Local-First Principles: a Behavioural Types Approach

Emilio Tuosto @ GSSI

joint work with

Roland Kuhn @ Actyx

and

Hernán Melgratti @ UBA

Tutorial at Discotec 2023
Lisbon 23 June, 2023
– Prelude –
To trade consistency for availability in systems of \textit{asymmetric replicated peers}
Take-away message

To trade consistency for availability in systems of **asymmetric replicated peers**

you can use **local-first**'s principles to (re-)gain **consistency** ... eventually
To trade consistency for availability in systems of asymmetric replicated peers you can use local-first’s principles to (re-)gain consistency ... eventually

And get some support by our behavioural typing discipline!
Take-away message

To trade consistency for availability in systems of **asymmetric replicated peers**

you can use *local-first*’s principles to (re-)gain consistency ... eventually

And get some support by our behavioural typing discipline!

- **swarm protocols**: systems from a **global** viewpoint
- **machines**: peers
- enforce **good behaviour** via behavioural typing
Take-away message

To trade consistency for availability in systems of **asymmetric replicated peers**
you can use local-first’s principles to (re-)gain **consistency** ... eventually

And get some support by our behavioural typing discipline!

- **swarm protocols**: systems from a **global** viewpoint
- **machines**: peers
- enforce **good behaviour** via behavioural typing

See our recent ECOOP 2023 paper (to appear; extended version available at
Distributed coordination

An “old” problem
- Distributed agreement
- Distributed sharing
- Security
- Computer-assisted collaborative work
...
Distributed coordination

An “old” problem
- Distributed agreement
- Distributed sharing
- Security
- Computer-assisted collaborative work
...

With some “solutions”
- Centralisation points
- Distributed consensus
- Commutative replicated data types
...

consistency
availability
partitioning
Local-first...first

Autonomy
Thou shall be autonomous
Thou shall collaborate
Thou shall recognise and embrace conflicts
Thou shall resolve conflicts
Thou shall be consistent

Some implications
- peers are not malicious
- peers can progress at all times...even under partial knowledge
- purity: inconsistencies resolved by “replaying” executions (invertible or compensatable actions)
- reliable communications
Alice and Bob decided to have spaghetti carbonara and tiramisù. They use a mobile app to agree on a grocery list and decide who buys what.
Alice and Bob decided to have spaghetti carbonara and tiramisù. They use a mobile app to agree on a grocery list and decide who buys what.

<table>
<thead>
<tr>
<th>Alice’s mobile</th>
<th>Bob’s mobile</th>
</tr>
</thead>
<tbody>
<tr>
<td>mascarpone cheese</td>
<td>smoked guanciale</td>
</tr>
<tr>
<td>eggs</td>
<td></td>
</tr>
</tbody>
</table>
Alice and Bob decided to have spaghetti carbonara and tiramisù. They use a mobile app to agree on a grocery list and decide who buys what.

<table>
<thead>
<tr>
<th>Alice’s mobile</th>
<th>Bob’s mobile</th>
</tr>
</thead>
<tbody>
<tr>
<td>mascarpone cheese</td>
<td>smoked guanciale</td>
</tr>
<tr>
<td>eggs</td>
<td>eggs</td>
</tr>
<tr>
<td>sugar</td>
<td></td>
</tr>
</tbody>
</table>
Alice and Bob decided to have spaghetti carbonara and tiramisù.
They use a mobile app to agree on a grocery list and decide who buys what.

<table>
<thead>
<tr>
<th>Alice’s mobile</th>
<th>Bob’s mobile</th>
</tr>
</thead>
<tbody>
<tr>
<td>mascarpone cheese</td>
<td>smoked guanciale</td>
</tr>
<tr>
<td>eggs</td>
<td>eggs</td>
</tr>
<tr>
<td>sugar</td>
<td>pecorino romano cheese</td>
</tr>
</tbody>
</table>
Alice and Bob decided to have spaghetti carbonara and tiramisù. They use a mobile app to agree on a grocery list and decide who buys what.

<table>
<thead>
<tr>
<th>Alice’s mobile</th>
<th>Bob’s mobile</th>
</tr>
</thead>
<tbody>
<tr>
<td>mascarpone cheese</td>
<td>smoked guanciale</td>
</tr>
<tr>
<td>eggs</td>
<td>eggs</td>
</tr>
<tr>
<td>sugar</td>
<td>pecorino romano cheese</td>
</tr>
<tr>
<td></td>
<td>spaghetti</td>
</tr>
</tbody>
</table>
Alice and Bob decided to have spaghetti carbonara and tiramisù. They use a mobile app to agree on a grocery list and decide who buys what.

<table>
<thead>
<tr>
<th>Alice’s mobile</th>
<th>Bob’s mobile</th>
</tr>
</thead>
<tbody>
<tr>
<td>mascarpone cheese</td>
<td>smoked guanciale</td>
</tr>
<tr>
<td>eggs</td>
<td>eggs</td>
</tr>
<tr>
<td>sugar</td>
<td>pecorino romano cheese</td>
</tr>
<tr>
<td>ground moka coffee</td>
<td>spaghetti</td>
</tr>
</tbody>
</table>
Alice and Bob decided to have spaghetti carbonara and tiramisù. They use a mobile app to agree on a grocery list and decide who buys what.

<table>
<thead>
<tr>
<th>Alice’s mobile</th>
<th>Bob’s mobile</th>
</tr>
</thead>
<tbody>
<tr>
<td>mascarpone cheese</td>
<td>smoked guanciale</td>
</tr>
<tr>
<td>eggs</td>
<td>eggs</td>
</tr>
<tr>
<td>sugar</td>
<td>pecorino romano cheese</td>
</tr>
<tr>
<td>ground moka coffee</td>
<td>spaghetti</td>
</tr>
<tr>
<td>savoiardi biscuits</td>
<td></td>
</tr>
</tbody>
</table>
Alice and Bob decided to have spaghetti carbonara and tiramisù. They use a mobile app to agree on a grocery list and decide who buys what.

<table>
<thead>
<tr>
<th>Alice’s mobile</th>
<th>Bob’s mobile</th>
</tr>
</thead>
<tbody>
<tr>
<td>mascarpone cheese</td>
<td>smoked guanciale</td>
</tr>
<tr>
<td>eggs</td>
<td>eggs</td>
</tr>
<tr>
<td>sugar</td>
<td>pecorino romano cheese</td>
</tr>
<tr>
<td>ground moka coffee</td>
<td>spaghetti</td>
</tr>
<tr>
<td>savoiardi biscuits</td>
<td></td>
</tr>
</tbody>
</table>

Eventually the lists can be merged somehow...but who’s going to buy the eggs?
Plan of the talk

A motivating case study

Our formalisation

Our typing discipline

Tool support

Open issues
A collaborative environment and its execution model
A collaborative environment and its execution model

People + Real-time controllers + IT systems and networks:
A collaborative environment and its execution model

People + Real-time controllers + IT systems and networks:
- work divided among autonomous production cells
A collaborative environment and its execution model

People + Real-time controllers + IT systems and networks:
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
A collaborative environment and its execution model

People + Real-time controllers + IT systems and networks:

- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on
A collaborative environment and its execution model

People + Real-time controllers + IT systems and networks:
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

Execution model

(local twin for each device/operator)
(twins are replicated where needed)
(events have unique IDs and record facts (e.g., from sensors) or decisions (e.g., from an operator))
(spread information asynchronously)
(logs are local to twins)
(a log determines the computational state of its twin)
(replicated logs are merged)
A collaborative environment and its execution model

People + Real-time controllers + IT systems and networks:
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

Execution model
- local twin for each device/operator
A collaborative environment and its execution model

People + Real-time controllers + IT systems and networks:
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

Execution model
- local twin for each device/operator
- twins are replicated where needed
A collaborative environment and its execution model

People + Real-time controllers + IT systems and networks:
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

Execution model
- local twin for each device/operator
- twins are replicated where needed
- events have unique IDs and
  - record facts (e.g., from sensors) or
  - decisions (e.g., from an operator)
- spread information asynchronously

(execution model details: local twin, event IDs, facts, decisions, asynchronous information spread)
A collaborative environment and its execution model

People + Real-time controllers + IT systems and networks:
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

Execution model:
- local twin for each device/operator
- twins are replicated where needed
- events have unique IDs and
  - record facts (e.g., from sensors) or
  - decisions (e.g., from an operator)
- spread information asynchronously
- logs are local to twins

(Images are courtesy of Actyx AG)
A collaborative environment and its execution model

People + Real-time controllers + IT systems and networks:
- Work divided among autonomous production cells
- Efficiency is determined by designing and controlling the flow of resource and information
- If disconnected, keep calm, and move on

Execution model:
- Local twin for each device/operator
- Twins are replicated where needed
- Events have unique IDs and
  - Record facts (e.g., from sensors) or decisions (e.g., from an operator)
  - Spread information asynchronously
- Logs are local to twins
- A log determines the computational state of its twin
A collaborative environment and its execution model

People + Real-time controllers + IT systems and networks:
- work divided among autonomous production cells
- efficiency is determined by designing and controlling the flow of resource and information
- if disconnected, keep calm, and move on

Execution model
- local twin for each device/operator
- twins are replicated where needed
- events have unique IDs and
  - record facts (e.g., from sensors) or
  - decisions (e.g., from an operator)
  - spread information asynchronously
- logs are local to twins
- a log determines the computational state of its twin
- replicated logs are merged
execute
+
propagate
+
merge
Other application domains / motivations

**More applications**

Robots (e.g., rescue missions or space applications)

Collaborative applications ([https://automerge.org/](https://automerge.org/))

Home automation
Other application domains / motivations

IoT...really?

Why your fridge and mobile should go in the cloud to talk to each other?
Other application domains / motivations

“Anytime, anywhere...” really?

like the AWS’s outage on 25/11/2020
or almost all Google services down on 14/12/2020

DSL typical availability of 97% (& some SLA have no lower bound)
checkout https://www.internetsociety.org/blog/2022/03/what-is-the-digital-divide/
Other application domains / motivations

Also, taking decisions locally can reduce downtime

shifts data ownership

gets rid of any centralization point...for real
Challenges

Specify application-level protocols where decisions
- don’t require consensus
Specify application-level protocols where decisions

- don’t require consensus
- are based on stale local states
Specify application-level protocols where decisions

- don’t require consensus
- are based on stale local states
- yet, collaboration has to be successful
Plan of the talk

A motivating case study

Our formalisation

Our typing discipline

Tool support

Open issues
– A formal model –
Ingredients (I): events & logs

Events

\[ e \]

\[ src(e) \]

Logs

\[ e_1 \cdot e_2 \cdots \]
Ingredients (I): events & logs

Events

\[ e : t \]

\[ \text{src}(e) \]

Logs

\[ e_1 \cdot e_2 \ldots : t_1 \cdot t_2 \ldots \]
Ingredients (I): events & logs

**Events**

\[ \vdash e : t \]

\[ \text{src}(e) \]

**Logs**

\[ \vdash e_1 \cdot e_2 \ldots : t_1 \cdot t_2 \ldots \]

order induced by \( \ell = e_1 \cdots e_n \) \( e_i <_\ell e_j \iff i < j \)
Ingredients (II): log shipping

Machine *Alice* emits logs upon *execution* of commands (we’ll see how in a moment)
Ingredients (II): log shipping

Machine Alice emits logs upon execution of commands (we’ll see how in a moment). Such events are appended to the logs of machines in two phases:
Ingredients (II): log shipping

Machine Alice emits logs upon execution of commands (we’ll see how in a moment). Such events are appended to the logs of machines in two phases:

**Phase I:** emitted events are appended to the local log of the emitting machine

```
a \cdot b \cdot c
```

Alice

```
e_1 \quad e_2 \quad e_3
```
Ingredients (II): log shipping

Machine Alice emits logs upon execution of commands (we’ll see how in a moment). Such events are appended to the logs of machines in two phases:

Phase I: emitted events are appended to the local log of the emitting machine.

\[ a \cdot b \cdot c \]

\begin{array}{cccccc}
  & e_1 & e_2 & e_3 & a & b & c \\
\end{array}

Bob \xrightarrow{- - - - - - - - -} Alice
Ingredients (II): log shipping

Machine *Alice* emits logs upon *execution* of commands (we’ll see how in a moment) Such events are appended to the logs of machines in *two phases*:

**Phase I:** emitted events are appended to the local log of the emitting machine

\[ a \cdot b \cdot c \]

\[ \begin{array}{c}
\text{Alice} \\
\hline
\text{e}_1 \quad \text{e}_2 \quad \text{e}_3 \quad a \quad b \quad c
\end{array} \]

**Phase II:** newly emitted events are shipped to other machines

\[ \begin{array}{c}
\text{Alice} \\
\hline
\text{e}_1 \quad \text{e}_2 \quad \text{e}_3 \quad a \quad b \quad c
\end{array} \]

\[ \begin{array}{c}
\text{Bob} \\
\hline
\text{e}_3
\end{array} \]
Ingredients (II): log shipping

Machine Alice emits logs upon execution of commands (we’ll see how in a moment). Such events are appended to the logs of machines in two phases:

Phase I: emitted events are appended to the local log of the emitting machine

Phase II: newly emitted events are shipped to other machines
Fix a set of \texttt{commands} ranged over by \texttt{c}

Let $\kappa$ range over finite maps from commands to non-empty log types
Fix a set of **commands** ranged over by $c$

Let $\kappa$ range over finite maps from commands to non-empty log types

A *machine* is a **regular term** of this co-inductive grammar

$$M \co ::= \kappa \cdot [t_1 ? M_1 & \cdots & t_n ? M_n]$$

for $i \in \{1, \ldots, n\}$, the **guard** of the $i$-th branch is $t_i$
Passenger $P$ launches an auction for a taxi $T$

\[
\text{InitialP} \quad = \quad \text{Request} \mapsto \text{Requested} \cdot [\text{Requested} \cdot \text{AuctionP}]
\]

\[
\text{AuctionP} \quad = \quad \text{Select} \mapsto \text{Selected} \cdot \text{PassengerId} \cdot [\text{Bid} \cdot \text{BidderId} \cdot \text{AuctionP} \\
& \& \\
& \& \text{Selected} \cdot \text{PassengerId} \cdot \text{RideP}
\]

\[
\text{RideP} \quad = \quad \ldots
\]
An example

Passenger $P$ launches an auction for a taxi $T$

$$\text{Initial}P = \text{Request} \mapsto \text{Requested} \cdot [\text{Requested} ? \text{Auction}P]$$

$$\text{Auction}P = \text{Select} \mapsto \text{Selected} \cdot \text{PassengerId} \cdot [$$

$$\text{Bid} ? \text{BidderId} ? \text{Auction}P \&$$

$$\text{Selected} ? \text{PassengerId} ? \text{Ride}P$$

$$]$$

$$\text{Ride}P = \ldots$$

Notation

- write $t_1 ? M_1 & \cdots & t_n ? M_n$ when $\kappa$ is the empty function
- if $n = 0$, $\kappa \cdot 0$ abbreviates $\kappa \cdot [t_1 ? M_1 & \cdots & t_n ? M_n]$
- write $\& 1 \leq i \leq n \, 1_i ? M_i$ in place of $t_1 ? M_1 & \cdots & t_n ? M_n$

Treat $\kappa$ as its graph and e.g. write $c / 1 \in \kappa$ for $\kappa(c) = 1$ or write $\kappa$ as

$$\{c_1 / l_1, \ldots, c_h / l_h\}$$

when $\kappa : c_i \mapsto l_i$ for $i \in \{1, \ldots, h\}$
Machines as automata

A machine \( M = \kappa \cdot [t_1?M_1 \& \cdots \& t_n?M_n] \) is an FSA where:

- \( \kappa \) yields command-enabling transitions
- A branch \( t_i?M_i \) yields a transition \( M \xrightarrow{t_i?} M_i \) when an event of type \( t_i \) is consumed
Machines as automata

A machine $M = \kappa \cdot [t_1?M_1 \& \cdots \& t_n?M_n]$ is an FSA where:

- $\kappa$ yields command-enabling transitions
- a branch $t_i?M_i$ yields a transition $M \xrightarrow{t_i?} M_i$ when an event of type $t_i$ is consumed

From machines to FSAs

- the states of the automaton are the subtrees of $M$
- the initial state is $M$ and
  - there is a self-loop transition to $M$ labelled $c/l$ for each $c/l \in \kappa$
  - there is a transition labelled $t_i?$ to state $M_i$ for each $i \in \{1, \ldots, n\}$
  - and likewise for $M_i$
Machines as automata

A machine $M = \kappa \cdot [t_1? M_1 \& \cdots \& t_n? M_n]$ is an FSA where:

- $\kappa$ yields command-enabling transitions
- a branch $t_i? M_i$ yields a transition $M \xrightarrow{t_i?} M_i$ when an event of type $t_i$ is consumed

From machines to FSAs

- the states of the automaton are the subtrees of $M$
- the initial state is $M$ and
  - there is a self-loop transition to $M$ labelled $c / l$ for each $c / l \in \kappa$
  - there is a transition labelled $t_i?$ to state $M_i$ for each $i \in \{1 \ldots, n\}$
  - and likewise for $M_i$

This construction yields a finite-state automaton by the regularity of $M$
An example

Let's build the FSA of the machine InitialP on slide 18.

\[
\text{InitialP} =
\]

\[
\rightarrow \text{InitialP}
\]

\[
\text{InitialP} =
\]

\[
\rightarrow \text{InitialP}
\]
An example

Let’s build the FSA of the machine InitialP on slide 18.

InitialP = Request \rightarrow Requested.
An example

Let’s build the FSA of the machine InitialP on slide 18.

\[
\text{InitialP} = \text{Request} \rightarrow \text{Requested} \cdot [\text{Requested}?] \text{AuctionP}
\]
An example

Let’s build the FSA of the machine InitialP on slide 18.

\[
\text{InitialP} = \text{Request} \rightarrow \text{Requested} \cdot [\text{Requested}?] \text{AuctionP}
\]

\[
\text{AuctionP} = \quad
\]
Let’s build the FSA of the machine InitialP on slide 18.

\[
\begin{align*}
\text{InitialP} & = \text{Request} \rightarrow \text{Requested} \cdot [\text{Requested? AuctionP}] \\
\text{AuctionP} & = \text{Select} \leftrightarrow \text{Selected} \cdot \text{PassengerId} 
\end{align*}
\]
An example

Let’s build the FSA of the machine \texttt{InitialP} on slide 18.

\[
\text{InitialP} \quad = \quad \text{Request} \xrightarrow{} \text{Requested} \cdot [\text{Requested}?] \text{AuctionP}
\]

\[
\text{AuctionP} \quad = \quad \text{Select} \leftrightarrow \text{Selected} \cdot \text{PassengerId} \cdot [\text{Bid}?] \text{BidderId}?] \text{AuctionP}
\]
Let’s build the FSA of the machine \textbf{InitialP} on slide 18.

\begin{align*}
\text{InitialP} & \quad = \quad \text{Request} \longrightarrow \text{Requested} \cdot \left[ \text{Requested} \cdot \text{AuctionP} \right] \\
\text{AuctionP} & \quad = \quad \text{Select} \longrightarrow \text{Selected} \cdot \text{PassengerId} \cdot \left[ \text{Bid} \cdot \text{BidderId} \cdot \text{AuctionP} \right]
\end{align*}
An example

Let’s build the FSA of the machine InitialP on slide 18.

\[
\begin{align*}
\text{InitialP} & \quad = \quad \text{Request} \rightarrow \text{Requested} \cdot [\text{Requested? AuctionP}] \\
\text{AuctionP} & \quad = \quad \text{Select} \leftrightarrow \text{Selected} \cdot \text{PassengerId} \cdot [ \\
& \quad \quad \text{Bid? BidderId? AuctionP} \ \& \\
& \quad \quad \text{Selected? PassengerId? RideP} \\
\text{RideP} & \quad = \quad \ldots
\end{align*}
\]
Machines’ semantics

So, think of $M = \kappa \cdot [t_1 ? M_1 \& \cdots \& t_n ? M_n]$ as an FSA where transitions are

- either self-loops (determined by the $\kappa$ part)
- or event consuptions (determined by the guards of the branches $t_i$)
Machines’ semantics

So, think of $M = \kappa \cdot [t_1?M_1 \& \cdots \& t_n?M_n]$ as an FSA where transitions are

- either self-loops (determined by the $\kappa$ part)
- or event consumptions (determined by the guards of the branches $t_i$)

We restrict to deterministic machines and treat them as emitters/consumers of events with a semantics given in terms of state transition function:

$$
\delta(M, e) = M
$$

$$
\delta(M, e \cdot \ell) = \begin{cases} 
\delta(M', \ell) & \text{if } \vdash e : t, \ M \xrightarrow{t?} M' \\
\delta(M, \ell) & \text{otherwise}
\end{cases}
$$
Machines’ semantics

So, think of $M = \kappa \cdot [t_1 \leftarrow M_1 \& \cdots \& t_n \leftarrow M_n]$ as an FSA where transitions are

- either self-loops (determined by the $\kappa$ part)
- or event consumptions (determined by the guards of the branches $t_i$)

We restrict to deterministic machines and treat them as emitters/consumers of events with a semantics given in terms of state transition function:

$$
\delta(M, e) = M
$$

$$
\delta(M, e \cdot \ell) = \begin{cases} 
\delta(M', \ell) & \text{if } \vdash e : t, \ M \xrightarrow{t?} M' \\
\delta(M, \ell) & \text{otherwise}
\end{cases}
$$

That is

$M$ with local log $\ell$ is in the implicit state $\delta(M, \ell)$ reached after processing each event in $\ell$. 

Machines’ semantics

So, think of $M = \kappa \cdot [t_1 ? M_1 \& \cdots \& t_n ? M_n]$ as an FSA where transitions are
- either self-loops (determined by the $\kappa$ part)
- or event consuptions (determined by the guards of the branches $t_i$)

We restrict to deterministic machines and treat them as emitters/consumers of events with a semantics given in terms of state transition function:

$\delta(M, e) = M$

$\delta(M, e \cdot \ell) = \begin{cases} 
  \delta(M', \ell) & \text{if } \vdash e : t, M \xrightarrow{t?} M' \\
  \delta(M, \ell) & \text{otherwise}
\end{cases}$

That is

$M$ with local log $\ell$ is in the implicit state $\delta(M, \ell)$ reached after processing each event in $\ell$

$\delta(M, \ell) \xrightarrow{c/1} \delta(M, \ell)$

$\ell'$ fresh

$\vdash \ell' : 1$

$\delta(M, \ell) \xrightarrow{c/1} \delta(M, \ell \cdot \ell')$

$\delta(M, \ell) \xrightarrow{\ell' \cdot \ell}$
Machines’ semantics

So, think of $M = \kappa \cdot [t_1 ? M_1 \& \cdots \& t_n ? M_n]$ as an FSA where transitions are
- either self-loops (determined by the $\kappa$ part)
- or event consuptions (determined by the guards of the branches $t_i$)

We restrict to deterministic machines and treat them as emitters/consumers of events with a semantics given in terms of state transition function:

$$
\delta(M, e) = M
$$

$$
\delta(M, e \cdot \ell) = \begin{cases} 
\delta(M', \ell) & \text{if } \vdash e : t, \ M \xrightarrow{t?} M' \\
\delta(M, \ell) & \text{otherwise}
\end{cases}
$$

$$
\delta(M, \ell) \xrightarrow{c/1} \delta(M, \ell) \quad \ell' \text{ fresh} \quad \vdash \ell' : 1
$$

That is
$M$ with local log $\ell$ is in the implicit state $\delta(M, \ell)$ reached after processing each event in $\ell$

That is
after processing the events in $\ell$, $M$ reaches a state enabling $c/1$ then the command execution can emit $\ell'$ of type 1 and append it to the local log of $M$
An example

Take the machine $\text{InitialP}$ (slide 20) with a local log $\ell = \text{ignoreMe} \cdot \text{ignoreMeToo}$ where $\not\vdash \text{ignoreMe} : \text{Requested}$ and $\not\vdash \text{ignoreMeToo} : \text{Requested}$

By definition of $\delta$

- $\delta(\text{InitialP}, \ell) = \text{InitialP}$
An example

Take the machine \( \text{InitialP} \) (slide 20) with a local log \( \ell = \text{ignoreMe} \cdot \text{ignoreMeToo} \) where \( \not\vdash \text{ignoreMe} : \text{Requested} \) and \( \not\vdash \text{ignoreMeToo} : \text{Requested} \)

By definition of \( \delta \)

\[
\delta(\text{InitialP}, \ell) = \text{InitialP} \quad \text{hence}
\]

\[
\delta(\text{InitialP}, \ell) \xrightarrow{\text{Request} / \text{Requested}} \delta(\text{InitialP}, \ell)
\]
An example

Take the machine \texttt{InitialP} (slide 20) with a local log $\ell = \textit{ignoreMe} \cdot \textit{ignoreMeToo}$ where $\not\vdash \textit{ignoreMe} : \text{Requested}$ and $\not\vdash \textit{ignoreMeToo} : \text{Requested}$

By definition of $\delta$

- $\delta(\texttt{InitialP}, \ell) = \texttt{InitialP}$ hence
- $\delta(\texttt{InitialP}, \ell) \xrightarrow{\text{Request} / \text{Requested}} \delta(\texttt{InitialP}, \ell)$
- $(\texttt{InitialP}, \ell) \xrightarrow{\text{Request} / \text{Requested}} (\texttt{InitialP}, \ell \cdot \text{Requested})$ hence with $\vdash \text{Requested} : \text{Request}$ and $\text{src}(\text{Requested}) = \text{P}$ is possible
An example

Take the machine $\text{InitialP}$ (slide 20) with a local log $\ell = \text{ignoreMe} \cdot \text{ignoreMeToo}$ where $\not\vdash \text{ignoreMe} : \text{Requested}$ and $\not\vdash \text{ignoreMeToo} : \text{Requested}$

![Diagram of the machine InitialP with transitions Request / Requested, Requested?, Requested?, and Selected?.

By definition of $\delta$

- $\delta(\text{InitialP}, \ell) = \text{InitialP}$ hence
- $\delta(\text{InitialP}, \ell) \xrightarrow{\text{Request} / \text{Requested}} \delta(\text{InitialP}, \ell)$
- $(\text{InitialP}, \ell) \xrightarrow{\text{Request} / \text{Requested}} (\text{InitialP}, \ell \cdot \text{Requested})$ hence with $\vdash \text{Requested} : \text{Request}$ and $\text{src}(\text{Requested}) = \text{P}$ is possible

Exercise

Calculate $\delta(\text{InitialP}, \ell \cdot \text{Requested})$
Some considerations

The commands are enabled only from the state reached after processing all the events in the local log of the machine.
Some considerations

The commands are enabled only from the state reached after processing all the events in the local log of the machine.

Deterministic machines may have non-deterministic behaviour! Recall: commands are triggered by the environment.
Some considerations

The commands are enabled only from the state reached after processing all the events in the local log of the machine.

Deterministic machines may have non-deterministic behaviour!
Recall: commands are triggered by the environment

We have formalised the emission of events and their consumption
We now focus on the formalisation of log shipping
A swarm (of size $n$) is a pair $(S, \ell)$ where

- $S$ maps each index $1 \leq i \leq n$ to a pair $(M_i, \ell_i)$
- $\ell$ is the (global) log
A swarm (of size $n$) is a pair $(S, \ell)$ where

- $S$ maps each index $1 \leq i \leq n$ to a pair $(M_i, \ell_i)$
- $\ell$ is the (global) log

Notation

\[
\begin{array}{c|c|c}
M_1 & \ell_1 & \ldots \\
\vdots & \vdots & \vdots \\
M_n & \ell_n & \ell
\end{array}
\]
A **swarm (of size \( n \))** is a pair \((S, \ell)\) where
- \( S \) maps each index \( 1 \leq i \leq n \) to a pair \((M_i, \ell_i)\)
- \( \ell \) is the (global) log

### Disclaimer

Seemingly, we’ve a contradiction: isn’t the global log a centralisation point?

Well...no, it isn’t: the global log is just a theoretical ploy!
- it abstracts away from low-level technical details for events’ dispatching

Log shipping middlewares rely on timestamp mechanisms (Actyx uses Lamport’s timestamps) and guarantee that events are in the same order in all the local logs
Swarms

A swarm (of size $n$) is a pair $(S, \ell)$ where

- $S$ maps each index $1 \leq i \leq n$ to a pair $(M_i, \ell_i)$
- $\ell$ is the (global) log

Disclaimer

Seemingly, we’ve a contradiction: isn’t the global log a centralisation point?

Well...no, it isn’t: the global log is just a theoretical ploy!

- it abstracts away from low-level technical details for events’ dispatching
- it elegantly (IOHO) models asynchrony
Swarms

A **swarm (of size** $n$) **is a pair** $(S, \ell)$ where
- $S$ maps each index $1 \leq i \leq n$ to a pair $(M_i, \ell_i)$
- $\ell$ is the (global) log

**Disclaimer**

Seemingly, we’ve a contradiction: isn’t the global log a centralisation point?

Well...no, it isn’t: the global log is just a theoretical ploy!
- it abstracts away from low-level technical details for events’ dispatching
- it elegantly (IOHO) models asynchrony
- it is not used in our algorithms and tools
A swarm $M_1[\ell_1] \ldots | M_n[\ell_n] | \ell$ is coherent if $\ell = \bigcup_{1 \leq i \leq n} \ell_i$ and $\ell_i \sqsubseteq \ell$ for $1 \leq i \leq n$. That is, all events of $\ell_1$ appear in the same order in $\ell_2$.
A swarm $M_1[\ell_1] \ldots M_n[\ell_n]$ is **coherent** if \( \ell = \bigcup_{1 \leq i \leq n} \ell_i \) and $\ell_i \sqsubseteq \ell$ for $1 \leq i \leq n$ where $\ell_1 \sqsubseteq \ell_2$ is the **sublog** relation defined as

- $\ell_1 \subseteq \ell_2$ and $<_{\ell_1} \subseteq <_{\ell_2}$ and
- $e <_{\ell_2} e'$, $\text{src}(e) = \text{src}(e')$ and $e' \in \ell_1 \implies e \in \ell_1$

That is all events of $\ell_1$ appear in the same order in $\ell_2$

That is the per-source partitions of $\ell_1$ are prefixes of the corresponding partitions of $\ell_2$
A swarm $\mathbf{M}_1[\ell_1] \ldots | \mathbf{M}_n[\ell_n] | \ell$ is coherent if $\ell = \bigcup_{1 \leq i \leq n} \ell_i$ and $\ell_i \sqsubseteq \ell$ for $1 \leq i \leq n$

where $\ell_1 \sqsubseteq \ell_2$ is the sublog relation defined as

- $\ell_1 \subseteq \ell_2$ and $\prec \ell_1 \subseteq \prec \ell_2$ and
- $e \prec \ell_2 e'$, $\text{src}(e) = \text{src}(e')$ and $e' \in \ell_1 \Rightarrow e \in \ell_1$

That is all events of $\ell_1$ appear in the same order in $\ell_2$

That is the per-source partitions of $\ell_1$ are prefixes of the corresponding partitions of $\ell_2$

Hereafter, we assume coherence
Exercise
Recall slide 16 and consider a swarm

\[
\begin{array}{c|c|c|c|c|c}
\vdots & \text{Alice} & e_1 & e_2 & e_3 & a & b & c & \vdots & \ell \\
\end{array}
\]

If \( \ell = e_1 \cdot e_2 \cdot e_3 \cdot e \), under which condition is (1) coherent?
Exercise

Recall slide 16 and consider a swarm

\[
\begin{array}{c|c|c|c|c|c}
\cdots & \text{Alice} & e_1 & e_2 & e_3 & a & b & c & \cdots & \ell \\
\end{array}
\]

If $\ell = e_1 \cdot e_2 \cdot e_3 \cdot e$, under which condition is (1) coherent?

The propagation of newly generated events happens by merging logs:

\textbf{Log merging:} $\ell_1 \bowtie \ell_2 = \{ \ell \mid \ell \subseteq \ell_1 \cup \ell_2 \text{ and } \ell_1 \sqsubseteq \ell \text{ and } \ell_2 \sqsubseteq \ell \}$
Semantics of swarms

By rule [Local] below, a command’s execution updates both local and global logs:

$$\text{S}(i) = M_{\ell_i} \quad M_{\ell_i} \xrightarrow{c/1} M_{\ell'_i} \quad \text{src}(\ell'_i \setminus \ell_i) = \{i\} \quad \ell' \in \ell \bowtie \ell'_i$$

$$\left(\text{S}, \ell\right) \xrightarrow{c/1} \left(\text{S}[i \mapsto M_{\ell'_i}], \ell'\right)$$
Semantics of swarms

By rule [Local] below, a command’s execution updates both local and global logs

\[ S(i) = M_{\ell_i} \quad M_{\ell_i} \xrightarrow{c/1} M_{\ell_i}' \quad src(\ell_i' \setminus \ell_i) = \{i\} \quad \ell' \in \ell \bowtie \ell_i' \]

\[ (S, \ell) \xrightarrow{c/1} (S[i \mapsto M_{\ell_i'}], \ell') \]

[Local]

By rule [Prop] above, the propagation of events happens

- by shipping a non-deterministically chosen subset of events in the global log
- to a non-deterministically chosen machine

\[ S(i) = M_{\ell_i} \quad \ell_i \subseteq \ell' \subseteq \ell \quad \ell_i \subset \ell' \]

\[ (S, \ell) \xrightarrow{\tau} (S[i \mapsto M_{\ell_i'}], \ell) \]

[Prop]
Semantics at work (I)

If \( B \vdash b \cdot c \cdot d \cdot e \) with \( \vdash d \cdot e : 1 \)
Semantics at work (I)

If

\[ B b \xrightarrow{c/1} B b \cdot d \cdot e \]

with

\[ \vdash d \cdot e : 1 \]

then, by [Local]

\[ A a \mid B b \mid C c \mid a \cdot b \cdot c \xrightarrow{c/1} A a \mid B b \cdot d \cdot e \mid C c \mid \ell \]
Semantics at work (I)

If

$$\text{B}_b \xrightarrow{c/1} \text{B} \cdot d \cdot e$$

with

$$\vdash d \cdot e : 1$$

then, by [Local]

$$\text{A}_a | \text{B}_b | \text{C}_c | a \cdot b \cdot c \xrightarrow{c/1} \text{A}_a | \text{B} \cdot d \cdot e | \text{C}_c | \ell$$

for all

$$\ell \in (a \cdot b \cdot c) \otimes (b \cdot d \cdot e)$$
Semantics at work (I)

If

\[ B b \xrightarrow{c/1} B b \cdot d \cdot e \]

with \( \vdash d \cdot e : 1 \)

then, by [Local]

\[ A a \mid B b \mid C c \mid a \cdot b \cdot c \xrightarrow{c/1} A a \mid B b \cdot d \cdot e \mid C c \mid \ell \]

for all \( \ell \in (a \cdot b \cdot c) \boxtimes (b \cdot d \cdot e) \)

Exercise

Compute \((a \cdot b \cdot c) \boxtimes (b \cdot d \cdot e)\)
Semantics at work (II)

Take from slide 28

\[
\begin{array}{c}
A \mid B \mid C \\
\mid b \cdot a \cdot c
\end{array} \quad \xrightarrow{c/1} \quad \begin{array}{c}
A \mid B \cdot d \cdot e \\
\mid C \\
\mid b \cdot a \cdot d \cdot e \cdot c
\end{array}
\]

and let's propagate some events
Semantics at work (II)

Take from slide 28

\[ \frac{Aa \mid Bb \mid Cc \mid b \cdot a \cdot c}{c / 1} \rightarrow Aa \mid Bb \cdot d \cdot e \mid Cc \mid b \cdot a \cdot d \cdot e \cdot c = \ell \]

and let's propagate some events

**Exercise**

Can we propagate just event \( e \)?
Semantics at work (II)

Take from slide 28

\[
\begin{array}{c|c|c|c}
A & B & C \\
\hline
a & b & c \\
\end{array}
\quad \xrightarrow{c/1} \quad
\begin{array}{c|c|c|c}
A & B & C \\
\hline
\phantom{a} & \phantom{b} & \phantom{c} \\
\end{array}
\]

and let’s propagate some events

**Exercise**

Can we propagate just event \( e \)?

By rule [Prop] we can propagate a non-deterministically chosen sublog of \( b \cdot d \cdot e \)
Semantics at work (II)

Take from slide 28

\[
\begin{array}{c|c|c|c}
A & B & C & b \cdot a \cdot c \\
\hline
a & b & c & c / l \\
\end{array}
\begin{array}{c}
\rightarrow
\end{array}
\begin{array}{c|c|c|c|c}
A & B & C & b \cdot a \cdot d \cdot e \cdot c \\
\hline
a & b & d & e \\
\end{array}
\]

and let's propagate some events

**Exercise**

Can we propagate just event \( e \)?

By rule [Prop] we can propagate a non-deterministically chosen sublog of \( b \cdot d \cdot e \)

Let's propagate \( d \cdot e \)

\[
\begin{array}{c|c|c|c|c|c}
A & B & C & \ell \\
\hline
a & b \cdot d & e & c & l \\
\end{array}
\begin{array}{c}
\tau
\end{array}
\begin{array}{c|c|c|c|c|c}
A & B & C & \ell \\
\hline
a & b \cdot d & e \cdot c & b \cdot d & e & c \\
\end{array}
\]

In both cases \( b \) must be shipped too. Why?

And why is event \( a \) not shipped to \( C \) together with the events from \( B \)?
Semantics at work (II)

Take from slide 28

\[ A \begin{array}{c} a \end{array} | B \begin{array}{c} b \end{array} | C \begin{array}{c} c \end{array} \rightarrow^{c/\ell} A \begin{array}{c} a \end{array} | B \begin{array}{c} b \cdot d \cdot e \end{array} | C \begin{array}{c} c \end{array} = \ell \]

and let's propagate some events

**Exercise**

Can we propagate just event \( e \)?

By rule [Prop] we can propagate a non-deterministically chosen sublog of \( b \cdot d \cdot e \)

Let's propagate \( d \cdot e \)

**Exercise**

In both cases \( b \) must be shipped too. Why?
And why is event \( a \) not shipped to \( C \) together with the events from \( B \)?
Plan of the talk

A motivating case study

Our formalisation

Our typing discipline

Tool support

Open issues
– Behavioural types for swarms –
A taxi service

An intuitive auction protocol for a passenger $P$ to get a taxi $T$:

We assume one passenger and one office (for simplicity) but an arbitrary number of taxis. A receipt is issued by the office $O$ at the end of the ride (if any).
An intuitive auction protocol for a passenger $P$ to get a taxi $T$:

We assume

- one passenger and one office (for simplicity)
A taxi service

An intuitive auction protocol for a passenger $P$ to get a taxi $T$:

We assume

- one passenger and one office (for simplicity)
- but an arbitrary number of taxis
An intuitive auction protocol for a passenger $P$ to get a taxi $T$:

We assume
- one passenger and one office (for simplicity)
- but an arbitrary number of taxis
- a receipt is issued by the office $O$ at the end of the ride (if any)
Quoting W3C:

“[...] a contract [...] of the common ordering conditions and constraints under which messages are exchanged [...] from a global viewpoint [...] Each party can then use the global definition to build and test solutions [...] global specification is in turn realised by combination of the resulting local systems”
Quoting W3C:

“[..] a contract [..] of the common ordering conditions and constraints under which messages are exchanged [..] from a global viewpoint [..]
Each party can then use the global definition to build and test solutions [..]
global specification is in turn realised by combination of the resulting local systems”
Choreographies

Quoting W3C:

“[...] a contract [...] of the common ordering conditions and constraints under which messages are exchanged [...] from a global viewpoint [...] Each party can then use the global definition to build and test solutions [...] global specification is in turn realised by combination of the resulting local systems”

Synchrony

Choreography G

global viewpoint

Asynchrony

\[ M_1 \quad M_i \quad M_n \]

Local viewpoint \(_1\) Local viewpoint \(_i\) Local viewpoint \(_n\)

spec,no code
Choreographies

Quoting W3C:

“[…] a contract […] of the common ordering conditions and constraints under which messages are exchanged […] from a global viewpoint […] Each party can then use the global definition to build and test solutions […] global specification is in turn realised by combination of the resulting local systems”

Synchrony

Asynchrony

Choreography G

global viewpoint

Well-formedness

spec,no code

M_1
Local viewpoint_1

M_i
Local viewpoint_i

M_n
Local viewpoint_n
Quoting W3C:

“[...] a contract [...] of the common ordering conditions and constraints under which messages are exchanged [...] from a global viewpoint [...] Each party can then use the global definition to build and test solutions [...] global specification is in turn realised by combination of the resulting local systems”
Choreographies

Quoting W3C:

“[…] a contract […] of the common ordering conditions and constraints under which messages are exchanged […] from a global viewpoint […] Each party can then use the global definition to build and test solutions […] global specification is in turn realised by combination of the resulting local systems”
Swarm protocols: global type for local-first applications

An **idealised** specification relying on **synchronous communication**

Fix a set of **roles** ranged over by $\mathbf{R}$ (e.g., $\mathbf{P}$, $\mathbf{T}$, and $\mathbf{O}$ on slide 32)

The syntax of **swarm protocols** is again given co-inductively:

$$G \co ::= \sum_{i \in I} c_i @ R_i \langle l_i \rangle \cdot G_i \mid 0 \quad \text{where } I \text{ is a finite set (of indexes)}$$
An example

A swarm protocol for the taxi scenario on slide 32:

\[ G = \text{Request}@P\langle\text{Requested}\rangle \cdot G_{\text{auction}} \]

\[ G_{\text{auction}} = \text{Offer}@T\langle\text{Bid} \cdot \text{BidderID}\rangle \cdot G_{\text{auction}} \]
\[ + \text{Select}@P\langle\text{Selected} \cdot \text{PassengerID}\rangle \cdot G_{\text{choose}} \]

\[ G_{\text{choose}} = \text{Arrive}@T\langle\text{Arrived}\rangle \cdot \text{Start}@P\langle\text{Started}\rangle \cdot G_{\text{ride}} \]
\[ + \text{Cancel}@P\langle\text{Cancelled}\rangle \cdot \text{Receipt}@O\langle\text{Receipt}\rangle \cdot 0 \]

\[ G_{\text{ride}} = \text{Record}@T\langle\text{Path}\rangle \cdot G_{\text{ride}} \]
\[ + \text{Finish}@P\langle\text{Finished} \cdot \text{Rating}\rangle \cdot \text{Receipt}@O\langle\text{Receipt}\rangle \cdot 0 \]
An example

A swarm protocol for the taxi scenario on slide 32:

\[ G = \text{Request}@P\langle \text{Requested} \rangle \cdot G_{\text{auction}} \]

\[ G_{\text{auction}} = \text{Offer}@T\langle \text{Bid} \cdot \text{BidderID} \rangle \cdot G_{\text{auction}} \\
+ \text{Select}@P\langle \text{Selected} \cdot \text{PassengerID} \rangle \cdot G_{\text{choose}} \]

\[ G_{\text{choose}} = \text{Arrive}@T\langle \text{Arrived} \rangle \cdot \text{Start}@P\langle \text{Started} \rangle \cdot G_{\text{ride}} \\
+ \text{Cancel}@P\langle \text{Cancelled} \rangle \cdot \text{Receipt}@O\langle \text{Receipt} \rangle \cdot 0 \]

\[ G_{\text{ride}} = \text{Record}@T\langle \text{Path} \rangle \cdot G_{\text{ride}} \\
+ \text{Finish}@P\langle \text{Finished} \cdot \text{Rating} \rangle \cdot \text{Receipt}@O\langle \text{Receipt} \rangle \cdot 0 \]
Swarm protocols as FSA

Like for machines, a swarm protocols $G = \sum_{i \in I} c_i \otimes R_i \langle 1_i \rangle$. $G_i$ has an associated FSA:

- the set of states consists of $G$ plus the states in $G_i$ for each $i \in \{1 \ldots, n\}$
- $G$ is the initial state
- for each $i \in I$, $G$ has a transition to state $G_i$ labelled with $c_i \otimes R_i \langle 1_i \rangle$, written $G \xrightarrow{c_i / 1_i} G_i$
An example

Request P ⟨Requested⟩

Offer T ⟨Bid · BidderID⟩

Select P ⟨Selected · PassengerID⟩

Arrive T ⟨Arrived⟩

Start T ⟨Started⟩

Record T ⟨Path⟩

Cancel P ⟨Cancelled⟩

Finish P ⟨Finished · Rating⟩

Receipt O ⟨Receipt⟩

There is a race in state 3!

the selected taxi may invoke Arrive while P loses patience and invokes Cancel

This protocol violates well-formedness conditions typically imposed on behaviour types due to the race in state 3 (because it has two selectors, which is also true of states 2 and 5)

Removing log types yields exactly the FSA of the swarm protocol on slide 32
An example

Removing log types yields exactly the FSA of the swarm protocol on slide 32

There is a race in state 3! The selected taxi may invoke Arrive while P loses patience and invokes Cancel.

This protocol violates well-formedness conditions typically imposed on behavioural types due to the race in state 3 (because it has two selectors, which is also true of states 2 and 5).

Removing log types yields exactly the FSA of the swarm protocol on slide 32.
An example

There is a race in state 3!

- the selected taxi may invoke Arrive
- while \( P \) loses patience and invokes Cancel

Removing log types yields exactly the FSA of the swarm protocol on slide 32
An example

There is a race in state 3!

- the selected taxi may invoke **Arrive**
- **while** P loses patience and invokes **Cancel**

Removing log types yields exactly the FSA of the swarm protocol on slide 32

This protocol violates well-formedness conditions typically imposed on behavioural types due to the race in state 3 (because it has two selectors, which is also true of states 2 and 5).
Semantics of swarm protocols

**One** rule only!

\[ (G, \ell) \xrightarrow{c/1} (G, \ell) \]
Semantics of swarm protocols

**One rule only!**

\[ \delta(G, \ell) \xrightarrow{c/1} G' \]

\[ (G, \ell) \xrightarrow{c/1} (G, \ell') \]

where

\[ \delta(G, \ell) = \begin{cases} 
G & \text{if } \ell = \epsilon \\
\delta(G', \ell'') & \text{if } G \xrightarrow{c/1} G' \text{ and } \vdash \ell' : 1 \text{ and } \ell = \ell' \cdot \ell'' \\
\bot & \text{otherwise}
\end{cases} \]

Logs to be consumed "atomically", hence \( \delta(G, \ell) \) may be undefined.
Semantics of swarm protocols

One rule only!

\[ \delta(G, \ell) \xrightarrow{c/1} G' \quad \vdash \ell' : 1 \quad \ell' \text{ log of fresh events} \]

\[ (G, \ell) \xrightarrow{c/1} (G, \ell \cdot \ell') \] [G-Cmd]

where

\[ \delta(G, \ell) = \begin{cases} 
G & \text{if } \ell = \epsilon \\
\delta(G', \ell'') & \text{if } G \xrightarrow{c/1} G' \text{ and } \vdash \ell' : 1 \text{ and } \ell = \ell' \cdot \ell'' \\
\bot & \text{otherwise}
\end{cases} \]

Logs to be consumed “atomically”, hence \( \delta(G, \ell) \) may be undefined.
Semantics of swarm protocols

One rule only!

\[ \delta(G, \ell) \xrightarrow{c/1} G' \quad \vdash \ell' : 1 \quad \ell' \quad \text{log of fresh events} \]

\[ (G, \ell) \xrightarrow{c/1} (G, \ell \cdot \ell') \quad \text{[G-Cmd]} \]

where

\[ \delta(G, \ell) = \begin{cases} 
G & \text{if } \ell = \epsilon \\
\delta(G', \ell'') & \text{if } G \xrightarrow{c/1} G' \text{ and } \vdash \ell' : 1 \text{ and } \ell = \ell' \cdot \ell'' \\
\bot & \text{otherwise}
\end{cases} \]

Logs to be consumed “atomically”, hence \( \delta(G, \ell) \) may be undefined

We restrict ourselves to deterministic swarm protocols that is, on different transitions from a same state

- log types start differently
- pairs (command,role) differ

log determinism
command determinism
Transitions of a swarm protocol $G$ are labelled with a role that may invoke the command
From swarm protocols to machines

Transitions of a swarm protocol $G$ are labelled with a role that may invoke the command

Each machine plays one role
Transitions of a swarm protocol $G$ are labelled with a role that may invoke the command

Each machine plays one role

Obtain machines by projecting $G$ on each role
Transitions of a swarm protocol $G$ are labelled with a role that may invoke the command

Each machine plays one role

Obtain machines by projecting $G$ on each role

First attempt

$$\left( \sum_{i \in I} c_i @ R_i \langle l_i \rangle \cdot G_i \right) \downarrow_R = \kappa \cdot [\&_{i \in I} l_i? G_i \downarrow_R]$$

where $\kappa = \{ (c_i / l_i) | R_i = R \text{ and } i \in I \}$
From swarm protocols to machines

Transitions of a swarm protocol $G$ are labelled with a role that may invoke the command

Each machine plays one role

Obtain machines by projecting $G$ on each role

First attempt

$$\left( \sum_{i \in I} c_i \otimes R_i \langle l_i \rangle \cdot G_i \right) \downarrow_R = \kappa \cdot [\&_{i \in I} 1_i ? G_i \downarrow_R]$$

where $\kappa = \{(c_i / l_i) \mid R_i = R \text{ and } i \in I\}$

simple, but

- projected machines are large in all but the most trivial cases
- processing all events is undesirable: security and efficiency
Another attempt

Let’s subscribe to subscriptions: maps from roles to sets of event types

In pub-sub, processes subscribe to “topics”
Another attempt

Let’s subscribe to subscriptions: maps from roles to sets of event types

Given $G = \sum_{i \in I} c_i \circ R_i \langle l_i \rangle \cdot G_i$, the projection of $G$ on a role $R$ with respect to subscription $\sigma$ is

$$G \downarrow^\sigma_R = \kappa \cdot [\&_{j \in J} \text{filter}(l_j, \sigma(R)) \ ? \ G_j \downarrow^\sigma_R]$$

where
Another attempt

Let’s subscribe to subscriptions: maps from roles to sets of event types

Given \( G = \sum_{i \in I} c_i @ R_i \cdot \langle l_i \rangle \cdot G_i \), the projection of \( G \) on a role \( R \) with respect to subscription \( \sigma \) is

\[
G \downarrow^\sigma_R = \kappa \cdot \left[ \land_{j \in J} \text{filter}(l_j, \sigma(R)) \right] G_j \downarrow^\sigma_R
\]

where

\[
\kappa = \left\{ c_i / l_i \mid R_i = R \text{ and } i \in I \right\}
\]

\[
J = \left\{ i \in I \mid \text{filter}(l_i, \sigma(R)) \neq \epsilon \right\}
\]

\[
\text{filter}(l, E) = \begin{cases} 
\epsilon, & \text{if } t = \epsilon \\
\epsilon, & \text{if } t \in E \text{ and } l = t \cdot l' \\
\text{filter}(l, E), & \text{otherwise}
\end{cases}
\]
An example

A reasonable subscription for $P$ is the total one since the passenger should be aware of all events: $\sigma(P)$ contains all event types.

Exercise

The taxi driver does not need to bother with the receipt: the subscription for $\sigma(T)$ consists of all messages but Receipt; give the projection of the taxi protocol on such subscription for $T$. 

Exercise (hard)

Is this a good idea?
Exercise

The taxi driver does not need to bother with the receipt: the subscription for $\sigma(T)$ consists of all messages but Receipt; give the projection of the taxi protocol on such subscription for $T$. 
Exercise

The taxi driver does not need to bother with the receipt: the subscription for $\sigma(T)$ consists of all messages but Receipt; give the projection of the taxi protocol on such subscription for $T$.

If we want the office to know only the details about the ride we set

$$\sigma(0) = \{\text{Started, Finished, Receipt}\}$$
An example

**Exercise**

The taxi driver does not need to bother with the receipt: the subscription for $\sigma(T)$ consists of all messages but Receipt; give the projection of the taxi protocol on such subscription for $T$.

If we want the office to know only the details about the ride we set $\sigma(0) = \{\text{Started}, \text{Finished}, \text{Receipt}\}$

**Exercise (hard)**

Is this a good idea?
Well-formedness: sufficient conditions for well-behaviour

Transitory deviations are tolerated provided that consistency is eventually recovered.
Transitory deviations are tolerated provided that consistency is eventually recovered.

Example

T may bid after P has made their selection if the selection event T has not yet been received.

This inconsistency is temporary: when the selection event reaches T this inconsistency is recognised and resolved.
Well-formedness: sufficient conditions for well-behaviour

Transitory deviations are tolerated provided that consistency is eventually recovered

**Example**

T may bid after P has made their selection if the selection event T has not yet been received.

This inconsistency is temporary: when the selection event reaches T this inconsistency is recognised and resolved

**Convention**

Let’s write \( R \in_\sigma G = \sum_{i \in I} c_i \oplus R_i(1_i \cdot G_i) \) when there is \( i \in I \) such that

\[
R = R_i \quad \text{or} \quad \sigma(R) \cap 1_i \neq \emptyset \quad \text{or} \quad R \in_\sigma G_i
\]

and set \( \text{roles}(G, \sigma) = \{ R \mid R \in_\sigma G \} \) and
Well-formedness

Trading consistency for availability has implications:
Well-formedness = Causality

Trading consistency for availability has implications:
  Propagation of events is non-atomic (cf. rule [Prop])
  \[ \implies \] differences in how machines perceive the (state of the) computation

Causality

Fix a subscription \( \sigma \). For each branch \( i \in I \) of \( G = \sum_{i \in I} c_i @ R_i \langle l_i \rangle \cdot G_i \)

Explicit re-enabling \( \sigma(R_i) \cap l_i \neq \emptyset \)

Command causality
  if \( R \) executes a command in \( G_i \)
  then \( \sigma(R) \cap l_i \neq \emptyset \) and \( \sigma(R) \cap l_i \supseteq \bigcup_{R' \in \sigma G_i} \sigma(R') \cap l_i \)

If \( R \) should have \( c \) enabled after \( c' \) then \( \sigma(R) \) contains some event type emitted by \( c' \)
Well-formedness = Causality + Determinacy

Trading consistency for availability has implications:
  Propagation of events is non-atomic (cf. rule \([\text{Prop}]\))
  \[\implies\] different roles may take inconsistent decisions

Causality & Determinacy

Fix a subscription \(\sigma\). For each branch \(i \in I\) of 
\(G = \sum_{i \in I} c_i \circ R_i(\ell_i) \cdot G_i\)

Explicit re-enabling
\[\sigma(R_i) \cap \ell_i \neq \emptyset\]

Command causality
\[
\text{if } R \text{ executes a command in } G_i \text{ then } \\
\sigma(R) \cap \ell_i \neq \emptyset \quad \text{and} \quad \sigma(R) \cap \ell_i \supseteq \bigcup_{R' \in \sigma G_i} \sigma(R') \cap \ell_i
\]

Determinacy
\[R \in_\sigma G_i \implies \ell_i[0] \in \sigma(R)\]
Well-formedness $= \text{Causality} + \text{Determinacy} - \text{Confusion}$

Trading consistency for availability has implications:
 propagation of events is non-atomic (cf. rule [Prop])
$\implies$ branches unambiguously identified and events emitted on eventually discharged branches ignored

**Causality & Determinacy & Confusion freeness**

Fix a subscription $\sigma$. For each branch $i \in I$ of $G = \sum_{i \in I} c_i @ R_i \langle l_i \rangle . G_i$

**Explicit re-enabling**

$\sigma(R_i) \cap l_i \neq \emptyset$

**Command causality**

if $R$ executes a command in $G_i$
then $\sigma(R) \cap l_i \neq \emptyset$ and $\sigma(R) \cap l_i \supseteq \bigcup_{R' \in \sigma G_i} \sigma(R') \cap l_i$

**Determinacy**

$R \in_{\sigma} G_i \implies l_i[0] \in \sigma(R)$

**Confusion freeness**

there is a unique subtree $G'$ of $G$ emitting $t$
for each $t$ starting a log emitted by a command in $G$
Some considerations

Further consequences:

- **Unspecified receptions** are just ignored according to the $\delta$ transition function of machines
- It is fine to violate **session fidelity**, provided that consistency is eventually attained
Some considerations

Further consequences:
- **Unspecified receptions** are just ignored according to the $\delta$ transition function of machines.
- It is fine to violate **session fidelity**, provided that consistency is eventually attained.

Care is therefore necessary
- for the definition of **correctness**
- and for the **correct realisation** of swarm protocols

Of course we appeal to projections.
On correctness

$(S, \ell)$ faithfully implements $G$ if it produces only logs possibly generated by $G$
On correctness

\((S, \ell)\) faithfully implements \(G\) if it produces only logs possibly generated by \(G\)

Exercise

Take the swarm \(S = \begin{array}{|c|c|c|c|c|c|}
\hline
P & T & O & T & \\
\hline
\end{array}\) implementing

(i.e., the swarm protocol \(G\) on slide 37). Check that \(S\) generates the log

\[\ell_{auc} = \text{requested} \cdot \text{bid} \cdot \text{bidderID} \cdot \text{selected} \cdot \text{bid} \cdot \text{bidderID} \cdot \text{passengerID}\]
On correctness

\((S, \ell)\) faithfully implements \(G\) if it produces only logs possibly generated by \(G\).

Exercise

Take the swarm \(S = P \parallel T \parallel O \parallel T\) implementing

(i.e., the swarm protocol \(G\) on slide 37). Check that \(S\) generates the log

\[
\ell_{auc} = \text{requested} \cdot \text{bid} \cdot \text{bidderID} \cdot \text{selected} \cdot \text{bid} \cdot \text{bidderID} \cdot \text{passengerID}
\]

Too strong a requirement!

Let’s consider only “good enough” logs, i.e., those typeable with \(G\)’s log types
Effective types

Let $\text{active}(\sum_{i \in I} c_i \otimes R_i \langle l_i \rangle \cdot G_i) = \bigcup_{i \in I} \{R_i\}$

\(l\) has effective type \(1\) wrt \(G\) and \(\sigma\) if \(G, \epsilon \vdash_\sigma l > 1\) is provable; where

\[
G, \epsilon \vdash_\sigma e \cdot l > t \cdot l
\]
Effective types

Let \( \text{active}(\sum_{i \in I} c_i @ R_i \langle \ell_i \rangle . G_i) = \bigcup_{i \in I} \{R_i\} \)

\( \ell \) has effective type \( 1 \) wrt \( G \) and \( \sigma \) if \( G, \epsilon \vdash_\sigma \ell \triangleright 1 \) is provable; where

\[ \vdash e : t \]

\[ G, \epsilon \vdash_\sigma e \cdot \ell \triangleright t \cdot 1 \]
Effective types

Let $\text{active}(\sum_{i \in I} c_i @ R_i \langle l_i \rangle . G_i) = \bigcup_{i \in I} \{R_i\}$

$l$ has effective type $1$ wrt $G$ and $\sigma$ if $G, \epsilon \vdash_\sigma l \triangleright 1$ is provable; where

\[
\vdash e : t \in \sigma(\text{roles}(G, \sigma)) \quad G \xrightarrow{c/t \cdot 1'} G' \\
\frac{}{G, \epsilon \vdash_\sigma e \cdot l \triangleright t \cdot 1}
\]
Effective types

Let $\text{active}(\sum_{i \in I} c_i @ R_i \langle l_i \rangle \cdot G_i) = \bigcup_{i \in I} \{R_i\}$

$l$ has effective type $1$ wrt $G$ and $\sigma$ if $G, \epsilon \vdash_{\sigma} l \triangleright 1$ is provable; where

\[
\vdash e : t \in \sigma(\text{roles}(G, \sigma)) \quad G \xrightarrow{c/t \cdot l'} G' \quad G', \text{filter}(l', \sigma(\text{active}(G'))) \vdash_{\sigma} l \triangleright 1
\]

\[
G, \epsilon \vdash_{\sigma} e \cdot l \triangleright t \cdot 1
\]
Let \( \text{active}(\sum_{i \in I} c_i @ R_i \langle l_i \rangle \cdot G_i) = \bigcup_{i \in I} \{ R_i \} \)

\( \ell \) has effective type \( l \) wrt \( G \) and \( \sigma \) if \( G, \epsilon \vdash_\sigma \ell \triangleright l \) is provable; where

\[
\frac{\vdash e : t \in \sigma(\text{roles}(G, \sigma)) \quad G \xrightarrow{c / t \cdot l'} G' \quad G', \text{filter}(l', \sigma(\text{active}(G'))) \vdash_\sigma \ell \triangleright l}{G, \epsilon \vdash_\sigma e \cdot \ell \triangleright t \cdot l}
\]

\[
\frac{\vdash e : t \quad G, l \vdash_\sigma \ell \triangleright l'}{G, t \cdot l \vdash_\sigma e \cdot \ell \triangleright t \cdot l'}
\]
Effective types

Let $\text{active}(\sum_{i \in I} c_i \Pi R_i \langle l_i \rangle . G_i) = \bigcup_{i \in I} \{R_i\}$

$l$ has effective type $1$ wrt $G$ and $\sigma$ if $G, \epsilon \vdash_\sigma l \triangleright 1$ is provable; where

\[
\vdash e : t \in \sigma(\text{roles}(G, \sigma)) \quad G \xrightarrow{c/t \cdot l'} G' \quad G', \text{filter}(l', \sigma(\text{active}(G')))) \vdash_\sigma l \triangleright 1
\]

\[
\vdash e : t \quad G, l \vdash_\sigma l \triangleright l'
\]

\[
G, t \cdot l \vdash_\sigma e \cdot l \triangleright t \cdot l'
\]

\[
G, l \vdash_\sigma \epsilon \triangleright \epsilon
\]
Effective types

Let \( \text{active}(\sum_{i \in I} c_i @ R_i \langle l_i \rangle \cdot G_i) = \bigcup_{i \in I} \{ R_i \} \)

\( \ell \) has effective type \( l \) wrt \( G \) and \( \sigma \) if \( G, \epsilon \vdash \sigma \ell \triangleright l \) is provable; where

\[
\begin{align*}
\vdash e : t \in \sigma(\text{roles}(G, \sigma)) & \quad G \xrightarrow{c/t \cdot l'} G' \quad G', \text{filter}(l', \sigma(\text{active}(G'))) \vdash \sigma \ell \triangleright l \\
\vdash e : t & \quad G, l \vdash \sigma \ell \triangleright l' \\
G, t \cdot l \vdash \sigma \ e \cdot \ell \triangleright t \cdot l' & \quad G, l \vdash \sigma \ e \triangleright \epsilon \\
G, 1 \vdash \sigma \ell \triangleright l' & \quad \text{none of the other rules applies} \\
G, 1 \vdash \sigma \ e \cdot \ell \triangleright l' &
\end{align*}
\]
Effective types

Let \( \text{active}(\sum_{i \in I} c_i \circ R_i \langle l_i \rangle \cdot G_i) = \bigcup_{i \in I} \{R_i\} \)

\( l \) has effective type \( 1 \) wrt \( G \) and \( \sigma \) if \( G, \epsilon \vdash \ell > 1 \) is provable; where

\[
\begin{align*}
\vdash e : t & \in \sigma(\text{roles}(G, \sigma)) & G \xrightarrow{c/t \cdot 1'} G' & G', \text{filter}(1', \sigma(\text{active}(G'))) \vdash \sigma \ell > 1 \\
\vdash e : t & \quad G, 1 \vdash \sigma \ell > 1' & G, 1 \vdash e \cdot \ell > t \cdot 1' & G, 1 \vdash e \cdot \ell \triangleright \epsilon \\
G, t \cdot 1 & \vdash _{\sigma} e \cdot \ell \triangleright t \cdot 1' & \quad \text{none of the other rules applies} & G, 1 \vdash \sigma \ell > 1'
\end{align*}
\]

Exercise

For the swarm protocol \( G \) on slide 37, find a condition on \( \sigma \) so that

\( G, \epsilon \vdash \ell_{\text{auc}} > \text{Requested} \cdot \text{Bid} \cdot \text{BidderID} \cdot \text{Selected} \cdot \text{PassengerID} \)
Implementations

Write \( \ell \equiv_{G,\sigma} \ell' \) when \( \ell \) and \( \ell' \) have the same effective type wrt \( G \) and \( \sigma \).

A swarm \((S, \epsilon)\) is eventually faithful to \( G \) and \( \sigma \) if \((S, \epsilon) \Rightarrow (S, \ell)\) then there is \((G, \epsilon) \Rightarrow (G, \ell')\) with \( \ell \equiv_{G,\sigma} \ell' \).
Implementations

Write \( \ell \equiv_{G, \sigma} \ell' \) when \( \ell \) and \( \ell' \) have the same effective type wrt \( G \) and \( \sigma \).

A swarm \((S, \epsilon)\) is eventually faithful to \( G \) and \( \sigma \) if \((S, \epsilon) \rightarrow (S, \ell)\) then there is \((G, \epsilon) \rightarrow (G, \ell')\) with \( \ell \equiv_{G, \sigma} \ell' \).

A \((\sigma, G)\)-realisation is a swarm \((S, \epsilon)\) of size \( n \) such that, for each \( 1 \leq i \leq n \), there exists a role \( R \in \text{roles}(G, \sigma) \) such that \( S(i) = G \downarrow_{\sigma} R \).
Implementations & projections

Write $\ell \equiv_{G,\sigma} \ell'$ when $\ell$ and $\ell'$ have the same effective type wrt $G$ and $\sigma$.

A swarm $(S,\epsilon)$ is eventually faithful to $G$ and $\sigma$ if $(S,\epsilon) \implies (S,\ell)$ then there is $(G,\epsilon) \implies (G,\ell')$ with $\ell \equiv_{G,\sigma} \ell'$

A $(\sigma,G)$-realisation is a swarm $(S,\epsilon)$ of size $n$ such that, for each $1 \leq i \leq n$, there exists a role $R \in \text{roles}(G,\sigma)$ such that $S(i) = G \downarrow_{\sigma} R$

Lemma (Projections of well-formed protocols are eventually faithful)

If $G$ is a $\sigma$-WF protocol and $(\delta(G \downarrow_{R,\ell})) \downarrow c/l$ then there exists $\ell' \equiv_{G,\sigma} \ell$ such that $(G,\epsilon) \implies (G,\ell')$ and $\delta(G,\ell') \xrightarrow{c/l} G'$
On correct realisations

A set of runs is consistent when its elements are pair-wise consistent.
On correct realisations

$(S, \ell_1)$

$(S, \epsilon)$ consistent if there is $\ell$ s.t. $(S, \epsilon) \rightarrow (S, \ell)$ with $\ell_1 = \ell \cdot \ell_1'$ and $\ell_2 = \ell \cdot \ell_2'$ and $\ell_1' \cap \ell_2' = \emptyset$

$(S, \ell_2)$

$(S, \ell')$

A set of runs is consistent when its elements are pair-wise consistent

Notation

For $(G, \epsilon) \xrightarrow{c_1/\ell_1} (G, \ell_1) \xrightarrow{c_2/\ell_2} \cdots \xrightarrow{c_n/\ell_n} (G, \ell_1 \cdot \ell_2 \cdots \ell_n)$

let $\ell(j) = \ell_j \cdots \ell_1$
On correct realisations

A set of runs is consistent when its elements are pair-wise consistent

Notation

Admissibility

A log $\ell$ is admissible for a $\sigma$-WF protocol $G$ if there are consistent runs

$\{(G, \epsilon) \rightarrow (G, \ell_i)\}_{1 \leq i \leq k}$

and a log $\ell' \in (\cup_{1 \leq i \leq k} \ell_i)$ such that $\ell = \bigcup_{1 \leq i \leq k} \ell_i$ and

$\ell' \equiv_{G, \sigma} \ell$ and $\ell_i \sqsubseteq \ell$ for all $1 \leq i \leq k$

Hereafter, $G$ denotes a $\sigma$-WF protocol
Results

Lemma (Well-formedness generates any admissible log)

If $\ell$ is admissible for $G$ then there is a log $\ell'$ such that $(G, \epsilon) \xrightarrow{c/1} (G, \ell')$ and $\ell \equiv_{G, \sigma} \ell'$

Lemma (Admissibility is preserved)

Let $\ell_1$ and $\ell_2 \subseteq \ell_1$ be admissible logs for $G$. If $(G, \ell_2) \xrightarrow{c/1} (G, \ell_2 \cdot \ell_3)$ and $\ell \in \ell_1 \bowtie (\ell_2 \cdot \ell_3)$ then $\ell$ is admissible for $G$

Theorem (Well-formed protocols generate only admissible logs)

If $(S, \epsilon) \xrightarrow{c/1} (S', \ell)$ for $(S, \epsilon)$ realisation of $G$ then $\ell$ is admissible for $G$

Corollary

Every realisation of $G$ is eventually faithful wrt $G$ and $\sigma$
On complete realisations

Complete realisations

A \((\sigma, G)\)-realisation \((S, \epsilon)\) of size \(n\) is **complete** if for all \(R \in \text{roles}(G, \sigma)\) there exists \(1 \leq i \leq n\) such that \(S(i) = G \downarrow_{\sigma} R\)

Lemma (Projections reflect swarm protocols)

If \((G, \epsilon) \Rightarrow (G, \ell)\) then \(\delta(G \downarrow_{R} \sigma, \ell) = \delta(G, \ell) \downarrow_{R} \sigma\) for all \(R \in \text{roles}(G, \sigma)\)

Theorem (Complete realisations reflect the protocol)

Let \((S, \epsilon)\) be a complete realisation of \(G\). If \((G, \epsilon) \Rightarrow (G, \ell)\) then there is a swarm \(S'\) such that \((S, \epsilon) \Rightarrow (S', \ell)\)
Plan of the talk

A motivating case study

Our formalisation

Our typing discipline

Tool support

Open issues
– Tooling –
// analogous for other events; "type" property matches type name (checked by tool)
type Requested = { type: 'Requested'; pickup: string; dest: string }
type Events = Requested | Bid | BidderID | Selected | ...

/** Initial state for role P */
@proto('taxiRide') // decorator injects inferred protocol into runtime
export class InitialP extends State<Events> {
  constructor(public id: string) { super() }
  execRequest(pickup: string, dest: string) {
    return this.events({ type: 'Requested', pickup, dest })
  }
  onRequested(ev: Requested) {
    return new AuctionP(this.id, ev.pickup, ev.dest, [])
  }
}

@proto('taxiRide')
export class AuctionP extends State<Events> {
  constructor(public id: string, public pickup: string, public dest: string,
              public bids: BidData[]) { super() }
  onBid(ev1: Bid, ev2: BidderID) {
    const [ price, time ] = ev1
    this.bids.push({ price, time, bidderID: ev2.id })
    return this
  }
  execSelect(taxiId: string) {
    return this.events({ type: 'Selected', taxiID },
                        { type: 'PassengerID', id: this.id })
  }
  onSelected(ev: Selected, id: PassengerID) {
    return new RideP(this.id, ev.taxiID)
  }
}

@proto('taxiRide')
export class RideP extends State<Events> { ... }
TypeChecking implements the functionalities of our typing discipline
simulator simulates the semantics of swarm realisations
machine-check and machine-runner integrate our framework in the Actyx platform
TypeChecking implements the functionalities of our typing discipline
simulator simulates the semantics of swarm realisations
machine-check and machine-runner integrate our framework in the Actyx platform
TypeChecking implements the functionalities of our typing discipline

 simulator simulates the semantics of swarm realisations

 machine-check and machine-runner integrate our framework in the Actyx platform
Architecture

- TypeChecking implements the functionalities of our typing discipline
- simulator simulates the semantics of swarm realisations
- machine-check and machine-runner integrate our framework in the Actyx platform
- **TypeChecking** implements the functionalities of our typing discipline
- **simulator** simulates the semantics of swarm realisations
- **machine-check** and **machine-runner** integrate our framework in the Actyx platform
If you want to play with our prototype?

Have a look at

- our ECOOP artifact paper (not online yet; extended version at https://arxiv.org/abs/2305.04848)

- code at https://doi.org/10.5281/zenodo.7737188

Plan of the talk

A motivating case study

Our formalisation

Our typing discipline

Tool support

Open issues
There are a number of future directions to explore:
There are a number of future directions to explore:

- Identify weaker conditions for well-formedness
There are a number of future directions to explore:

Identify weaker conditions for well-formedness

“Efficiency”
There are a number of future directions to explore:

- Identify weaker conditions for well-formedness
- “Efficiency”
- Subscriptions are hard to determine
There are a number of future directions to explore:

- Identify weaker conditions for well-formedness
- “Efficiency”
- Subscriptions are hard to determine
- Relax some of our assumptions
There are a number of future directions to explore:

Identify weaker conditions for well-formedness

“Efficiency”

Subscriptions are hard to determine

Relax some of our assumptions

Compensations

Unreliable propagation
There are a number of future directions to explore:

Identify weaker conditions for well-formedness

“Efficiency”

Subscriptions are hard to determine

Relax some of our assumptions

Compensations

Unreliable propagation

Adversarial contexts
There are a number of future directions to explore:

- Identify weaker conditions for well-formedness
- “Efficiency”
- Subscriptions are hard to determine
- Relax some of our assumptions
  - Compensations
  - Unreliable propagation
  - Adversarial contexts
Summary

An interesting paradigm grounded on principles for local-first software
Summary

An interesting paradigm grounded on principles for local-first software

We defined an operational semantics that captures the platform of Actyx AG
Summary

An interesting paradigm grounded on principles for local-first software

We defined an operational semantics that captures the platform of Actyx AG

We introduced behavioural types to specify and verify eventual consistency
An interesting paradigm grounded on principles for local-first software

We defined an operational semantics that captures the platform of Actyx AG

We introduced behavioural types to specify and verify eventual consistency

The key idea is to trade consistency for availability: temporary inconsistency are tolerated provided that they can be resolved at some point
Thank you!
– Solutions –
Solutions to exercises

- Slide 22: \( \delta(\text{InitialP}, \ell \cdot \text{Requested}) = \text{AuctionP} \)
- Slide 26: \( \text{src}(e) \neq \text{Alice} \)
- Slide 28: \( (a \cdot b \cdot c) \mathbin{\bowtie} (b \cdot d \cdot e) = \{a \cdot b \cdot c \cdot d \cdot e, a \cdot b \cdot d \cdot c \cdot e, a \cdot b \cdot d \cdot e \cdot c\} \)
- Slide 29: Because [prop] won’t apply since \( e \) is not a sublog of the local log of \( B \)
- Slide 41: The solution of the first exercise is in our ECOOP paper. For the second exercise, the idea is not bad because with such subscription the protocol is not well-formed (work out why)
- Slide 45: Apply the operational semantics of swarms
- Slide 46: \( \sigma(P) \ni \text{Requested, BidderID, Selected, PassengerID} \)