving a passenger role P, a taxi role T, and an accounting office role as O.



When a passenger needs a ride (state 1 in the diagram above), they open an auction by executing a **Request**. Then, any taxi with capacity can proceed with an **Offer**. In this description, we do not make any assumption about the number of instances playing each role; in fact, we expect to have many taxis playing role T and hence many offers. The passenger ends the auction by using **Select** to pick a winner; note that we do not capture which taxi won the contract, instead we assume that only the winning taxi will perform actions after state 3 (we could model one selected branch per taxi and replicate states 3–6, but this only makes the example larger without additional insight).

The second phase starts with a race in state 3: the taxi begins the ride invoking **Arrive** while the passenger loses patience and uses the **Cancel** command to back out. The office will create a receipt in either case, but we must settle the dispute whether a ride happened. ┘

The classic solution to such a problem would use a central database to present an up-to-date view of the respective data each participant is allowed to see. This solution avoids conflicts, maintains invariants, and steers the whole process by virtue of there being only one source of truth – one of the **Arrive** or **Cancel** commands would happen first and the

¹ This use-case has been implemented and is used in factories based on the Actyx Pond library, which uses the same event log replication mechanism as described in this article but does not model the interaction of differently shaped machines nor does it provide formal verification of the protocol. For a different factory the implementation and deployment is ongoing using the theory presented herein – developers report greater confidence and productivity than with established approaches. The main difficulty in factory logistics is reasoning about transient network partitions of mobile participants. Note that orchestrating the movement of goods on the shop floor is most critical for a factory’s success!

other would be rejected. It is well-known (cf. the CAP [16] and FLP [14] results) that such a solution suffers from unavailability if the system model includes network partitions, since either the database is localised and may be unreachable, or it is distributed and may need to reject requests to maintain consistency. With the advent of *conflict-free replicated data types (CRDTs)* [33] a different solution came into view: instead of avoiding conflicts through coordination, CRDTs provide a data model of such a structure that is conflict-free by construction. CRDTs facilitate that by demanding a join semi-lattice for the data structure, i.e. that there is a merge function that given two differently evolved states will compute a new state that represents the sum of all operations that were done to either input. Applied to Example 1.1 this would typically be achieved by systematically preferring either side of the choice at state 3, e.g. “cancellation always wins” (cf. the *add-wins set* CRDT). This illustrates that CRDTs are not well-suited for capturing and fairly resolving conflicts such as the one in our example. Nevertheless, the increasing research focus on coordination-free systems inspired the formulation of *local-first software* [25] in which all participants in a distributed system maintain full autonomy and control over their data. Besides the focus on agency and ownership, local-first principles can also be used to build software that is fully available and maximally resilient [26], where each participant can independently make decisions that are globally valid.

We propose a new way of approaching local-first software based on embracing conflict, recognising it, and finally reconciling it to reach eventual consensus [38]. Our model builds upon the middleware developed at Actyx [1], which provides a reliable durable pub-sub mechanism for event logs as well as a coordination-free total order of all events. A distributed system in our model is realised by a set of participants, dubbed *machines*, that can exhibit discordant behaviour and interact by broadcasting and reacting to *events*. Events are generated locally in response to the execution of *commands*, added to the *local log* and then propagated to other participants, to be merged into the log local at the recipient. We assume that events propagate asynchronously and that there is no traditional mechanism for coordination (like consensus or central nodes): machines liaise with each other purely on the basis of the events spreading in the system. We refer to such systems as *swarms*.

Each machine in a swarm implementing the scenario in Example 1.1 plays a role, i.e. it subscribes to a defined subset of event types and applies an assigned logic to interpret those. Depending on its local state (which it computes from its local log), any machine may decide to execute a command. For instance, a machine playing the role P may execute **Request**, which generates new events containing details of the request. Such events are appended to the local log of the machine and then propagated asynchronously to other machines that have subscribed to such events, e.g. the machines corresponding to taxis. After receiving such events, a taxi updates its local state and decides whether to place an offer for the ride. In such case, it executes **Offer**, which generates the events describing the offer, appends them to local log and propagates them to the rest of the system. The interaction described in Example 1.1 may proceed to completion in this way. Our swarms protocols specify the intended communications in an ideal run of the protocol, assuming that different sessions are independent from each other – we therefore do not represent sessions in our syntax.

Noteworthy, our computational model does not preclude the execution of conflicting commands. For instance, the race in state 3 of Example 1.1 allows two machines to generate their respective events and propagate them through the system. When receiving such events, each machine will be in charge of detecting and properly resolving the conflict. This is achieved by using the total order between events – interpreted as manifestation of (logical) time: the earliest event emitted after a choice decides which branch is taken (events from a

losing branch are ignored). Note that we do not assume knowledge of when the event log is complete, i.e. a machine cannot detect whether an event is globally the earliest after a choice; thus, computed local states may transiently diverge. As soon as events up to and including a given choice are fully replicated across the swarm, all machines will agree upon which branch is taken (i.e. we achieve *eventual consensus* [38]).

The fact that events are processed only by subscribers makes the resolution of choices subtle. Assume that the accounting office incorrectly subscribes to the cancellation event but not to the arrival one. In the presence of a conflicting choice, it may incorrectly conclude that the ride was cancelled even when all other roles understand that the ride took place. We rely on a typing discipline for ruling out such inconsistencies. We follow a top-down approach featuring *swarm protocols*, namely abstractions similar to the diagram in Example 1.1 that – akin to global types [19, 20] – formalise a description of the expected protocol from a *global* viewpoint. A projection operation can automatically generate local specifications of each role formally defined as *machines* (cf. Section 2.3). Our typing discipline establishes sufficient conditions – *well-formedness* of swarm protocols – to guarantee that well-typed systems will resolve conflicting choices consistently once information has sufficiently spread to participants. This and the fact that swarm protocols fully abstract away from the number of instances enacting a role are distinguished features of our approach.

Main contributions. We develop a behavioural typing discipline for local-first software tailored to a formal operational model distilled from a real middleware. More specifically:

1. We introduce an operational model for distributed computation based on replication of event logs to drive the behaviour of machines (Section 2). We do not assume log stability but combine speculative computation with a “rewind” mechanism à la *time warp machine* [22]: a conflict is resolved by backtracking and re-execution along the right path.
2. We define a novel behavioural type approach (Section 4) in which swarm protocols are specified in terms of the information injected into a heterogeneous swarm through the actions performed by participants of specific roles. Swarm protocols enjoy a lightweight syntax and simple operational semantics; they are deadlock-free and communication-safe; yet they are expressive enough for modelling complex protocols.
3. We define well-formedness conditions for swarm protocols (Definition 6.7) and a projection operation to derive local machine specifications (Section 5). These ensure eventual consistency between local observations and globally specified behaviour (Theorem 7.14 and Corollary 7.15), which is non-trivial due to the absence of any infrastructure coordination, non-homogeneous event subscriptions across roles, and the ability to implement a role with an arbitrary positive number of replicas.
4. We apply our approach to the TypeScript language and Actyx middleware in the form of a runtime library and a tool for checking protocol well-formedness and conformance, as well as a stochastic simulation tool exploring possible executions.

Assumptions. We work under the following assumptions (cf. Section 9 for extensions):

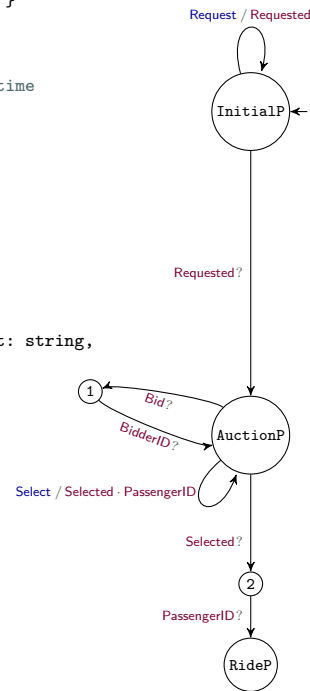
- *collaborative setting*: we consider P2P participants to not be malicious or adversarial;
- *pure effects*: effects performed by machines can be reverted or compensated;
- *reliable pub-sub mechanism*: events (incl. metadata) are neither forged nor lost;
- *session encoding*: machines only receive events pertaining to their session.

Structure. We present the semantics of machines and of log shipping in Section 2. An overview of the accompanying tooling is in Section 3. Swarm protocols are introduced in Section 4 and their local projection onto machines in Section 5. We define well-formed

```

1 // analogous for other events; "type" property matches type name (checked by tool)
2 type Requested = { type: 'Requested'; pickup: string; dest: string }
3 type Events = Requested | Bid | BidderID | Selected | ...
4
5 /** Initial state for role P */
6 @proto('taxiRide') // decorator injects inferred protocol into runtime
7 export class InitialP extends State<Events> {
8   constructor(public id: string) { super() }
9   execRequest(pickup: string, dest: string) {
10    return this.events({ type: 'Requested', pickup, dest })
11  }
12  onRequested(ev: Requested) {
13    return new AuctionP(this.id, ev.pickup, ev.dest, [])
14  }
15 }
16 @proto('taxiRide')
17 export class AuctionP extends State<Events> {
18   constructor(public id: string, public pickup: string, public dest: string,
19     public bids: BidData[]) { super() }
20   onBid(ev1: Bid, ev2: BidderID) {
21     const [ price, time ] = ev1
22     this.bids.push({ price, time, bidderID: ev2.id })
23     return this
24   }
25   execSelect(taxiId: string) {
26     return this.events({ type: 'Selected', taxiID },
27       { type: 'PassengerID', id: this.id })
28   }
29   onSelected(ev: Selected, id: PassengerID) {
30     return new RideP(this.id, ev.taxiID)
31   }
32 }
33 @proto('taxiRide')
34 export class RideP extends State<Events> { ... }

```



■ Listing 1 Definition of state machines in TypeScript.

swarm protocols in Section 6 and study eventual correctness in Section 7. Related works are discussed in Section 8 and Section 9 yields final remarks together with possible generalisations and future work.

2 Asymmetric Replicated State Machines

Our model hinges on three ingredients: machines, event emission and consumption, and logging. The behaviour of a machine is captured by a finite-state automaton as described in Section 2.1. In Section 2.2 we show how machines may offer *commands* that, upon execution, emit events as well as how machines consume (typed) events stored in their *local log*. Such events are immediately stored in the local log of the emitting machine and later asynchronously shipped to the other machines as described in Section 2.3.

2.1 From TypeScript to automata

We formalise (and elaborate on) the computational model realised in the middleware of Actyx by offering a new library for writing endpoint code. Like the Actyx SDK, we use the TypeScript language. Our API focuses on a concise but well-structured expression of finite-state machines for interpreting the current state of distributed computation.

We illustrate this by considering the implementation of the request and auction part of our running taxi example from the passenger's point of view, with the TypeScript code given in Listing 1. For the purposes of this section – all details, including runtime evaluation, are given in Section 3 – it suffices to know that each state of a machine is represented by a

■ **Table 1** Notation for machines.

Notation:	State computation:
$\vdash e : \mathbf{t}$: event e is of type \mathbf{t} $src(e)$: identity of the machine generating e l : event log (seq. without repetition) \mathbf{l} : event log type (sequence of types) \mathbf{c} / \mathbf{l} : command \mathbf{c} emits log type \mathbf{l} $\mathbf{t}?$: consumption of event of type \mathbf{t} \mathbf{M} : machine (labelled transition system)	Let q_0 be the initial state of \mathbf{M} and $\mathbf{M}[q]$ be machine \mathbf{M} with initial state changed to q : $\delta(\mathbf{M}, \epsilon) = q_0$ $\delta(\mathbf{M}, e \cdot l) = \begin{cases} \delta(\mathbf{M}[q], l) & \text{if } \vdash e : \mathbf{t}, q_0 \xrightarrow{\mathbf{t}^?} q \text{ in } \mathbf{M} \\ \delta(\mathbf{M}, l) & \text{otherwise} \end{cases}$

TypeScript class, with methods for command invocation whose name is prefixed with “**exec**”, and event handler methods to compute the next state (names prefixed with “**on**”). The types of the event handler method arguments are significant, as are the return types of command methods. Listing 1 also depicts the finite-state automaton corresponding to the snippet, as inferred by the `machine-check` build tool (cf. Section 3.2):

- states `InitialP`, `AuctionP`, and `RideP` of the automaton respectively correspond to the classes in the snippet with the same name;
- states 1 and 2 of the automaton correspond to the implicit states interspersed between the events specified as arguments to the event handler methods `onBid` and `onSelected`;
- command methods correspond to self-loops in the automaton, labelled with the command name and the resulting event log type, such as `Request / Requested` in state `InitialP`;
- event handlers correspond to transitions or sequences thereof, where each transition is labelled with an event type, such as `Requested?` from state `InitialP`.

The correspondence sketched above is the basis for our formalisation of swarms and it is at the heart of the library `machine-runner` introduced in Section 3.

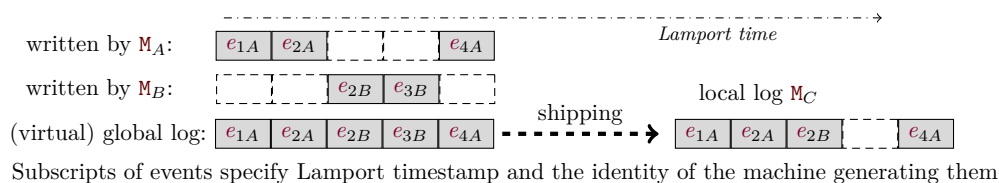
Note that in the automaton we abstract away from payloads, considering only the types of events. We also ignore internal computations not involving event emission/consumption (e.g. the computation of the constructor arguments for state `AuctionP` is immaterial).

2.2 Commands execution and events consumption

A machine can be thought of as the proxy of an agent (algorithm or human) that processes the information in the local log, comes to conclusions, and makes decisions which may lead to the invocation of an enabled command. For instance, in state `AuctionP`, the machine in Section 2.1 enables the passenger to execute the command `Select` triggering the emission of a log like `selected · passengerid`. This sequence of events is added to the local event log making the machine move to state 2 first by consuming the event `selected` and then to state `RideP` by consuming the event `passengerid`. This is why the inferred machine state diagram records commands as self-loops while only event consumption may induce state changes.

The automata representation of machines discussed in Section 2.1 allows us to limit technicalities in defining the behaviour of machines. We illustrate the notation given in Table 1 by abstracting the above example: a machine \mathbf{M} enabling command \mathbf{c} will upon invoking that command emit a sequence of events $e_1 \cdot e_2 \cdot \dots = l$ (called a log); the events are decidable typed as $\vdash e_i : \mathbf{t}_i$, with $\mathbf{t}_1 \cdot \mathbf{t}_2 \cdot \dots = \mathbf{l}$ being the log type associated with \mathbf{c} . We only consider deterministic machines, i.e. the labels of event transitions $\mathbf{t}?$ are pairwise distinct.

For the purpose of enabling commands a machine \mathbf{M} with log l is implicitly in a state denoted $\delta(\mathbf{M}, l)$. The determinism of \mathbf{M} ensures that there is a unique such state. $\delta(\mathbf{M}, l)$ is a *transition function* defined by adapting the standard transition function of finite-state automata. Starting with the initial state of \mathbf{M} we inductively remove the oldest event, say e ,



■ **Figure 1** Events sliced by their source: each event starts out at the machine where it is emitted. Logs are disseminated such that the recipient (like machine M_C) holds a prefix of each of the source slices, which is a partial view of the global log. The recipient’s local log is ordered like the global log. Eventually every event arrives at all machines, filling the transient gaps that may have existed.

and check it against the outgoing transitions of the current state: if M has a transition with label t ? and e has type t we transition to its target state, otherwise the event e is dropped; in either case we repeat this step until the log is empty.

It is important to note that a command may be invoked only if it is enabled in the state reached after fully processing the local log. Also, emitted events are *appended* to the local log of the machine they originate at. This ensures causality preservation since the new events are ordered *after* all events previously known by this machine.

2.3 Swarms and log-shipping

The last piece of our computational model is the mechanism for disseminating event logs among the machines of a swarm. This mechanism affects the behaviour of a recipient: as described in Section 2.2, the local log contains more events, leading to a new current state being computed, which in turn may change the set of available commands.

Our goal is eventual consensus between machines, in particular different replicas of the same machine must reach the same state when consuming the same events. According to the definition of the state transition function in Table 1, this can be achieved only if the events are ordered in the same way in the local logs. We address this by enforcing a total order between events, without requiring coordination between machines. As discussed in the previous section this total order preserves causality. Note that the ordering of events that are not yet locally known is arbitrary but well-defined, and to capture this concept we introduce the notion of a global log. Figure 1 illustrates the dissemination of logs, where each event begins in the local log of the emitting machine and simultaneously takes its place in the virtual global log based on the total order. We model log-shipping as a machine enlarging its local log with events from the global log; in practice, events flow from the local log of a machine to another machine’s local log. The precise algorithm for selecting the source and destination is not relevant to our theory. Due to the uncoordinated total order, it may happen that an incoming event is sorted into the middle of a local log, which can alter the interpretation of all subsequent events and affect the computation of the current state.

For example, consider the case in our running example in which the passenger selects the taxi at the same time that another taxi places a new bid. If the passenger’s selection is ordered before the bid, a later inspection of the log may reveal that the passenger selected suboptimally, but the selection still remains in effect and the bid is ignored. (In a system based on a central database the “bid” transaction would be rejected instead.) On the other hand, if the event *selected* were placed in between *bid* and *bidderid*, it would be ignored once the logs are replicated. In this case there are two reasonable paths forward: honouring the passenger’s previous wish would require a *compensating action* of executing the same selection again, or the selection could be redone including the new bid, possibly leading to a different outcome – this workflow choice needs to be made by the application designer.

■ **Table 2** Swarm semantics: coinductively unfold machines to inductively build up logs.

<p>Notation:</p> <p>\mathbf{S} : a list of pairs (\mathbf{M}_i, l_i)</p> <p>(\mathbf{S}, l) : a system (machines with local logs paired with a global log)</p> <p>κ : set of command invocations $\mathbf{c} / \mathbf{1}$</p> <p>$\Rightarrow = \rightarrow^*$ where $\rightarrow = \xrightarrow{\tau} \cup \bigcup_{\mathbf{c}, \mathbf{1}} \xrightarrow{\mathbf{c} / \mathbf{1}}$</p>	<p>Log merging:</p> <p>Let \sqsubseteq be the sublog relation of Definition 2.1:</p> $l_1 \bowtie l_2 = \{l \mid l \subseteq l_1 \cup l_2 \text{ and } l_1 \sqsubseteq l \text{ and } l_2 \sqsubseteq l\}$ <p>Machine step:</p> <p>If $\delta(\mathbf{M}, l)$ has a self-loop with label $\mathbf{c} / \mathbf{1}$ and $\vdash l' : \mathbf{1}$ then $(\mathbf{M}, l) \xrightarrow{\mathbf{c} / \mathbf{1}} (\mathbf{M}, l \cdot l')$.</p>
<p>Operational semantics:</p> $\frac{\mathbf{S}(i) = (\mathbf{M}, l_i) \quad (\mathbf{M}, l_i) \xrightarrow{\mathbf{c} / \mathbf{1}} (\mathbf{M}, l'_i) \quad \text{src}(l'_i \setminus l_i) = \{i\} \quad l' \in l \bowtie l'_i}{(\mathbf{S}, l) \xrightarrow{\mathbf{c} / \mathbf{1}} (\mathbf{S}[i \mapsto (\mathbf{M}, l'_i)], l')} \text{[LOCAL]}$ $\frac{\mathbf{S}(i) = (\mathbf{M}, l_i) \quad l_i \sqsubseteq l'_i \sqsubseteq l \quad l_i \subset l'_i}{(\mathbf{S}, l) \xrightarrow{\tau} (\mathbf{S}[i \mapsto (\mathbf{M}, l'_i)], l)} \text{[PROP]}$	

2.4 Formalisation

A *swarm* (of size n) is a pair (\mathbf{S}, l) where \mathbf{S} maps indices $1 \leq i \leq n$ to machines and their local log, i.e. $\mathbf{S}(i) = (\mathbf{M}_i, l_i)$ and l is the global log. It is convenient to let $(\mathbf{M}_1, l_1) \mid \dots \mid (\mathbf{M}_n, l_n) \mid l$ denote the swarm (\mathbf{S}, l) such that $\mathbf{S}(i) = (\mathbf{M}_i, l_i)$ for $1 \leq i \leq n$. A swarm $(\mathbf{M}_1, l_1) \mid \dots \mid (\mathbf{M}_n, l_n) \mid l$ is *coherent* when the local log l_i of each machine \mathbf{M}_i is made of events actually emitted in \mathbf{S} in the order in which they appear in the global log.

► **Definition 2.1** (Sublogs and coherence). *A log $l = e_1 \dots e_n$ induces a total order $<_l$ on its elements as follows: $e_i <_l e_j \iff i < j$. The sublog relation on logs \sqsubseteq demands an order-preserving and downward-complete morphism from l_1 into l_2 . Formally, $l_1 \sqsubseteq l_2$ if*

1. *all events of l_1 appear in l_2 ($l_1 \subseteq l_2$) in the same order ($<_{l_1} \subseteq <_{l_2}$); and*
2. *the per-source partitions of l_1 are prefixes of the corresponding partitions of l_2 , i.e. for all $e_1 \in l_1, e_2 \in l_2$ from a given src , $e_2 <_{l_2} e_1$ implies $e_2 \in l_1$.*

A swarm $(\mathbf{M}_1, l_1) \mid \dots \mid (\mathbf{M}_n, l_n) \mid l$ is coherent if $\bigcup_{1 \leq i \leq n} l_i = l$ and $l_i \sqsubseteq l$ for $1 \leq i \leq n$.

The operational semantics of swarms accounts for the construction of the global log, i.e. the total order defined over generated events. To model that this assignment is non-deterministic we rely on the merge operator $_ \bowtie _$ defined in Table 2 to combine two logs that may share events. This operator generates all possible logs that contain the events from both original logs while maintaining their input order.

The operational semantics is given by the rules LOCAL and PROP in Table 2. They respectively formalise the effects of command execution described in Section 2.2 and the non-deterministic log-shipping mechanism illustrated in Section 2.3. Recall that a machine \mathbf{M} with a log l is implicitly in state $\delta(\mathbf{M}, l)$. We can hence define the relation $(\mathbf{M}, l) \xrightarrow{\mathbf{c} / \mathbf{1}} (\mathbf{M}, l \cdot l')$ holding when the state $\delta(\mathbf{M}, l)$ enables the command $\mathbf{c} / \mathbf{1}$ and l' has type $\mathbf{1}$. The type of a log $\mathbf{1}$ is the sequence of the types of its events. We write $\vdash l : \mathbf{1}$ when l has type $\mathbf{1}$; this is decidable since the typing of events (cf. Table 1) is decidable.

Rule LOCAL describes the invocation of the command \mathbf{c} enabled at the i -th machine of the swarm (\mathbf{S}, l) . In addition to updating the local log of the i -th machine to l'_i , which extends l_i with the events generated by \mathbf{c} , the rule also updates the global log. The new global log now

includes the events generated by the i -th machine, assigning their place in the total order by picking one of the possible orders generated by the merge operator defined in Table 2. Rule PROP defines event log propagation between machines. The idea is to non-deterministically select a machine whose local log is a strict sublog of the global log, identify a larger sublog $l'_i \sqsubseteq l$, and transfer events to the machine by assigning l'_i as its new local log.

Note that our formalisation acts on the logs which grow by appending newly generated events. These features play an important role in the realisation of the local-first principle and permit to formally represent the conflicts discussed in Section 1.

Let \implies be the reflexive and transitive closure of the operational semantics relation (cf. Table 2). The following properties hold on coherent swarms.

- **Lemma 2.2** (Coherence preservation and eventual consistency). *Given a coherent swarm $(S, l) = (M_1, l_1) \mid \dots \mid (M_n, l_n) \mid l$ then*
- coherence preservation: $(S, l) \xrightarrow{\alpha} (S', l')$ implies that (S', l') is coherent
 - eventual consistency: $(S, l) \implies (M_1, l) \mid \dots \mid (M_n, l) \mid l$.

3 Tool support

Our theoretical development is accompanied by a set of software tools that support the implementation of swarms as a composition of type-checked TypeScript machines [30] and runs them based on the Actyx middleware [1]. The ecosystem is depicted in Figure 2.

3.1 Execution of compiled machines

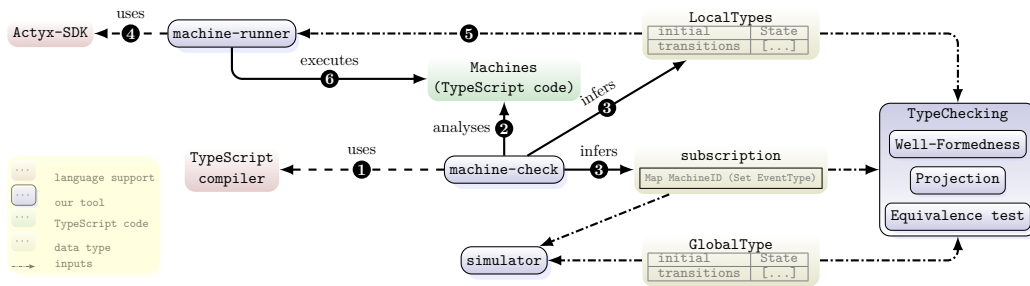
The `machine-runner` library uses the Actyx SDK (cf. arrow ④ in Figure 2) to drive machines written in TypeScript; more precisely, it employs ⑤ local types to interpret incoming events and execute ⑥ the corresponding machine logic. The declaration of a machine revolves around the event types that it can handle. Referring back to Listing 1, we show the `Requested` event type on line 2 as an example: using a property called `type` to hold a string of singleton type (here: `'Requested'`) is a customary way to express a *tagged union* in TypeScript, as shown on line 3.

Every machine state is represented by a class that derives from the `State` base class (or *prototype* in JavaScript terms) provided by the `machine-runner` library. This serves both as a marker for machine states and to carry the type parameter constraining all emitted events to a common type: the inherited `this.events` function used for example on line 28 is a utility for helping TypeScript to correctly capture the tuple type `[Selected, PassengerID]` (instead of the otherwise inferred array type `Events[]`) and assert that each of the event types conforms to type `Events`.

A program using the passenger’s machine would start by constructing the initial state using for example `new InitialP('myID')`. Together with a suitable set of Actyx event tags (like `'ride:12345'` to tag this particular taxi ride protocol session) and a state change callback (see below). This initial state is then passed to the library’s `runMachine` function. This will set up a subscription for events with those tags using the Actyx SDK, where Actyx will first deliver all historic events already locally known and then switch to live mode.

Whenever an event is received, it is slated for consumption by the appropriate event handler method; to this end the handler method for the event’s type is dynamically looked up in the JavaScript object underlying the state’s class. If that method takes only a single argument then the event is immediately consumed by calling the method, which returns the next state of the machine. Otherwise, the event is enqueued, awaiting the receipt of the following required event type etc. until the desired event sequence is complete and

15:10 Behavioural Types for Local-First Software



■ **Figure 2** Tool ecosystem.

the method can be invoked with all arguments. During this whole process, whenever the incoming event type does not have a matching handler or is not of the next required type in an argument list, the event is discarded as detailed in Section 2.2.

Whenever the machine state changes (i.e. when $\delta(M, l)$ computes a new value), the new state is passed to a function that the application passed to `runMachine` earlier – this scheme is termed a *callback* in TypeScript (note that this language implements an imperative style with mutable bindings). This could update a user interface or trigger an algorithm to compute reactions. The state’s command methods can therein be used to construct adequate event payloads for enabled commands, which would then be stored in Actyx using an SDK function and come back via the event subscriptions – now with metadata – to be applied to the current state and eventually trigger another invocation of the callback.

3.2 Enforcing typing at run-time

Readers versed in TypeScript may have noticed that we glossed over a difficulty here: TypeScript types are fully erased at runtime, meaning that the `machine-runner` code will not be able to find the event handler method by using the event type, and it will also not know how many arguments that method takes and what its types are. Therefore, the first responsibility of the `machine-check` build tool is to analyse ② the TypeScript code and ascertain that all event types are declared such that they can be recognised at runtime on their `type` property – the handler method’s name can then be constructed by prefixing the value of this property with `'on'`. The second responsibility is to extract the function signatures of all event handlers, check that each handler’s name corresponds to the name of its first argument type, and then construct a per-state mapping from first event type to the list of following events (possibly empty). This information is made available to `runMachine` by decorating [37] the user-written state class: the `@proto` decorator transforms the class definition as it is loaded by the JavaScript VM, overriding the `reactions` method inherited from the `State` prototype. To do that, the implementation of the `proto` function needs to access the `machine-check`’s extraction results. This is done by importing ⑤ a source module generated by `machine-check` that contains all protocol information in JSON format [36].

While the aforementioned duties of `machine-check` are crucial for `machine-runner`’s operation, the more interesting function of this build tool relates to the inference of local types and subscriptions (arrows ③ in Figure 2) as well as initiating the type-checking process on swarm protocols. To this end, the TypeScript compiler is used ① as a library to obtain ② a fully typed AST representation of the user program. Since we are only interested in machines, our entry points are `State` subclasses that are marked as initial states by a documentation

comment starting with “Initial state for role”, as is shown on line 5 of Listing 1. This comment serves the secondary purpose of naming the role this machine aspires to play (P for passenger in this example). The `@proto` decorator on line 6 not only has its runtime duties as explained above, it also carries in its argument the name of the swarm protocol that provides the context for the role name – we discuss both concepts in detail in the following sections; `machine-check` expects to find the definition of the swarm protocol in a correspondingly named file in JSON format. Finally, `machine-check` assembles the lists of command and event handler methods by inspecting (arrow ⑤) a state class’s method names and signatures and follows up with recursively processing the result types of event handlers in the same fashion. Any event type seen in an event handler argument list is automatically added to the subscription set of the machine (needed for the projection as explained in Section 5).

3.3 Type-checking, simulation and more

As a result of the analysis described above `machine-check` has assembled the following pieces for each machine definition within the user program: swarm protocol, role name, subscriptions, states, and transitions. Each such tuple is then passed – again in JSON format – to the `typechecking` tool, our third artifact contribution, written in Haskell. This tool first checks that the provided swarm protocol and subscription are well-formed (according to the rules presented in Section 6), computes the projection for the given role, and finally checks the inferred machine type for equivalence to the projection result (where state names are immaterial).

We provide a tool written in Haskell for the simulation of the formal operational semantics of the model. For a given protocol and subscription, the tool computes the projections and simulates the execution of swarms consisting of machines according to those projections. It supports both exhaustive and random generation of traces up to a given length. The tool has been used for checking claims and results about our running example.

The aforementioned tools are detailed in [28]. Through an example project, the accompanying paper also demonstrates the use of the inferred machine type to generically render a machine UI. Besides showing the current state of the computation, the UI gives the user the possibility to interact with machines by invoking enabled commands, where command arguments are gathered using automatically generated HTML forms.

4 Swarm protocols

A *swarm protocol* (hereafter also called *protocol* for short) describes the intended overarching event log structure realised by a swarm of machines; it corresponds to a global type in the terminology of session types, with our machines playing a similar role to local types. The protocol captures the overall communication structure as well as the details relevant for implementing it with machines. As it is customary with behavioural types, swarm protocols rely on an idealised environment where all communication is infallible and instantaneous. The link to the realisation in terms of machines is given in the following section by way of a projection operation.

When defining the syntax of swarm protocols, we follow the approach initiated in [7] that avoids fixing a syntactic representation of recursion and simplifies later treatment by instead using infinite regular trees. A swarm protocols is a possible infinite, *regular* term coinductively generated by the grammar in Table 3. A term is regular if it consists of finitely many *distinct* subterms. The language generated by the coinductive grammar is thus finitely representable either using the so-called “ μ notation” [31] or as solutions of finite sets of equations [9]. The

■ **Table 3** Swarm protocols: traverse a coinductive type to inductively build up an event log.

<p>Swarm protocol syntax: Regular terms generated by:</p> $\mathbf{G} ::= \sum_{i \in I} \mathbf{c}_i @ \mathbf{R}_i \langle \mathbf{l}_i \rangle . \mathbf{G}_i \mid \mathbf{0}$ <p>\mathbf{R}_i : role $\mathbf{c}, \mathbf{l}, l$: as for machines</p>	<p>State computation:</p> $\mathbf{G} \xrightarrow{\mathbf{c}_j / \mathbf{l}_j} \mathbf{G}_j \iff \mathbf{G} = \sum_{i \in I} \mathbf{c}_i @ \mathbf{R}_i \langle \mathbf{l}_i \rangle . \mathbf{G}_i \text{ and } j \in I$ $\delta(\mathbf{G}, l) = \begin{cases} \mathbf{G} & \text{if } l = \epsilon \\ \delta(\mathbf{G}', l'') & \text{if } \mathbf{G} \xrightarrow{\mathbf{c}/\mathbf{l}} \mathbf{G}' \text{ and } \vdash l' : \mathbf{l} \text{ and } l = l' \cdot l'' \\ \perp & \text{otherwise} \end{cases}$
<p>A swarm protocol is well-formed (Def. 6.7) if it is causal consistent (Def. 6.1), choice determinate (Def. 6.3), and confusion-free (Def. 6.5).</p>	
<p>Operational semantics:</p>	$\frac{\delta(\mathbf{G}, l) \xrightarrow{\mathbf{c}/\mathbf{l}} \mathbf{G}' \quad \vdash l' : \mathbf{l} \quad l' \text{ log of fresh events}}{(\mathbf{G}, l) \xrightarrow{\mathbf{c}/\mathbf{l}} (\mathbf{G}, l \cdot l')} \text{[G-CMD]}$

interested reader is referred to [9] for a comprehensive treatment. Intuitively, the protocol progresses by some role \mathbf{R}_i invoking command \mathbf{c}_i , appending a non-empty event sequence of type \mathbf{l}_i to the global log and continuing as protocol \mathbf{G}_i . As discussed at the end of Section 2.4, the resolution of the choice specified in a swarm protocol is not coordinated among the instances of the roles involved in the choice (i.e. there is no unique selector). In fact, instances of different roles involved in the choice may enable commands at the same time as well as different instances of the same role may enable different commands (recall that each machine tracks a separate local log and event replication is asynchronous). This is in contrast to most other behavioural type systems hitherto, which do not permit such race conditions.

► **Definition 4.1** (Determinism). *A protocol $\mathbf{G} = \sum_{i \in I} \mathbf{c}_i @ \mathbf{R}_i \langle \mathbf{l}_i \rangle . \mathbf{G}_i$ is log-deterministic if the event types $\mathbf{l}_i[0]$ are pairwise different and all \mathbf{G}_i are log-deterministic. \mathbf{G} is command-deterministic if the tuples $(\mathbf{c}_i, \mathbf{R}_i)$ are pairwise different and all \mathbf{G}_i are command-deterministic. \mathbf{G} is deterministic if it is log-deterministic and command-deterministic.*

Hereafter we only consider deterministic swarm protocols. Note that determinism is evidently decidable on swarm protocols due to the regularity constraint.

► **Example 4.2** (Taxi service). The swarm protocol for the scenario in Example 1.1 is

$$\begin{aligned} \mathbf{G} &= \text{Request}@P \langle \text{Requested} \rangle . \mathbf{G}_{\text{auction}} \\ \mathbf{G}_{\text{auction}} &= \text{Offer}@T \langle \text{Bid} \cdot \text{BidderID} \rangle . \mathbf{G}_{\text{auction}} + \text{Select}@P \langle \text{Selected} \cdot \text{PassengerID} \rangle . \mathbf{G}_{\text{choose}} \\ \mathbf{G}_{\text{choose}} &= \text{Arrive}@T \langle \text{Arrived} \rangle . \text{Start}@P \langle \text{Started} \rangle . \mathbf{G}_{\text{ride}} + \text{Cancel}@P \langle \text{Cancelled} \rangle . \text{Receipt}@O \langle \text{Receipt} \rangle . \mathbf{0} \\ \mathbf{G}_{\text{ride}} &= \text{Record}@T \langle \text{Path} \rangle . \mathbf{G}_{\text{ride}} + \text{Finish}@P \langle \text{Finished} \cdot \text{Rating} \rangle . \text{Receipt}@O \langle \text{Receipt} \rangle . \mathbf{0} \end{aligned}$$

The structure in terms of commands and roles is straightforwardly induced from Example 1.1, with event log types filled in according to further requirements. The event type **BidderID** represents identifying information illustrating that not all events are of interest to all roles (e.g., the office does not need to know all bidders, it only needs to know which taxi was **Selected**). It is straightforward to check that \mathbf{G} is both log- and command-deterministic. \lrcorner

Mirroring the formulation of machines we ascribe operational semantics to a swarm protocol via the generation and processing of an event log. The main difference in state computation is that swarm protocols generate and consume logs instead of events, as illustrated with the **Offer**, **Select**, and **Finish** commands in the example above.

The state reached by a swarm protocol \mathbf{G} after processing a log l is computed using an extension of the transition function δ as defined in Table 3. Analogously to the definition for machines, the transition function $\delta(\mathbf{G}, l)$ returns the continuation of the swarm protocol \mathbf{G} after processing the entire log l . We stress that δ is a partial function on swarm protocols, it is undefined when log l cannot be generated according to \mathbf{G} (unlike the definition of δ on machines, which is a total function since it just discharges unrecognised events). Note also that δ is well-defined over log-deterministic swarm protocols because a log in a branch cannot appear as a prefix of the logs of the remaining branches of the choice.

The operational semantics of a swarm protocol is defined as a labelled transition system given by rule [G-CMD] in Table 3. This rule states that a swarm protocol \mathbf{G} with log l enables command c , upon whose invocation the log is extended with fresh events l' of type $\mathbf{1}$ before possibly allowing another command to be invoked. The freshness of the events in l' can e.g. be guaranteed by the inclusion of node ID and logical timestamp.

► **Example 4.3** (Idealised taxi service). Consider the protocol from Example 4.2, starting out with (\mathbf{G}, ϵ) . After invoking the **Request** command our log contains *requested* and we reach state $\mathbf{G}_{\text{Auction}}$. Two bids later the passenger makes their selection, leading us to

$$\delta(\mathbf{G}, \text{requested} \cdot \text{bid}_A \cdot \text{bidderid}_A \cdot \text{bid}_B \cdot \text{bidderid}_B \cdot \text{selected} \cdot \text{passengerid}) = \mathbf{G}_{\text{choose}}$$

with $\mathbf{G}_{\text{choose}}$ offering two options: either the passenger invokes **Cancel** or the taxi **Arrives**. ◻

5 Projection

For the definition of our projection operation it is convenient to introduce a textual presentation of machines equivalent to the automata-based presentation used so far. Let κ denote a finite function mapping commands to non-empty log types; we allow ourselves to treat κ as set (the graph of function κ) and e.g. write $c / \mathbf{1} \in \kappa$ for $\kappa(c) = \mathbf{1}$ or else write $\{c_1 / \mathbf{1}_1, \dots, c_h / \mathbf{1}_h\}$ for the function κ mapping c_i to $\mathbf{1}_i$ for each $i \in \{1, \dots, h\}$. t_i ranges over event types.

Similarly to swarm protocols, the textual presentation of our machines is a regular term² of the following coinductive grammar:

$$\mathbf{M} ::=^{\text{co}} \kappa \cdot [t_1?M_1 \ \& \ \dots \ \& \ t_n?M_n] \tag{1}$$

and we abbreviate $\kappa \cdot [t_1?M_1 \ \& \ \dots \ \& \ t_n?M_n]$ as $\kappa \cdot \mathbf{0}$ when $n = 0$ and as $t_1?M_1 \ \& \ \dots \ \& \ t_n?M_n$ when κ is the empty map. We also write $\&_{1 \leq i \leq n} \mathbf{1}_i?M_i$ in place of $t_1?M_1 \ \& \ \dots \ \& \ t_n?M_n$.

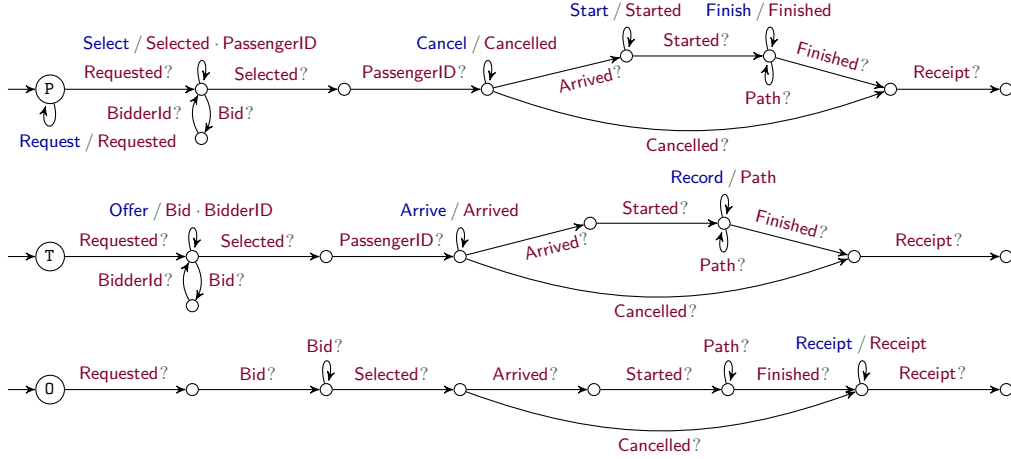
In turning our attention to projection operations we first note that the responsibility for driving the protocol forward is distributed across the participants: each transition in the swarm protocol is labelled with one role that may trigger it by invoking the command. Each machine plays one role, whose machine specification is obtained by the projection operation $\mathbf{G} \downarrow_{\mathbf{R}}$. Note that multiple machines may implement the same role.

One could define $\mathbf{G} \downarrow_{\mathbf{R}}$ such that each transition in \mathbf{G} produces a series of event transitions in the machine plus a command invocation on the originating state if the role matches.

$$\left(\sum_{i \in I} c_i @_{\mathbf{R}_i} \langle \mathbf{1}_i \rangle \cdot \mathbf{G}_i \right) \downarrow_{\mathbf{R}} = \kappa \cdot [\&_{i \in I} \mathbf{1}_i? \mathbf{G}_i \downarrow_{\mathbf{R}}] \quad \text{where} \quad \kappa = \{(c_i / \mathbf{1}_i) \mid \mathbf{R}_i = \mathbf{R} \text{ and } i \in I\}$$

² The correspondence between these regular terms and finite-state automata yields exactly the presentation of machines in terms of finite-state automata that we have adopted so far.

15:14 Behavioural Types for Local-First Software



■ **Figure 3** Projection of Example 4.2 on P, T, and O as automata.

Albeit simple, this projection scheme generates unnecessarily large machines in all but the most trivial cases. More crucially, forcing each machine to process all events is undesirable for reasons of security and efficiency. It would be highly desirable to allow some information to be kept secret from certain roles (like `passengerID` in Example 4.2), and it would be most efficient if every role processed just enough information to correctly enable and disable command invocations. We therefore define a more appealing construction.

Our projections are based on the notion of whether a machine shall process a certain type of event. Formally, the projection operation is parameterised by a *subscription*, namely a map σ assigning to each role the set of event types that it reacts to. Given a set of log types E , let $filter(_, E)$ be a function transforming a log type, retaining only the event types in E while preserving their relative order. Intuitively, subscriptions correspond to topics in a publish–subscribe framework whereby processes declare which kinds of messages they are interested in receiving.

► **Definition 5.1** (Projection). *Given a swarm protocol G and a subscription σ , the projection of G over a role R with respect to σ , written $G \downarrow_R^\sigma$, is defined as follows:*

$$\left(\sum_{i \in I} c_i @ R_i \langle \mathbf{1}_i \rangle \cdot G_i \right) \downarrow_R^\sigma = \{ c_i / \mathbf{1}_i \mid R_i = R \text{ and } i \in I \} \cdot [\&_{j \in J} filter(\mathbf{1}_j, \sigma(R)) ? (G_j \downarrow_R^\sigma)]$$

where $J = \{i \in I \mid filter(\mathbf{1}_i, \sigma(R)) \neq \epsilon\}$.

Notice that we omit the projection of a branch when a role R is not subscribed to any of the event types emitted by the command that selects that branch. We opted for this simplification of the formalism because our well-formedness conditions (cf. Section 6) ensure that if a role is involved in the continuation it will subscribe to the the first event in the branch. Further, note that this pruning applies to branches in isolation, later states reachable by other paths remain part of the projection.

► **Example 5.2.** Let σ be such that $\sigma(P)$ consists of all the event types in the protocol G defined in Example 4.2. The projections of G are in Figure 3. ┘

6 Well-formedness

We now focus on the *well-formedness* conditions of our swarm protocols. As is standard in behavioural types, sufficient conditions are established on global specifications that guarantee relevant properties on projections such as deadlock or lock freedom and absence of orphan messages. The properties of interest to us are quite different from those common in standard settings since we aim to guarantee that eventual consensus is reached even when some of the participants make choices that are discordant due to their incomplete view on the global log. The idea is that transitory deviations are *tolerated* provided that consistency is eventually recovered, which happens once information has sufficiently spread within the swarm. For instance, a taxi in our running example may keep bidding for a passenger's auction after the passenger has made their selection as long as the selection event has not yet been received. This temporary inconsistency is recognised and resolved once the events have propagated to the deviating taxi and the passenger, respectively.

Realising swarms with this property is not straightforward. The rest of this section illustrates the problems arising in our setting with a few examples. For each problem we identify sufficient conditions on our swarm protocols that rule out the problem for coordination issues in realistic scenarios based on our running example). These conditions culminate in our definition of *well-formedness* (cf. Definition 6.7).

6.1 On causality and propagation

The first problem we look at is related to how a command is disabled once it has been invoked. In our setting, this boils down to fine tuning the registration of roles to event types. For example, if a command c should be enabled only after another, say c' has been executed, then the role executing c' should be subscribed to some event type emitted in response to the execution of c . Another example is that a command can stay perpetually enabled if the role executing it is oblivious of all resulting events (cf. [27]).

Another class of problems is caused by the fact that events propagate asynchronously within a swarm and that an emission of multiple events is not guaranteed to reach all other machines as one atomic transmission (cf. [27]) This anomaly may be excluded by a runtime system that never applies the [PROP] rule to a strict subset of the event log emitted by a single command, i.e. it treats the log from each command invocation as an atomic unit. We chose to not restrict the way in which a runtime system should propagate events between network sites because we consider it important that implementations be free to optimise their strategy in different ways (e.g. for latency, bandwidth, efficiency, or consistency).

We define the *active* roles of a swarm protocol G as those that can select one of the branches in the top-level choice of G and – given a subscription σ – the *roles* of G as those that can invoke commands or are subscribed to events in G . Formally,

$$\begin{aligned} \text{active}\left(\sum_{i \in I} c_i @ R_i \langle \mathbf{1}_i \rangle . G_i\right) &= \{R_i \mid i \in I\} \\ \text{roles}\left(\sum_{i \in I} c_i @ R_i \langle \mathbf{1}_i \rangle . G_i, \sigma\right) &= \bigcup_{i \in I} (\{R \in \text{dom } \sigma \mid R_i = R \vee \mathbf{1}_i \cap \sigma(R) \neq \emptyset\} \cup \text{roles}(G_i, \sigma)) \end{aligned}$$

(note that the latter is a coinductive definition). With this notation we define the following sufficient condition for avoiding the aforementioned problems.

► **Definition 6.1** (Causal consistency). *A swarm protocol $\sum_{i \in I} c_i @ R_i \langle \mathbf{1}_i \rangle . G_i$ is causal-consistent in a subscription σ if for all $i \in I$*

1. $\mathbf{1}_i \cap \sigma(\mathbf{R}_i) \neq \emptyset$, and
2. $\mathbf{R} \in \text{active}(\mathbf{G}_i)$ implies $\mathbf{1}_i \cap \sigma(\mathbf{R}) \neq \emptyset$ and for all $\mathbf{R}' \in \text{roles}(\mathbf{G}_i, \sigma)$, $\mathbf{1}_i \cap \sigma(\mathbf{R}') \subseteq \mathbf{1}_i \cap \sigma(\mathbf{R})$

Condition (1) requires that the role that performs one of the commands c_i should observe some of the corresponding emitted events $\mathbf{1}_i$. This simple mechanism ensures that repeated command invocation can only occur where foreseen in the swarm protocol. Condition (2) ensures the adequate tracking of causality for subsequent command invocations. The first part ensures that the immediately following command must wait for the enabling transition to occur, while the second part guarantees the ordering of the subsequent command's generated events after all events from the preceding command that are observed by some role in the further evolution of the protocol.

► **Example 6.2.** The protocol of the running example \mathbf{G} in Example 4.2 is causal-consistent for the subscription σ in Example 5.2. In fact, the commands generate logs that start with pairwise-different event types. Hence, the conditions straightforwardly hold for roles P and T, which observe every event. For O, we observe that they only execute the command `receipt`; which should be performed after `Cancelled` or `Finished`, which are also observed by O. ◻

6.2 On distributed choices

The next anomaly we study is caused by the fact that our model permits multiple roles to be active at the same time without coordination – this property is essential for perfect availability as demanded by local-first cooperation. Such behaviour would be ruled out in all the global type systems we are aware of. Our strategy for coping with the inevitably arising conflicts is that we permit machines to make inconsistent local decisions but reconcile those once the corresponding events have propagated to all relevant parties (e.g., the office in our example can make a choice inconsistent with the decisions taken by other participants as shown in see [27]). We fix this by requiring *determinacy*.

► **Definition 6.3** (Determinacy). *A swarm protocol $\mathbf{G} = \sum_{i \in I} c_i @ \mathbf{R}_i \langle \mathbf{1}_i \rangle$. \mathbf{G}_i is determinate for subscription σ if it is causal-consistent and $\mathbf{R} \in \text{roles}(\mathbf{G}_i, \sigma)$ implies $\mathbf{1}_i[0] \in \sigma(\mathbf{R})$ for $i \in I$.*

This definition of determinacy is prompted by our determinism rule (Definition 4.1): we identify a branch by the first event type of its emitted log. Note that a role involved in one branch but not in another may invoke commands that are later invalidated without that role being able to recognise this situation; we will explore mechanisms for compensating such errors in future work, which may require strengthening the rule above.

► **Example 6.4.** The swarm protocol \mathbf{G} in Example 4.2 is determinate for the subscription σ in Example 5.2. ◻

6.3 On interference

Events emitted by the losing parties to a conflict should be ignored in order to let every machine eventually agree on each choice. Each machine must locally be able to ignore such events, which means that it would be problematic to confuse a machine by emitting such a branch-choosing event in another context (e.g. while proceeding along a sibling branch which this machine does not follow). We avoid this confusion by requiring that any branch of a swarm protocol is communicated using a dedicated event type, i.e. that event type cannot be emitted by any other command. We formulate this notion in terms of the set *subterms*(\mathbf{G}) of all subterms (incl. indirect) of a swarm protocol \mathbf{G} . Recall that this set is finite because our swarm protocols are regular.

In what follows we write $events(\mathbf{G})$ and $guards(\mathbf{G})$ respectively for the sets of all event types and the ones that identify branches; formally, if $\mathbf{G} = \sum_{i \in I} c_i @ \mathbf{R}_i \langle \mathbf{1}_i \rangle . \mathbf{G}_i$ then

$$events(\mathbf{G}) = \bigcup_{i \in I} \left(events(\mathbf{G}_i) \cup \bigcup_j \mathbf{1}_i[j] \right) \quad \text{and} \quad guards(\mathbf{G}) = \bigcup_{i \in I} (\{\mathbf{1}_i[0]\} \cup guards(\mathbf{G}_i))$$

(observe that $events(\mathbf{0}) = guards(\mathbf{0}) = \emptyset$ and that these coinductively defined sets correspond to computable greatest fixpoint since swarm protocols are regular trees).

A swarm protocol \mathbf{G} is *invariant under event type* \mathbf{t} if either (i) \mathbf{t} does not appear in \mathbf{G} , i.e. $\mathbf{t} \notin events(\mathbf{G})$ or (ii) it only appears as part of the same choice, i.e. there is a unique $\mathbf{G}' \in subterms(\mathbf{G})$ such that $\mathbf{G}' \xrightarrow{c/1}$ and $\mathbf{t} \in \mathbf{1}$.

► **Definition 6.5** (Confusion-freeness). *A swarm protocol \mathbf{G} is confusion-free if \mathbf{G} is invariant for all event types in $guards(\mathbf{G})$.*

► **Example 6.6.** It is easy to check that the protocol of the running example \mathbf{G} in Example 4.2 is invariant for all types. The only type appearing in two guards is **Receipt**; however, the occurrences are associated to the same subterm. Hence, the protocol is confusion-free. ◻

6.4 Putting the pieces together

With this, we can finally state our *well-formedness* condition.

► **Definition 6.7** (Well-formedness). *A swarm protocol $\mathbf{G} = \sum_{i \in I} c_i @ \mathbf{R}_i \langle \mathbf{1}_i \rangle . \mathbf{G}_i$ is well-formed with respect to a subscription σ (σ -WF for short) if*

1. \mathbf{G} is causal-consistent, determinate, and confusion-free; and
2. \mathbf{G}_i is σ -WF for all $i \in I$;

Note that well-formedness is defined coinductively and decidable on swarm protocols.

► **Example 6.8.** Examples 6.2, 6.4, and 6.6 imply that the protocol \mathbf{G} in Example 4.2 is well-formed with respect to the subscription σ given in Example 5.2. ◻

Projection preserves determinism in well-formed protocols:

► **Proposition 6.9.** *Let \mathbf{G} and σ respectively be a swarm protocol and a subscription. If \mathbf{G} is σ -WF then $\mathbf{G} \downarrow_{\mathbf{R}}^{\sigma}$ is deterministic for all \mathbf{R} .*

Well-formed swarm protocols guarantee that local machines reach eventual consensus [38] on each choice, as we show next. However, anomalies (cf. [27]) occur at system level:

- Machines could have commands enabled that would be disabled if the model were synchronous; this may lead to the emission of events that need to be ignored later.
- Events are ignored according to their type only, therefore even after full propagation of the events in the global log a machine may process events stemming from the anomalous invocation of a command.

We note that the first anomaly above is inherent to local-first architecture requirements.

The second anomaly can be avoided by a runtime system that tracks full causality information. We chose to not require full causality tracking since it imposes additional storage, communication, and computation requirements on the implementation. Our weaker causality model supports deployment on less capable hardware where needed.

7 Correct Realisations of swarm protocols

We now turn our attention to the formal characterisation of *correct implementations* of swarm protocols. As discussed in the previous sections, we deviate from the usual expected properties of mainstream (multiparty) session types, such as communication safety, session fidelity, and progress (i.e., absence of deadlocks or its variants). We first note that there are no communication mismatches in our model because every machine simply ignores unexpected or unwanted events (recall the definition of δ in Section 2.1). Session fidelity instead advocates implementations that behave as described by their types, which customarily means that the states of all components are always aligned with the global state of the protocol. Contrastingly, we aim to tolerate deviations provided that all machines eventually agree on the state of the execution of the protocol. In our setting, an implementation may be correct even if machines temporarily diverge, executing different branches of the protocol; this is quite expected if we allow independent decisions taken based on incomplete views of the global state. Consequently, correct implementations may perform sequences of commands – and hence generate logs – that are different from those derived from the corresponding protocol. In such cases, we still expect machines to be able to eventually agree on an interpretation of the log that matches one possible execution of the specification.

We tackle this problem by first defining the relevant events of an execution, namely those that are part of the *effective log*. Based on this, we establish an equivalence relation on logs that allows us to characterise the logs that can be produced by an execution of a swarm protocol's realisation as a swarm. Armed with these tools we then state that all correct realisations produce valid effective logs and that all swarm protocol executions have corresponding swarms that realise them.

7.1 Eventual fidelity

We start by introducing some machinery for making precise the notion of correct implementation of a swarm protocol.

Roughly, one may think that (\mathbf{S}, ϵ) is a faithful implementation of a swarm protocol \mathbf{G} if it produces only global logs that can be generated by \mathbf{G} . However, this notion is too strong for our setting; in fact, we appeal to a weaker notion of fidelity such that for any global log l produced by (\mathbf{S}, ϵ) , i.e. $(\mathbf{S}, \epsilon) \Longrightarrow (\mathbf{S}, l)$, there is a *related* log l' that \mathbf{G} admits, i.e. $(\mathbf{G}, \epsilon) \Longrightarrow (\mathbf{G}, l')$. We postpone for a moment the formal definition of the expected relation between logs, and convey some intuitions in the following example.

► **Example 7.1.** Consider the swarm protocol \mathbf{G} in Example 4.2, and the swarm $(\mathbf{P}, \epsilon) \mid (\mathbf{T}, \epsilon) \mid (\mathbf{O}, \epsilon) \mid (\mathbf{T}, \epsilon) \mid \epsilon$ having three taxis dubbed A , B , and C . The swarm can produce the global log

$$l_{\text{auc}} = \text{requested} \cdot \text{bid}_B \cdot \text{bidderid}_B \cdot \text{bid}_A \cdot \text{bidderid}_A \cdot \text{selected} \cdot \text{bid}_C \cdot \text{bidderid}_C \cdot \text{passengerid}$$

Contrastingly, \mathbf{G} cannot generate such log; in fact, the protocol continuation after generating the prefix $l_1 = \text{requested} \cdot \text{bid}_B \cdot \text{bidderid}_B \cdot \text{bid}_A \cdot \text{bidderid}_A$ is $\delta(\mathbf{G}, l_{\text{auc}}) = \mathbf{G}_{\text{Bid}}$; hence, log l_1 can only grow by appending $\text{bid}_C \cdot \text{bidderid}_C$ or $\text{selected} \cdot \text{passengerid}$. In the second case, we obtain the log $l_2 = \text{requested} \cdot \text{bid}_B \cdot \text{bidderid}_B \cdot \text{bid}_A \cdot \text{bidderid}_A \cdot \text{selected} \cdot \text{passengerid}$. Remarkably, all the machines discard the events bid_C and bidderid_C when processing l_{auc} , i.e., they behave as if they were processing l_2 . In fact, $\delta(\mathbf{P}, l_{\text{auc}}) = \delta(\mathbf{P}, l_2)$, $\delta(\mathbf{T}, l_{\text{auc}}) = \delta(\mathbf{T}, l_2)$ and $\delta(\mathbf{O}, l_{\text{auc}}) = \delta(\mathbf{O}, l_2)$. \lrcorner

As highlighted by the previous example, despite the actual log generated by the swarm differing from the logs generated by the protocol, all the machines are able to consistently discard those ill-generated events after complete propagation. In other words, the states of the machines in the swarm depend only on a subset of the events in the log. We characterise such subset via a type, called *effective type*. Intuitively, the effective type of a log is the type of the sublog containing all those events that are effectively relevant to the machines in the swarm. Given a protocol \mathbf{G} and a subscription σ , we expect an implementation to process only those events which some role has been subscribed to; consequently, our notion of *effective type* is relative to a subscription. However, the effective type of a log is not just the projection of its type with respect to the image of the subscription σ . This is illustrated in the following example.

► **Example 7.2** (Effective type and projection). The type of the log l_{auc} in Example 7.1 is

$$\mathbf{l}_{\text{auc}} = \text{Requested} \cdot \text{Bid} \cdot \text{BidderID} \cdot \text{Bid} \cdot \text{BidderID} \cdot \text{Selected} \cdot \text{Bid} \cdot \text{BidderID} \cdot \text{PassengerID}$$

which differs from the type of l_2 which is $\text{Requested} \cdot \text{Bid} \cdot \text{BidderID} \cdot \text{Bid} \cdot \text{BidderID} \cdot \text{Selected} \cdot \text{PassengerID}$. Let us consider a subscription σ such that $\sigma(\mathbf{P}) \supseteq \{\text{Requested}, \text{Bid}, \text{BidderID}, \text{Selected}, \text{PassengerID}\}$. Then, if we just keep the sublog of \mathbf{l}_{auc} containing all types for which at least one role has been subscribed to, then we obtain exactly \mathbf{l}_{auc} , which does not reflect the type of the sublog that is effectively processed by the machines. For this reason, the effective type depends also on the protocol being implemented. \lrcorner

► **Definition 7.3** (Effective type). *The effective type of a log l with respect to a swarm protocol \mathbf{G} and a subscription σ , written $\mathbb{T}_\sigma(l, \mathbf{G}) = \mathbb{T}_\sigma(l, \mathbf{G}, \epsilon)$, is defined as follows*

$$\mathbb{T}_\sigma(\epsilon, \mathbf{G}, \mathbf{l}) = \epsilon \quad (2)$$

$$\mathbb{T}_\sigma(e \cdot l, \mathbf{G}, \epsilon) = \mathbf{t} \cdot \mathbb{T}_\sigma(l, \mathbf{G}', \mathbf{l}') \quad \text{if } \vdash e : \mathbf{t}, \mathbf{t} \in \sigma(\text{roles}(\mathbf{G}, \sigma)), \mathbf{G} \xrightarrow{\mathbf{c}/\mathbf{t} \cdot \mathbf{l}} \mathbf{G}', \text{ and } \mathbf{l}' = \text{filter}(\mathbf{l}, \sigma(\text{active}(\mathbf{G}')))) \quad (3)$$

$$\mathbb{T}_\sigma(e \cdot l, \mathbf{G}, \mathbf{t} \cdot \mathbf{l}) = \mathbf{t} \cdot \mathbb{T}_\sigma(l, \mathbf{G}, \mathbf{l}) \quad \text{if } \vdash e : \mathbf{t} \quad (4)$$

$$\mathbb{T}_\sigma(e \cdot l, \mathbf{G}, \mathbf{l}) = \mathbb{T}_\sigma(l, \mathbf{G}, \mathbf{l}) \quad \text{otherwise} \quad (5)$$

As expected, the effective type of an empty log is the empty log type. The effective type of a non-empty log keeps track of those events that match the type of a log that can be generated by the protocol (and discards all ill-ordered events). Hence, the effective type of $e \cdot l$ with respect to \mathbf{G} records the type \mathbf{t} of the first event e only if it has a type that is expected by the protocol (namely at least one of the roles in the protocol is subscribed to that type). According to case (3), a protocol \mathbf{G} expects some event whose type \mathbf{t} coincides with the guard of one of its branches and at least one role is subscribed to \mathbf{t} . In such case, the effective type of the remaining log l is processed first by consuming events of type \mathbf{l}' (rule (4)), which is the sequence of the remaining types generated by the branch that are observed by the *active* roles in the continuation \mathbf{G}' , followed by considering the continuation \mathbf{G}' . We remark here that only the types of events that are relevant for active roles are reflected in the effective type (more details are given in Section 7.2.1). Note that the types that are not observed are just disregarded from the effective type as per (5). We have that $\mathbb{T}_\sigma(l, \mathbf{G})$ is a well-defined function over deterministic swarm protocols because log-determinism (Definition 4.1) ensures that at most one branch of \mathbf{G} can match the event type \mathbf{t} of the event e .

► **Example 7.4.** Let $\mathbf{G}_{\text{auction}}$ and $\mathbf{G}_{\text{choose}}$ the swarm protocol in Example 4.2. We have

$$\mathbf{G}_{\text{auction}} \xrightarrow{\text{Offer}@\mathbf{T}(\text{Bid} \cdot \text{BidderID})} \mathbf{G}_{\text{auction}} \quad (6) \quad \mathbf{G}_{\text{auction}} \xrightarrow{\text{Select}@\mathbf{T}(\text{Selected} \cdot \text{PassengerID})} \mathbf{G}_{\text{choose}} \quad (7)$$

15:20 Behavioural Types for Local-First Software

Let us compute the effective type of the log $l = \text{bid}_A \cdot \text{bidderid}_A \cdot \text{rating} \cdot l'$ on $\mathbf{G}_{\text{auction}}$ using a subscription σ for which \mathbf{P} , \mathbf{T} and $\mathbf{0}$ are subscribed to all events but rating . We have

$$\begin{aligned} \mathbb{T}_\sigma(l, \mathbf{G}_{\text{auction}}) &= \mathbf{Bid} \cdot \mathbb{T}_\sigma(\text{bidderid}_A \cdot \text{rating} \cdot l', \mathbf{G}_{\text{auction}}, \mathbf{BidderID}) \\ &= \mathbf{Bid} \cdot \mathbf{BidderID} \cdot \mathbb{T}_\sigma(\text{rating} \cdot l', \mathbf{G}_{\text{auction}}, \epsilon) \\ &= \mathbf{Bid} \cdot \mathbf{BidderID} \cdot \mathbb{T}_\sigma(l', \mathbf{G}_{\text{auction}}, \epsilon) \end{aligned}$$

where the first equality holds by (3) since $\mathbf{Bid} \in \sigma(\mathbf{T})$ and (6), the second equality holds by (4) since $\vdash \text{bidderid}_A : \mathbf{BidderID}$, and the third equation holds by (5) since by hypothesis $\text{rating} \notin \sigma(\mathbf{P}) \cup \sigma(\mathbf{T}) \cup \sigma(\mathbf{0})$. If $l' = \epsilon$ then $\mathbb{T}_\sigma(l, \mathbf{G}_{\text{auction}}) = \mathbf{Bid} \cdot \mathbf{BidderID}$ by (2).

Suppose instead that $l' = \text{selected} \cdot \text{passengerid} \cdot \text{bid}_B \cdot \text{bidderid}_B$. Then, similarly to the first two equations above (using (7)), we have

$$\mathbb{T}_\sigma(l', \mathbf{G}_{\text{auction}}, \epsilon) = \mathbf{Selected} \cdot \mathbf{PassengerID} \cdot \mathbb{T}_\sigma(\text{bid}_B \cdot \text{bidderid}_B, \mathbf{G}_{\text{choose}}, \epsilon)$$

And, by (4), $\mathbb{T}_\sigma(\text{bid}_B \cdot \text{bidderid}_B, \mathbf{G}_{\text{choose}}, \epsilon) = \mathbb{T}_\sigma(\epsilon, \mathbf{G}_{\text{choose}}, \epsilon) = \epsilon$. \lrcorner

The relation between logs of an implementation with those of a specification that we need is the equivalence induced by the equality of their effective types.

► **Definition 7.5** (Log equivalence). *Two logs l and l' are equivalent with respect to a swarm protocol \mathbf{G} and a subscription σ , written $l \equiv_{\mathbf{G}, \sigma} l'$, if they have the same effective type with respect to \mathbf{G} and σ , i.e., $\mathbb{T}_\sigma(l, \mathbf{G}) = \mathbb{T}_\sigma(l', \mathbf{G})$.*

Then, the notion of correct implementation is simply stated as follows.

► **Definition 7.6** (Eventual fidelity). *A swarm (\mathbf{S}, ϵ) is eventually faithful to a swarm protocol \mathbf{G} and a subscription σ if $(\mathbf{S}, \epsilon) \implies (\mathbf{S}, l)$ implies $(\mathbf{G}, \epsilon) \implies (\mathbf{G}, l')$ and $l \equiv_{\mathbf{G}, \sigma} l'$ for a log l' .*

7.2 Implementation correctness by projection

Our projection operation yields an effective procedure for obtaining correct implementations out of well-formed swarm protocols which we call *realisations*.

► **Definition 7.7** (Realisation). *Let \mathbf{G} be a swarm protocol and σ be a subscription. A realisation (of size n) of \mathbf{G} with respect to σ , shortened as (σ, \mathbf{G}) -realisation, is a swarm (\mathbf{S}, ϵ) of size n such that, for each $1 \leq i \leq n$, there exists a role $\mathbf{R} \in \text{roles}(\mathbf{G}, \sigma)$ such that $\mathbf{S}(i) = (\mathbf{G} \downarrow_{\mathbf{R}}^\sigma, \epsilon)$. A realisation \mathbf{S} is complete if for all $\mathbf{R} \in \text{roles}(\mathbf{G}, \sigma)$ there exists $1 \leq i \leq n$ such that $\mathbf{S}(i) = (\mathbf{G} \downarrow_{\mathbf{R}}^\sigma, \epsilon)$; we call partial a realisation which is not complete.*

Remarkably, the number of machines in a realisation is not related to the number of roles in the corresponding swarm protocol. Indeed, Definition 7.7 simply requires that each machine in the swarm plays one of the roles in the swarm protocol. Concretely, we may have several components implementing the same role (i.e., the role is replicated) as well as roles without a corresponding machine, that is partial realisations.

► **Example 7.8** (Realisations). The swarm protocol in Example 4.2 would typically be realised by one machine for the passenger $\mathbf{G} \downarrow_{\mathbf{P}}^\sigma$, several taxis running the machine $\mathbf{G} \downarrow_{\mathbf{T}}^\sigma$, and at least one accounting office running $\mathbf{G} \downarrow_{\mathbf{0}}^\sigma$. A partial realisation could be one without an accounting office, in which case no machine can generate **receipt** events. \lrcorner

The rest of this section is devoted to showing that realisations (either complete or partial) are eventually faithful if they are obtained by projecting well-formed swarm protocols.

7.2.1 Projections and effective types

We first establish a correspondence between the behaviour of a single projection and that of the respective protocol. In particular, we show that effective types provide an accurate abstraction of the information contained in a log that is relevant for a role. Concretely, the next result states that a projection enables a command after processing a log l only when the protocol enables the same command after producing an equivalent log l' .

► **Lemma 7.9.** *If G is a σ -WF swarm protocol and $(\delta(G \downarrow_R^\sigma, l)) \downarrow_{c/1}$ then there exists $l' \equiv_{G,\sigma} l$ such that $(G, \epsilon) \Longrightarrow (G, l')$ and $\delta(G, l') \xrightarrow{c/1} G'$.*

Apparently equivalent logs are indistinguishable for a machine, i.e., $l \equiv_{G,\sigma} l'$ implies $\delta(G \downarrow_R^\sigma, l) = \delta(G \downarrow_R^\sigma, l')$. However this might not be the case if logs do not include all the events generated by the same command as shown in the next example.

► **Example 7.10.** Consider the swarm protocol $G = c@R\langle a \cdot b \rangle . d@S\langle c \rangle . 0$ with $\sigma = \{R \mapsto \{a, b\}, S \mapsto \{a, c\}\}$. If $\vdash a : a$ and $\vdash b : b$ then $\mathbb{T}_\sigma(a, G) = a$ and $\mathbb{T}_\sigma(a \cdot b, G) = a$; in fact the first equation holds by definition and the second holds because $b \notin \sigma(\text{active}(d@S\langle c \rangle . 0')) = \sigma(S) = \{a, c\}$. Therefore, $a \equiv_{G,\sigma} a \cdot b$. Now take the projection of G over R with respect to σ , i.e., $M_R = G \downarrow_R^\sigma = a?b?(d@S\langle c \rangle . 0 \downarrow_R^\sigma) = a?b?0$. Clearly $\delta(M_R, a \cdot b) \neq \delta(M_R, a)$. ◻

Two considerations on Example 7.10 are worthwhile. On the one hand, while $a \cdot b$ has all the events produced by the execution of the command c , the log consisting of just the event a does not. Since we assume that all events are eventually propagated, our technical development in the next section will disregard incomplete (global) logs. On the other hand, one may wonder about the fact that effective types do not collect information of events that are not observed by active roles. This is essential to account for the fact that a realisation may interject events. For instance, a realisation may actually generate a log of type $a \cdot c \cdot b$ because a machine that implements the role S may perform the command d as soon as it processes an event of type a ; hence the generated event can precede the one of type b in the consolidated log. If our notion of log equivalence were fine enough to distinguish logs of type $a \cdot c \cdot b$ from $a \cdot b \cdot c$ then we would rule out implementations behaving as above, which is not what we want because the interaction does not violate the protocol.

7.2.2 Characterisation of the logs admitted by a protocol

We now provide a characterisation of the logs that can be generated by a realisation of a well-formed swarm protocol. To do this we have to take into account for the possible reordering and the spurious events that can be generated by machines that faithfully implement a protocol. Intuitively, we may think that a realisation generates logs that correspond to the combination of several executions of the protocol, which might share a common prefix. Consider again the swarm protocol G in Example 4.2. As discussed in Example 7.1, we expect realisations to be able to generate logs such as l_{auc} of this example. Note that such log can be generated by merging, among others, two different reductions of G , e.g. $(G, \epsilon) \Longrightarrow (G, \text{requested} \cdot \text{bid}_B \cdot \text{bidderid}_B \cdot \text{bid}_A \cdot \text{bidderid}_A \cdot \text{selected} \cdot \text{passengerid})$ and $(G, \epsilon) \Longrightarrow (G, \text{requested} \cdot \text{bid}_B \cdot \text{bidderid}_B \cdot \text{bid}_C \cdot \text{bidderid}_C)$. Note that the reductions share events (accounting for an scenario in which the computation has diverged). Intuitively, two runs can be combined either if they produce disjoint logs or they share events that come from a common execution prefix (as in the previous example). Formally, two runs $(G, \epsilon) \Longrightarrow (G, l_i)$ with $i = 1, 2$ are *consistent* if there exist logs $l, l'_1 \cap l'_2 = \emptyset$, such that $l_i = l \cdot l'_i$ and $(G, \epsilon) \Longrightarrow (G, l) \Longrightarrow (G, l \cdot l'_i)$

for $i = 1, 2$. The notion of consistency is lifted to sets of runs $\{(\mathbf{G}, \epsilon) \Longrightarrow (\mathbf{G}, l_i)\}_{1 \leq i \leq k}$, by requiring pair-wise consistency. We write l_i^j for the sequence of events produced by the j -th step in the reduction i , i.e., $(\mathbf{G}, \epsilon) \Longrightarrow^{j-1} (\mathbf{G}, l_i^j) \xrightarrow{c/1} (\mathbf{G}, l_i^j \cdot l_i^j) \Longrightarrow (\mathbf{G}, l_i)$.

► **Definition 7.11** (Admissible log). *A log l is admissible for a σ -WF protocol \mathbf{G} if there are consistent runs $\{(\mathbf{G}, \epsilon) \Longrightarrow (\mathbf{G}, l_i)\}_{1 \leq i \leq k}$ and a log $l' \in (\boxtimes_{1 \leq i \leq k} l_i)$ with $l' \equiv_{\mathbf{G}, \sigma} l = \bigcup_{1 \leq i \leq k} l_i$, and for all $1 \leq i \leq k$ and $l_i^j \sqsubseteq l$ for all events l_i^j produced by the j -th step in reduction i .*

Remarkably, admissible logs are not just those that can be obtained by merging several logs l_i ; it may be the case that l is admissible but $l \notin (\boxtimes_{1 \leq i \leq k} l_i)$. In fact the notion is weaker and accounts for the possible reorderings of events that do not change the effective type of the log. Consider the protocol in Example 7.10 and its complete realisation consisting of two machines. As previously discussed, that realisation may generate a log of type $a \cdot c \cdot b$. With a single run of the protocol, i.e. by fixing $k = 1$ and taking $(\mathbf{G}, \epsilon) \xrightarrow{c/a \cdot b} (\mathbf{G}, a \cdot b) \xrightarrow{d/c} (\mathbf{G}, l_1)$ with $l_1 = a \cdot b \cdot c$, we can conclude that $a \cdot c \cdot b$ is generated by some realisation. Note that $\boxtimes_{1 \leq i \leq 1} l_i = \{l_1\} = \{a \cdot b \cdot c\}$. Hence, $l' \in \boxtimes_{1 \leq i \leq 1} l_i$ iff $l' = a \cdot b \cdot c$ and $\mathbb{T}_\sigma(l', \mathbf{G}) = a \cdot c$. Then, the log $a \cdot c \cdot b$ is equivalent (i.e., it has the same effective type), and moreover it has the same elements and preserves the relative order between events generated by the same command (i.e., a precedes b). Hence, we conclude that the protocol admits the log $a \cdot c \cdot b$. On the contrary, the last condition $l_i^j \sqsubseteq l$ about the preservation of the relative order of events generated by the same command bans logs such as $b \cdot a \cdot c$.

Next lemma ensures that any admissible log of a well-formed protocol is equivalent to a log obtained by the sequential execution of the protocol. Namely, despite a log may contain events produced by conflicting decisions, its effective type corresponds to a sequential run.

► **Lemma 7.12.** *If l is admissible for a σ -WF swarm protocol \mathbf{G} then there exists a log l' such that $(\mathbf{G}, \epsilon) \Longrightarrow (\mathbf{G}, l')$ and $l \equiv_{\mathbf{G}, \sigma} l'$.*

Moreover, if we extend an admissible log with events generated by the execution of command enabled over a partial view of the global log, then we obtain an admissible log. This property is instrumental for our main result in the following section (Theorem 7.14).

► **Lemma 7.13.** *Let l_1 and $l_2 \subseteq l_1$ be admissible logs for a σ -WF swarm protocol \mathbf{G} . If $(\mathbf{G}, l_2) \xrightarrow{c/1} (\mathbf{G}, l_2 \cdot l_3)$ and $l \in l_1 \boxtimes (l_2 \cdot l_3)$ then l is admissible for \mathbf{G} .*

7.2.3 Realisations are faithful

Our key result shows that realisations of well-formed protocols only generate admissible logs.

► **Theorem 7.14.** *Let (\mathbf{S}, ϵ) be a realisation of a σ -WF swarm protocol \mathbf{G} . If $(\mathbf{S}, \epsilon) \Longrightarrow (\mathbf{S}', l)$ then l is admissible for \mathbf{G} .*

Since every admissible log is equivalent to a log generated by the protocol (Lemma 7.12), we conclude that any realisation of a well-formed swarm protocol is eventually faithful (i.e., correct). Note that this implies that all realisations of a well-formed swarm protocol exhibit eventual consensus [38] regarding which branch is taken in its choices (concretely, there is a $\xrightarrow{\tau}$ step after which all machines take the same branch in their state computation).

► **Corollary 7.15.** *Every realisation of size n of a σ -WF swarm protocol \mathbf{G} is eventually faithful with respect to \mathbf{G} and σ .*

The above result is independent from the number of replicas that implement each role; it also holds for partial realisations (i.e., when some roles are absent).

7.3 Implementation completeness

Differently from common session type systems, the behaviour of a complete realisation (i.e., one in which every role is implemented) is complete with respect to the protocol, in the sense that every reduction of the protocol can be mimicked by the realisation. This derives from the fact that non-determinism in our model arises from the execution of external commands but not because of the abstraction of internal (and customary deterministic) choices. Firstly, we note that logs that are generated sequentially according to the protocol drives the machines to the corresponding states.

► **Lemma 7.16.** *If G is σ -WF swarm protocol and $(G, \epsilon) \Longrightarrow (G, l)$ then $\delta(G \downarrow_R^\sigma, l) = \delta(G, l) \downarrow_R^\sigma$ for all $R \in \text{roles}(G, \sigma)$.*

Moreover, Proposition 7.17 below states that a complete realisation is able to generate the logs that are generated by the protocol (the result is obtained by using previous result and by propagating all events to all replicas right after a machine performs a command).

► **Proposition 7.17.** *Let (S, ϵ) be a complete realisation of size n of the σ -WF swarm protocol G . If $(G, \epsilon) \Longrightarrow (G, l)$ then there is a swarm S' such that $(S, \epsilon) \Longrightarrow (S', l)$.*

8 Related work

It is widely accepted that solutions to distributed coordination problems strongly depend on the adopted computation model [15, 21, 2]. Our proposal is grounded on the principles of *local-first cooperation* [26, 25]. Key to this architecture is the *autonomy* of each participating node within a swarm. Autonomy allows each node to make progress independently of network connections, availability of other nodes, or delay in the communications. Our target model features specific hues that distinguish it from other behavioural types systems. In our case, distributed heterogeneous components interact asynchronously by emitting and consuming *events* according to a role specified in a given protocol (such as passenger and the taxi in our running example). More precisely, events are the side effects of commands non-deterministically executed by components; events are locally logged by each of the components and asynchronously spread through the swarm. Crucially, we do not make any assumption on the relative speed of communications and simply require that logs *eventually* agree on the order of events [4]. This liberal setting permits inconsistencies: components may take discordant decisions which compromise the execution of the protocol and exhibit behaviour precluded in strongly consistent models. Our approach lies within methods related to data replication, which are notoriously complex. In fact, standard techniques have to trade-off among availability, consistency, and partition tolerance [16]. Several techniques have been proposed, such as conflict-free replicated data types [34], cloud types [5], consistency contracts [35], invariants [18, 24, 3], linearizability [40], and operational models for applications such as GSP [6, 17]. An original facet of our approach is that we use behavioural types to discipline data replication in order to eventually reach consistency. We focus on the consequences that arise from the ability of each node to take decisions based exclusively on *local information*.

Our proposal is inspired by the choreographic framework introduced in the seminal work on *multiparty session types* [19, 20]. However, the peculiarities of our execution model as well as on the properties that we target require a radical change in the definition of well-established notions of global types, such as projections and well-formedness. The main originality of our approach is that components speculatively proceed along several (possibly inconsistent)

branches of distributed choices provided that an agreement is eventually reached. Intuitively, this is attained by disregarding all executions bar one when the local logs “consolidate”, namely when relevant events have propagated to all relevant components. As far as we are aware of, multiple selectors are forbidden in the well-formedness conditions of most behavioural type systems [21]. A slight weakening of this condition is given in [23] but the conditions there still reject the protocol in Example 1.1. Noteworthy, we divert from the research path of behavioral types with respect to the properties we are after. We aim to guarantee that projections of global specifications yield realisations of swarms that eventually reach a consistent view of the distributed execution, even in presence of transitory inconsistencies. This is in contrast with behavioural type systems designed to attain (dead)lock freedom or some notions of progress (see [21] and [10] for a recent account on the binary case).

Secondly, our behavioural specifications completely abstract over the number of instances enacting a role. This is often not the case for multiparty session types. Parametric multiparty session types have been considered in [41, 11, 12] and more recently in [8, 23]. These proposals aim to capture the fact that roles in a protocol are “connected” to form a topology that can be generalised (e.g. parameterising a ring topology by its size). These behavioural type systems therefore require to explicitly handle the parameters of the protocol. Our specifications are instead completely oblivious of such parameters. To the best of our knowledge, multiparty session types have focused on point-to-point, message-passing communication model, even to deal with highly dynamic scenarios, as those involving robot coordination [29].

9 Final Remarks

We proposed rather unconventional behavioural types deviating from point-to-point, message-passing communication, which is a common practice (see e.g. [13, 39, 21]). Components in our setting interact via a shared distributed log built without any further coordination mechanism. More precisely, each component keeps a local, possibly partial and inconsistent view of the global log. Based on that view alone a component may perform an action with immediate effect on its local state; those effects are then propagated asynchronously to the rest of the system. This implies that components can perform globally invalid actions (as long as they are locally valid), but we require every component be able to recognise these and eventually behave as if only valid actions were performed. Technically this means that we renounce established properties like *session fidelity* to guarantee instead that systems eventually agree without dedicated coordination (our typing discipline guarantees deadlock-freedom, though).

Our target applications intrinsically involve sets of components whose number is statically unknown: components may dynamically join and leave the execution of an interaction. A reference application domain is factory logistics where all the assumptions listed at the end of Section 1 apply: collaborative components act in a trustworthy setting, compensations are specified for irreversible actions, and the underlying communication infrastructure is controlled by the business owner. The collaborative assumption, while commonplace in the literature on behavioural types, may be unrealistic in some domains. Extension to adversarial settings will for example require enforcing that machines cannot violate causality when emitting events, i.e. they cannot artificially truncate their local log to undo an earlier choice via an event that they maliciously sort before it. This can be achieved by requiring them to cryptographically sign their logs, allowing other nodes to prove illicit behaviour; similarly, machines joining later would be required to sign recent events before being allowed to emit.

Common practice in behavioural types is to describe protocols in terms of the roles that components may play. Unlike most behavioural types, ours are agnostic to the number of instances playing each role. We assume that any role in any swarm protocol can be replicated

as many times as needed. This choice also impacts the interpretation of choices. In standard approaches, every choice is assigned to a single role implemented by a single component responsible for coordinating the decision. This is problematic when the implementation of a role can be replicated, even more when the states of the replicas may be misaligned: different components may decide differently. Consequently, choices in our swarm protocols are intended to be resolved distributedly among components that may implement different roles. Our solution is based on speculative computation: different choices can proceed concurrently until components are able to agree (by inspecting their local state) on the branch that has been selected based on a total order for all events (implemented for example using Lamport timestamps and unique node identifiers).

Our computational model gives an abstract description of Actyx infrastructure [1], in which machines are actually implemented as programs in `TypeScript`: the machines presented here play the role of local types that describe the intended behaviour of each component.

Subscriptions can be seen as a minimum requirement for which events need to be available to each participant in a swarm protocol. This directly translates to which events need to (or shall) be sent to a swarm member participating in a given protocol session. Swarm members not partaking in the session do not need to see any of the events (e.g. other passengers), so the middleware should not send them there (this can be elevated to a security guarantee if needed). Our system does not cause additional information to be sent, it is minimal within the constraints of our well-formedness conditions (which are sufficient but not necessary, so there is some room for further improvement).

Determining a suitable subscription could be hard in general. We conjecture that the swarm protocol itself could be used to infer a suitable “minimal” registration enforcing well-formedness to be suggested to designers. Such subscription could then manually be refined, provided that well-formedness is preserved. Moreover, we may envision programmers specifying just the relevant information that needs to be transmitted and then automatically infer the events needed for coordinating choices (pretty much in the style of the communication of labels in session types).

An underlying assumption about speculative execution is that the effects of performing invalid actions can be discarded. In other words, invalid actions have no consequences. In several situations this may be unacceptable. Swarm protocols could be used to systematically identify such situations – e.g. by noting when corresponding events are disregarded by machines – and enable principled treatment at the application level. We plan to study suitable extensions for our projections that automatically inject the required behaviour for executing amending actions. Alternatively, we will explore monitoring approaches equipped with sanitisers responsible for compensation.

We have only partly addressed failures in our model: while we do model transient inability to receive (which would inhibit the event propagation transition [PROP]) or to operate (i.e. inhibit command invocation [LOCAL]), we do not model permanent inability to send. In the presence of a stop failure a machine could communicate the first event of a choice but then fail in propagating all the expected following events resulting from the command invocation, in which case the system could get stuck. A fix for this issue could be to only proceed with an external choice once all specified events are present in the local log, allowing the swarm to permanently discard a choice made by a failing machine; this could be expressed by ingesting logs instead of single events in the definition of machine semantics.

Another worthwhile extension, hinted at in Section 2.3, is to achieve a per-choice notion of non-interference if the first event of a choice not only decided which branch to take but also from which source machine to consume the rest of the choice’s events. This would

further strengthen the failure handling sketched above by making sure that inputs from failing machines are consistently discarded. Characterising the precise guarantees that derive from such a scheme will be an interesting topic for further study.

References

- 1 Actyx AG. Actyx developer website, 2020-2022. accessed 2022/07/06. URL: <https://developer.actyx.com>.
- 2 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniérou, Simon J Gay, Nils Gesbert, Elena Giachino, Raymond Hu, et al. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016.
- 3 Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno M. Preguiça. IPA: invariant-preserving applications for weakly consistent replicated databases. *Proc. VLDB Endow.*, 12(4):404–418, 2018. doi:10.14778/3297753.3297760.
- 4 Sebastian Burckhardt. Principles of eventual consistency. *Found. Trends Program. Lang.*, 1(1–2):1–150, October 2014. doi:10.1561/2500000011.
- 5 Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud types for eventual consistency. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, pages 283–307, Berlin, Heidelberg, 2012. Springer.
- 6 Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 568–590, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2015.568.
- 7 Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(5):1–61, 2009.
- 8 David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed programming using role-parametric session types in go: statically-typed endpoint apis for dynamically-instantiated communication structures. *Proc. ACM Program. Lang.*, 3(POPL):29:1–29:30, 2019. doi:10.1145/3290342.
- 9 Bruno Courcelle. Fundamental properties of infinite trees. *Theor. Comput. Sci.*, 25:95–169, 1983. doi:10.1016/0304-3975(83)90059-2.
- 10 Ornela Dardha and Jorge A. Pérez. Comparing type systems for deadlock freedom. *J. Log. Algebraic Methods Program.*, 124:100717, 2022. doi:10.1016/j.jlamp.2021.100717.
- 11 Pierre-Malo Deniérou and Nobuko Yoshida. Dynamic multirole session types. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 435–446. ACM, 2011. doi:10.1145/1926385.1926435.
- 12 Pierre-Malo Deniérou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. *Log. Methods Comput. Sci.*, 8(4), 2012. doi:10.2168/LMCS-8(4:6)2012.
- 13 Mariangiola Dezani-Ciancaglini and Ugo de’Liguoro. Sessions and session types: An overview. In *WS-FM*, volume 6194 of *LNCS*, pages 1–28. Springer, 2009. doi:10.1007/978-3-642-14458-5_1.
- 14 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. doi:10.1145/3149.214121.
- 15 Simon Gay and Antonio Ravara, editors. *Behavioural Types: from Theory to Tools*. Automation, Control and Robotics. River, 2009.
- 16 Seth Gilbert and Nancy Lynch. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. doi:10.1145/564585.564601.

- 17 Alexey Gotsman and Sebastian Burckhardt. Consistency Models with Global Operation Sequencing and their Composition. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.DISC.2017.23.
- 18 Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. ‘Cause i’m strong enough: reasoning about consistency choices in distributed systems. In *POPL 2016*, pages 371–384, 2016.
- 19 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on principles of programming languages*, pages 273–284, 2008.
- 20 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM (JACM)*, 63(1):1–67, 2016.
- 21 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniérou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016.
- 22 David R Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, 1985.
- 23 Sung-Shik Jongmans and Nobuko Yoshida. Exploring type-level bisimilarity towards more expressive multiparty session types. In Peter Müller, editor, *Programming Languages and Systems*, pages 251–279, Cham, 2020. Springer International Publishing.
- 24 Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. Safe replication through bounded concurrency verification. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018.
- 25 Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: You own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, pages 154–178, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3359591.3359737.
- 26 Roland Kuhn. Local-first cooperation: Autonomy at the edge, secured by crypto, 100% available, 2021. accessed 2021/06/20. URL: <https://www.infoq.com/articles/local-first-cooperation/>.
- 27 Roland Kuhn, Hernán Melgratti, and Emilio Tuosto. Behavioural types for local-first software, 2023. arXiv:2305.04848.
- 28 Roland Kuhn, Hernán Melgratti, and Emilio Tuosto. Behavioural Types for Local-First Software (Artifact). *Dagstuhl Artifacts Series*, 9(2):14:1–14:5, 2023. doi:10.4230/DARTS.9.2.14.
- 29 Rupak Majumdar, Marcus Pirron, Nobuko Yoshida, and Damien Zufferey. Motion session types for robotic interactions (brave new idea paper). In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
- 30 Microsoft. Typescript: Javascript with syntax for types, 2012-2023. accessed 2023/02/02. URL: <https://www.typescriptlang.org/>.
- 31 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.
- 32 Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.
- 33 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer, 2011.
- 34 Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *SSS 2011*, pages 386–400, 2011. doi:10.1007/978-3-642-24550-3_29.

- 35 KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *PLDI 2015*, pages 413–424. ACM, 2015.
- 36 Ecma TC39. Ecma-404, 2017. accessed 2023/02/13, alternative RFC8259. URL: <https://www.ecma-international.org/publications-and-standards/standards/ecma-404/>.
- 37 Ecma TC39. Decorators, 2022. accessed 2023/02/02. URL: <https://github.com/tc39/proposal-decorators>.
- 38 Lewis Tseng. Eventual consensus: Applications to storage and blockchain : (extended abstract). In *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 840–846, 2019. doi:10.1109/ALLERTON.2019.8919675.
- 39 Vasco T. Vasconcelos. Fundamentals of session types. *Inf. and Comp.*, 217:52–70, 2012. doi:10.1016/j.ic.2012.05.002.
- 40 Chao Wang, Constantin Enea, Suha Orhun Mutluergil, and Gustavo Petri. Replication-aware linearizability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 980–993, 2019.
- 41 Nobuko Yoshida, Pierre-Malo Deniérou, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. In C.-H. Luke Ong, editor, *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6014 of *Lecture Notes in Computer Science*, pages 128–145. Springer, 2010. doi:10.1007/978-3-642-12032-9_10.