Behavioural Types for Local-First Software

Emilio Tuosto @ GSSI

joint work with

Roland Kuhn @ Actyx and Hernán Melgratti @ UBA

It-Matters
Lucca 11-12 July, 2023
– Prelude –
Take-away message

An approach to trade consistency for availability in systems of asymmetric replicated peers
Take-away message

An approach to

trade consistency for availability in systems of \textit{asymmetric replicated peers}

using \textit{local-first}'s principles to establish \textit{eventual consensus}
Take-away message

An approach to

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

using *local-first*’s principles to establish *eventual consensus*

formally supported by behavioural types

Take-away message

An approach to

trade consistency for availability in systems of asymmetric replicated peers

using local-first’s principles to establish eventual consensus

formally supported by behavioural types

- swarm = (machines + local logs) * imaginary global log
- swarm protocols: systems from an abstract global viewpoint
- enforce good behaviour via behavioural typing

An approach to

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

using **local-first**’s principles to establish **eventual consensus**

formally supported by behavioural types

- **swarm** = (machines + local logs) * imaginary global log
- **swarm protocols**: systems from an abstract **global** viewpoint
- enforce **good behaviour** via behavioural typing

See our recent ECOOP 2023 paper
Distributed coordination

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

With some “solutions”
Centralisation points
Consensus protocols
Commutative replicated data types
...

Availability = Money
Kohavi et al. KDD’14
Amazon sales down 1% if 100ms delay
Google searches down 0.2% - 0.6% if 100-400ms delay
Bing's revenue down $1.5% if 250ms delay
Distributed coordination

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

Availability = Money
Kohavi et al. KDD’14
- Amazon sales down 1% if 100ms delay
- Google searches down 0.2% - 0.6% if 100-400ms delay
- Bing’s revenue down ~1.5% if 250ms delay

With some “solutions”
Centralisation points
Consensus protocols
Commutative replicated data types
...

Availability
Consistency
Partitioning
A new (?) solution

What about using local-first principles?

Thou shall be autonomous

Thou shall collaborate

Thou shall recognise conflicts

Thou shall resolve conflicts

Thou shall be consistent
Plan of the talk

Some motivations

Our formalisation

Our typing discipline

Tool support

Open issues
– Motivations –
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
- when disconnected, keep calm and move on

(the pictures 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
- when disconnected, keep calm and move on

Operational model
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
- when disconnected, keep calm and move on

Operational 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
- when disconnected, keep calm and move on

Operational 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
- when disconnected, keep calm and move on

Operational 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
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
- when disconnected, keep calm and move on

Operational 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 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
- when disconnected, keep calm and move on

Operational 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
- when disconnected, keep calm and move on

Operational 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
The execution scheme

```python
while true:
    execute;
    propagate;
    merge
```
Other application domains / motivations

More applications

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

Collaborative applications (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
Also, taking decisions locally

can reduce downtime

shifts data ownership

gets rid of any centralization point...for real
Plan of the talk

A motivating case study

Our formalisation

Our typing discipline

Tool support

Future work
– A formal model –
Ingredients (I): events & logs

**Events**

\[ e \]

**Logs**

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

Events

\[ \vdash e : t \]

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

Logs

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

Events

⊢ e : t

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

Logs

⊢ e_1 \cdot e_2 \cdots : t_1 \cdot t_2 \cdots

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

[Diagram showing events e1, e2, e3]
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.
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

\[a \cdot b \cdot c\]

Phase II: newly emitted events are shipped to other machines
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
InitialP  =
Machines by example

\[
\text{InitialP} = \text{Request} \rightarrow \text{Requested}.
\]
Machines by example

\[ \text{InitialP} = \text{Request} \rightarrow \text{Requested} \cdot [\text{Requested} \Rightarrow \text{AuctionP}] \]
Machines by example

Initial\(P = \text{Request} \rightarrow \text{Requested} \cdot [\text{Requested} \ ? \ \text{AuctionP}]\)

Auction\(P = \)
Machines by example

InitialP = Request → Requested · [Requested? AuctionP]

AuctionP = Bid? BidderId? AuctionP
Machines by example

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

\[
\text{AuctionP} = \begin{cases} 
\text{Bid? BidderId? AuctionP} 
\end{cases}
\]
Machines by example

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

\[\text{AuctionP} \quad = \quad \text{Select} \iff \text{Selected} \cdot \text{PassengerId} \cdot [\text{Bid} \iff \text{BidderId} \iff \text{AuctionP}]\]
Machines by example

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

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

Think of machines as emitters/consumers of events with a semantics given in terms of state transition function:

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

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

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

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

Fix a set of commands ranged over by $c$

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

**Machine:** deterministic regular term of $M$  

\[ \kappa \cdot [t_1 ? M_1 \& \cdots \& t_n ? M_n] \]
Machines, formally

Fix a set of commands ranged over by $c$

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

**Machine:** deterministic regular term of $M \coloneqq \kappa \cdot [t_1?M_1 \& \cdots \& t_n?M_n]$

Think of machines 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 $\ell$
Fix a set of **commands** ranged over by $c$

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

**Machine:** deterministic regular term of $M \coloneqq \kappa[M_1 \& \cdots \& M_n]$

Think of machines as emitters/consumers of events with a semantics given in terms of **state transition function**:

\[
\delta(M, e) = M
\]

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

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

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 $M$
Swarms

\[ \text{Swarms: } M_1 \ell_1 | \ldots | M_n \ell_n | \ell \text{ s.t. } \ell = \bigcup_{1 \leq i \leq n} \ell_i \text{ and } \ell_i \sqsubseteq \ell \text{ for } 1 \leq i \leq n \]
Swarms: $M_1 \ell_1 | \ldots | M_n \ell_n | \ell$ s.t. $\ell = \bigcup_{1 \leq i \leq n} \ell_i$ 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
- $e <\ell_2 e', \ src(e) = src(e')$ and $e' \in \ell_1 \implies e \in \ell_1$

That is
- all events of $\ell_1$ appear in the same order in $\ell_2$
- the per-source partitions of $\ell_1$ are prefixes of the corresponding partitions of $\ell_2$
Swarms: \( M_1 \ell_1 | \ldots | M_n \ell_n | \ell \) s.t. \( \ell = \bigcup_{1 \leq i \leq n} \ell_i \) 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

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

The propagation of newly generated events happens by merging logs:

Log merging: \( \ell_1 \Join \ell_2 = \{ \ell \mid \ell \subseteq \ell_1 \cup \ell_2 \) and \( \ell_1 \sqsubseteq \ell \) and \( \ell_2 \sqsubseteq \ell \} \)

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 \( \ell_2 \)
By rule [Local] below, a command’s execution updates both local and global logs

\[
\begin{align*}
S(i) &= M_{\ell_i} \\
M_{\ell_i} &\xrightarrow{c/1} M_{\ell_i}' \\
\text{src}(\ell_i' \setminus \ell_i) &= \{i\} \\
\ell' &\in \ell \bowtie \ell_i' \\
(S, \ell) &\xrightarrow{c/1} (S[i \mapsto M'_{\ell_i}], \ell')
\end{align*}
\]
Semantics of swarms

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

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

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

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

- by shipping a \textbf{non-deterministically chosen} subset of events in the global log
- to a \textbf{non-deterministically chosen} machine
Plan of the talk

A motivating case study

Our formalisation

Our typing discipline

Tool support

Future work
– Behavioural types for swarms –
Inspired by 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”
Inspired by 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”
Inspired by 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

$M_1$
Local viewpoint$_1$

$M_i$
Local viewpoint$_i$

$M_n$
Local viewpoint$_n$

spec,no code
Inspired by 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

Well-formedness
Inspired by 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”
Inspired by 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”
An intuitive auction protocol for a passenger $P$ to get a taxi $T$:
An intuitive auction protocol for a passenger $P$ to get a taxi $T$:
Swarm protocols: global type for local-first applications

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

Fix a set of **roles** ranged over by $R$ (e.g., $P$, $T$, and $O$ on slide 31)

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

$$G \coloneqq \sum_{i \in I} c_i @ R_i \langle l_i \rangle \cdot G_i \mid 0$$

where $I$ is a finite set (of indexes)
An example

A swarm protocol for the taxi scenario on slide 31:

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

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

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

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

A swarm protocol for the taxi scenario on slide 31:

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

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

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

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

Like for machines, a swarm protocols $G = \sum_{i \in I} c_i@R_i\langle l_i \rangle \cdot G_i$ 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 $c_i@R_i\langle l_i \rangle$, written $G \xrightarrow{\text{c}_i/l_i} G_i$
Semantics of swarm protocols

One rule only!

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

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

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 log determinism pairs (command,role) differ command determinism.
Semantics of swarm protocols

**One rule only!**

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

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

\[ \text{[G-Cmd]} \]

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, i.e., on different transitions from a same state, log types start differently. Log determinism pairs \((\text{command,role})\) differ. Command determinism.
Semantics of swarm protocols

**One rule only!**

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

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

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!**

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

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

where

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

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

\(\text{log determinism}\)

\(\text{command determinism}\)
From swarm protocols to machines

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

Each machine plays one role.
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
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

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

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

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

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

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

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

where
Another attempt

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

In pub-sub, processes subscribe to “topics”

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

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

where

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

$$J = \{i \in I \mid \text{filter}(l_i, \sigma(R)) \neq \epsilon\}$$

$$\text{filter}(l, E) = \begin{cases} 
\epsilon, & \text{if } t = \epsilon \\
t \cdot \text{filter}(l', E), & \text{if } t \in E \text{ and } l = t \cdot l' \\
\text{filter}(l, E), & \text{otherwise}
\end{cases}$$
Well-formedness

Trading consistency for availability has implications:

- Propagation of events is non-atomic (cf. rule \[Prop\])
- Causality & Determinacy & Confusion freeness
- Fix a subscription \(\sigma\).
- For each branch \(i \in I\) of \(G = P_i \in I \circ_i \leftarrow R_i \langle l_i \rangle\).
- Explicit re-enabling \(\sigma(R_i) \cap l_i \neq \emptyset\)
- Command causality if \(R\) can execute a command in \(G_i\) then \(\sigma(R) \cap l_i \neq \emptyset\) and \(\sigma(R) \cap l_i \supseteq S_{R'} \in \sigma G_i \sigma(R') \cap l_i \neq \emptyset\)
- Determinacy \(R \in \sigma G_i \Rightarrow l_i[0] \in \sigma(R)\)
- Confusion freeness for each \(t\) starting a log emitted by a command in \(G\) there is a unique state \(G'\) reachable from \(G\) which emits \(t\)
- If \(R\) should have \(c\) enabled after \(c'\) then \(\sigma(R)\) contains some event type emitted by \(c'\).
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 \( G = \sum_{i \in I} c_i \oplus R_i \langle l_i \rangle \cdot G_i \)

- Explicit re-enabling \( \sigma(R_i) \cap l_i \neq \emptyset \)
- Command causality if \( R \) can execute a command in \( G_i \)
  then \( \sigma(R) \cap l_i \neq \emptyset \) and \( \sigma(R) \cap l_i \supseteq \bigcup_{R' \in \sigma G_i} \sigma(R') \cap l_i \)
Well-formedness $= \text{Causality} + \text{Determinacy}$

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

### Causality & Determinacy

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

- **Explicit re-enabling** $\sigma(R_i) \cap l_i \neq \emptyset$
- **Command causality** if $R$ can execute 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** $R \in_\sigma G_i \implies l_i[0] \in \sigma(R)$
Well-formedness = Causality + Determinacy - 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 \( G = \sum_{i \in I} c_i @ R_i (l_i) . G_i \)

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

**Command causality**
if \( R \) can execute 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**
for each \( t \) starting a log emitted by a command in \( G \)
there is a unique state \( G' \) reachable from \( G \) which emits \( t \)
Implementations

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

A \((\sigma, G)\)-realisation is a swarm \((S, \epsilon)\) such that, for each \(i \in \text{dom} \ S\), there exists a role \(R \in \text{roles}(G, \sigma)\) such that \(S(i) = G \downarrow^\sigma_R\)

Write \(\ell \equiv_{G, \sigma} \ell'\) when \(\ell\) and \(\ell'\) have the same effective type \(^{\text{effective type}}\) 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 \(\ell \equiv_{G, \sigma} \ell'\)
Implementations & projections

A \((\sigma, G)\)-realisation is a swarm \((S, \epsilon)\) such that, for each \(i \in \text{dom } S\), there exists a role \(R \in \text{roles}(G, \sigma)\) such that \(S(i) = G \downarrow^\sigma_R\)

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 \((S, \epsilon) \leadsto (S, \ell)\) then there is \((G, \epsilon) \leadsto (G, \ell')\) with \(\ell \equiv_{G, \sigma} \ell'\)

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

If \(G\) is a \(\sigma\)-WF protocol and \((\delta(G \downarrow^\sigma_R, \ell)) \downarrow^{c/1}\) then there exists \(\ell' \equiv_{G, \sigma} \ell\) such that \((G, \epsilon) \leadsto (G, \ell')\) and \(\delta(G, \ell') \xrightarrow{c/1} G'\)
On correct realisations

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

\[(S, \epsilon) \text{ consistent if there is } \ell \text{ s.t. } (S, \epsilon) \rightarrow (S, \ell) \text{ with } \ell_1 = \ell \cdot \ell_1' \text{ and } \ell_2 = \ell \cdot \ell_2' \text{ and } \ell_1' \cap \ell_2' = \emptyset\]
On correct realisations

\[(s, \ell_1) \xrightarrow{\text{consistent}} (s, \ell_2)\]

if there is \(\ell\) s.t.

\[(s, \epsilon) \rightarrow (s, \ell) \quad \text{with} \quad \ell_1 = \ell \cdot \ell_1' \quad \text{and} \quad \ell_2 = \ell \cdot \ell_2' \quad \text{and} \quad \ell_1' \cap \ell_2' = \emptyset\]

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

**Notation**

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

let \(\ell(j) = \ell_1 \cdot \ldots \cdot \ell_j\)
On correct realisations

(\(S, \ell_1\)) \[\rightarrow\] (\(S, \ell_2\))

(\(S, \epsilon\)) 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 \(\ell_1' \cap \ell_2' = \emptyset\)

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

\[\text{Notation}\]

For (\(G, \epsilon\)) \[\overset{c_1}{\rightarrow}^{\ell_1}\] (\(G, \ell_1\)) \[\overset{c_2}{\rightarrow}^{\ell_2}\] \[\ldots\] \[\overset{c_n}{\rightarrow}^{\ell_n}\] (\(G, \ell_1 \cdot \ell_2 \cdot \ldots \cdot \ell_n\))

let \(\ell(j) = \ell_1 \cdot \ldots \cdot \ell_j\)

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 \(\ell' \in (\bigwedge_{1 \leq i \leq k} \ell_i)\) such that

\[\ell = \bigcup_{1 \leq i \leq k} \ell_i,\quad \ell' \equiv_{G, \sigma} \ell,\quad \text{and} \quad \ell^{(j)}_i \sqsubseteq \ell \text{ for all } 1 \leq i \leq k\]
Results

Let $G$ be well-formed; a **realisation** is a swarm whose components are projections of $G$

**Lemma (Well-formedness generates any admissible log)**

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

**Theorem (Realisations of WF protocols are admissible)**

If $(S, \epsilon) \implies (S', \ell)$ for $(S, \epsilon)$ realisation of $G$ then $\ell$ is admissible for $G$

**Corollary**

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

**Theorem (Full realisations are complete)**

If $S$ is a **full realisation** of $G$ and $(G, \epsilon) \implies (G, \ell')$ then there is $S'$ s.t. $(S, \epsilon) \implies (S', \ell)$
Plan of the talk

A motivating case study

Our formalisation

Our typing discipline

Tool support

Future work
– 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
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
- **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 (https://drops.dagstuhl.de/opus/volltexte/2023/18254/)
- code at https://doi.org/10.5281/zenodo.7737188
Plan of the talk

A motivating case study

Our formalisation

Our typing discipline

Tool support

Future work
— Epilogue —
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
An interesting paradigm grounded on principles for local-first principles: temporary inconsistency are tolerated provided that they can be (and are) resolved at some point
An interesting paradigm grounded on principles for local-first principles: temporary inconsistency are tolerated provided that they can be (and are) resolved at some point.

A formal semantics that faithfully captures Actyx’s platform.
An interesting paradigm grounded on principles for local-first principles: temporary inconsistency are tolerated provided that they can be (and are) resolved at some point.

A formal semantics that faithfully captures Actyx’s platform

and behavioural types to specify and verify eventual consensus
Thank you!