SPLASH 2021 (series) /
OOPSLA

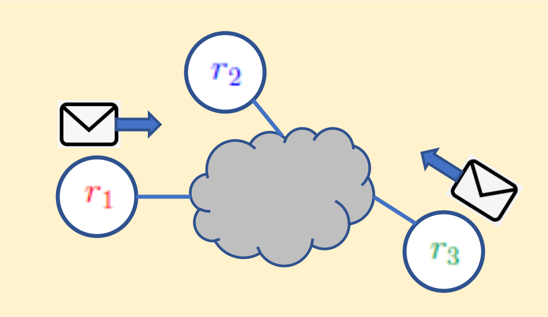
Functional and Reusable

A Multiparty Session Typing Discipline for Fault-tolerant Event-driven Distributed Programming

Malte Viering[1], **Raymond Hu**[2], Patrick Eugster[3] and Lukasz Ziarek[4]

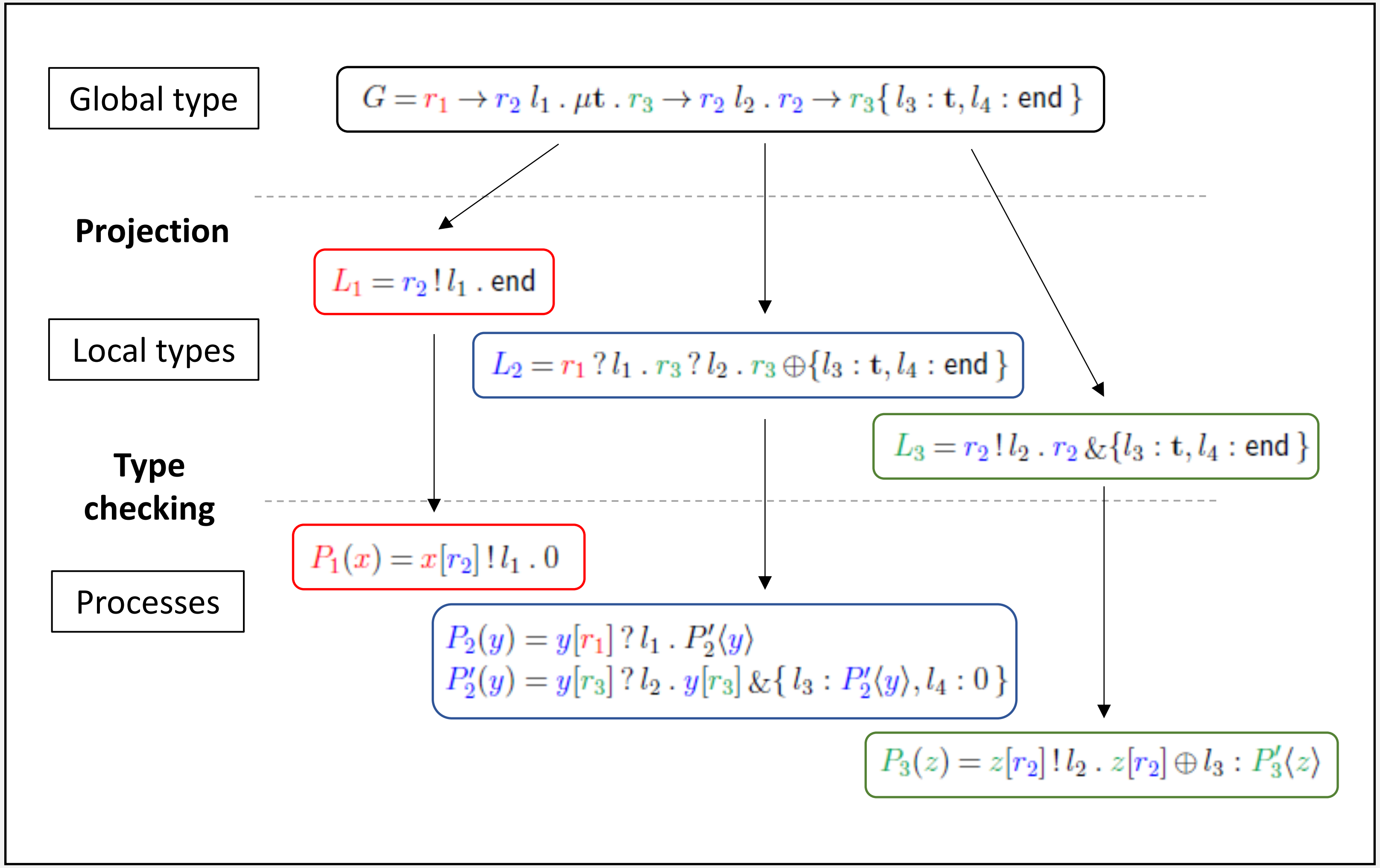
[1] Technische Universität Darmstadt malte.viering@posteo.de
[2] Queen Mary University of London r.hu@qmul.ac.uk
[3] Università della Svizzera italiana and Purdue University eugstp@usi.ch
[4] University at Buffalo lziarek@buffalo.edu

Overview: Multiparty Session Types (MSTs)



- A theoretical framework of **types for concurrent processes** that **interact in communication sessions**
  - Originally developed in (a variant) of the π-calculus **[POPL08]**

Static Typing ⇒ *Communication* Safety



Global type

$G = r_1 \to r_2\ l_1 \,.\, \mu \mathbf{t} \,.\, r_3 \to r_2\ l_2 \,.\, r_2 \to r_3 \{\, l_3 : \mathbf{t}, l_4 : \mathsf{end} \,\}$

Projection

Local types

$L_1 = r_2 \,!\, l_1 \,.\, \mathsf{end}$

$L_2 = r_1 \,?\, l_1 \,.\, r_3 \,?\, l_2 \,.\, r_3 \oplus \{ l_3 : \mathbf{t}, l_4 : \mathsf{end} \,\}$

$L_3 = r_2 \,!\, l_2 \,.\, r_2 \,\&\{ l_3 : \mathbf{t}, l_4 : \mathsf{end} \,\}$

Type checking

Processes

$P_1(x) = x[r_2] \,!\, l_1 \,.\, 0$

$P_2(y) = y[r_1] \,?\, l_1 \,.\, P_2'\langle y \rangle$
$P_2'(y) = y[r_3] \,?\, l_2 \,.\, y[r_3] \,\&\{\, l_3 : P_2'\langle y \rangle, l_4 : 0 \,\}$

$P_3(z) = z[r_2] \,!\, l_2 \,.\, z[r_2] \oplus l_3 : P_3'\langle z \rangle$

[POPL08, JACM16] *Multiparty Asynchronous Session Types*. Honda, Yoshida and Carbone.
[MSCS16] *Global progress for dynamically interleaved multiparty sessions*. Coppo, Dezani-Ciancaglini, Yoshida and Padovani.

Failures and MSTs – Existing work

> *A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.*
>
> L. Lamport (1987)

Failures are a long standing challenge for MSTs

- **[FORTE17]** *Session types for link failures*. Adameit, Peters and Nestmann.

  Synchronous communication model; failure masking via default values; not a “programming model”.

- **[ESOP18]** *A typing discipline for statically verified crash failure handling in distributed systems*.

  Detailed model of asynchronous *oracle*-based infrastructure (e.g., Zookeeper, Chubby);
  try-catch based construct to coordinate process behaviour with oracle; possibly unintuitive programming model

- “Exceptions”: e.g., **[FMSD15]** Demangeon et al., **[MSCS16]** Capecchi et al., **[CONCUR08b]** Carbone et al., …

  “Application-level” failures, rather than actual failures – all processes present and functioning correctly

```
   opt ⟨s[A]!B:l(42) … ⟩
|| opt[0]⟨s[B]?A:l(x) … ⟩
```

[FORTE17]

```
try ( m → w₁, w₂ {l₁: …, l₂: … } )
handle ( w₁ : …,
         w₂: …,
         {w₁, w₂}: … ) …
```

[ESOP18]

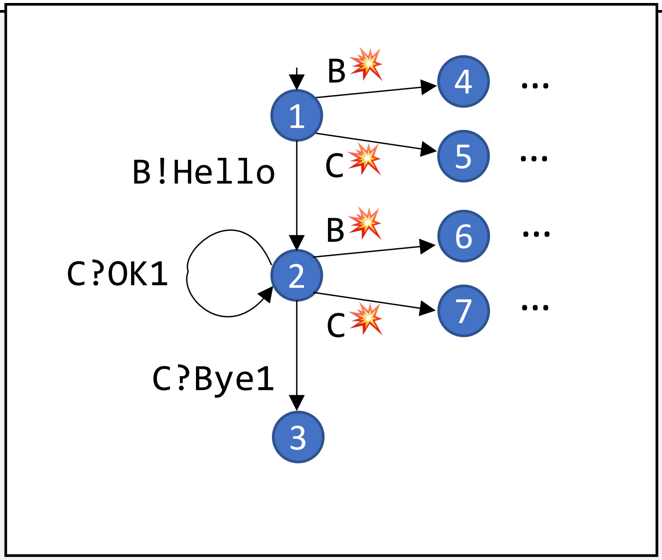
```
interruptible {
  μ t. A → B: data(). t
} with {
  B → A: stop()
}
```

[FMSD15]

Failures and MSTs – Intuition/characteristics of the challenges

L_A = ?? `B!Hello.` μ `t. C?{ OK1: t, Bye1: end }` ??

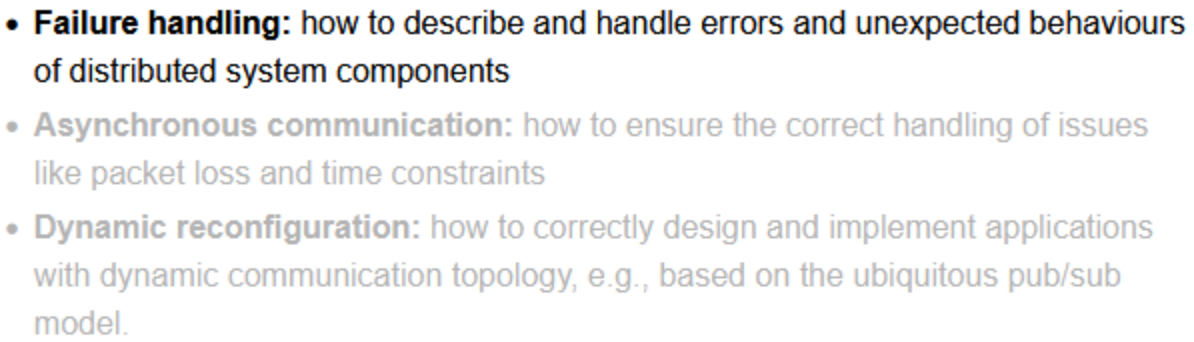
Classical MSTs
- Deterministic choice
- “Directed” choice
  (No “mixed” choice)
- “Balanced” choice cases
  (cf. projection)



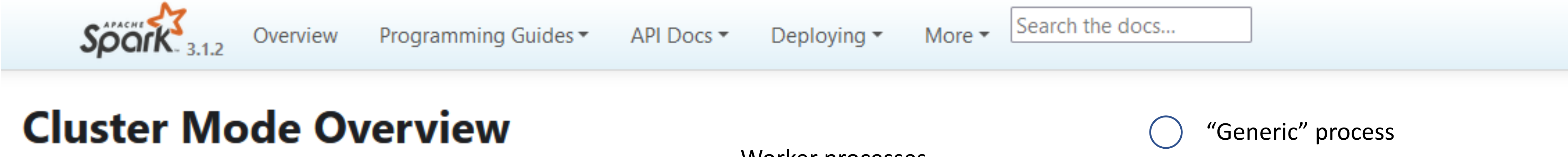
Process failures
- Asynchronous, non-deterministic and concurrent
- “Mixed” choice
- *(Unreliable* failure detection!)
- Process/role is gone!
  “Unbalanced” choice cases

Moreover: not just about modelling failures
⇒ MSTs for **fault-tolerant** application protocols
- Need a range of “advanced” features...

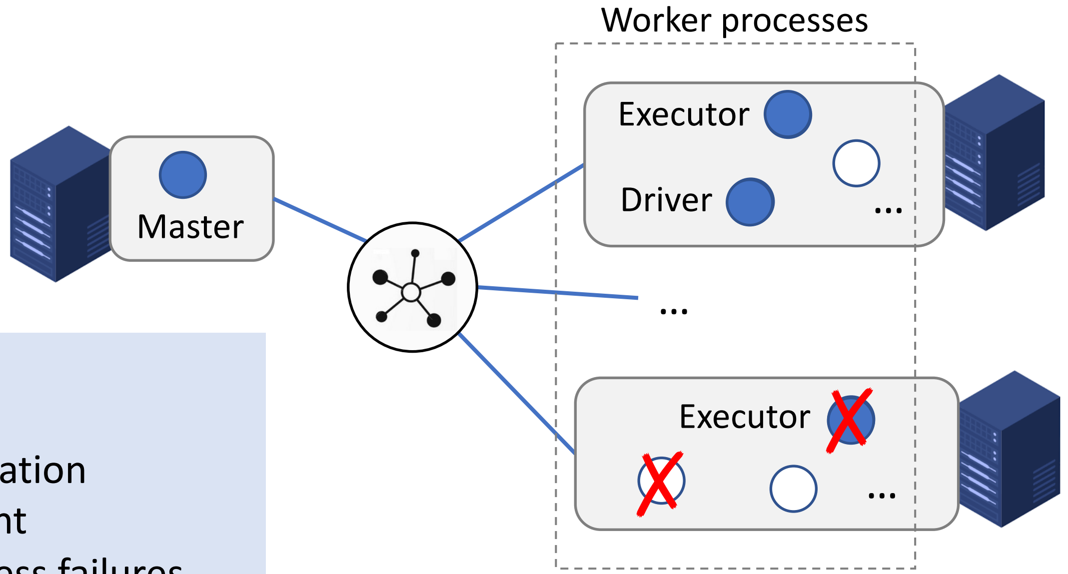
Dagstuhl Seminar 21372, Sep 2021
Behavioural Types: Bridging Theory and Practice



- **Failure handling:** how to describe and handle errors and unexpected behaviours of distributed system components
- **Asynchronous communication:** how to ensure the correct handling of issues like packet loss and time constraints
- **Dynamic reconfiguration:** how to correctly design and implement applications with dynamic communication topology, e.g., based on the ubiquitous pub/sub model.



Cluster Mode Overview



○ “Generic” process

● “Assigned” process

✗ (Suspected) Process failure

Protocol features:
- Participant parameterisation
- Dynamic role assignment
- Non-deterministic process failures

Fault-tolerant application protocols:
- *Dynamic* replacement of failed roles
- Retrying failed segments of an *ongoing* session

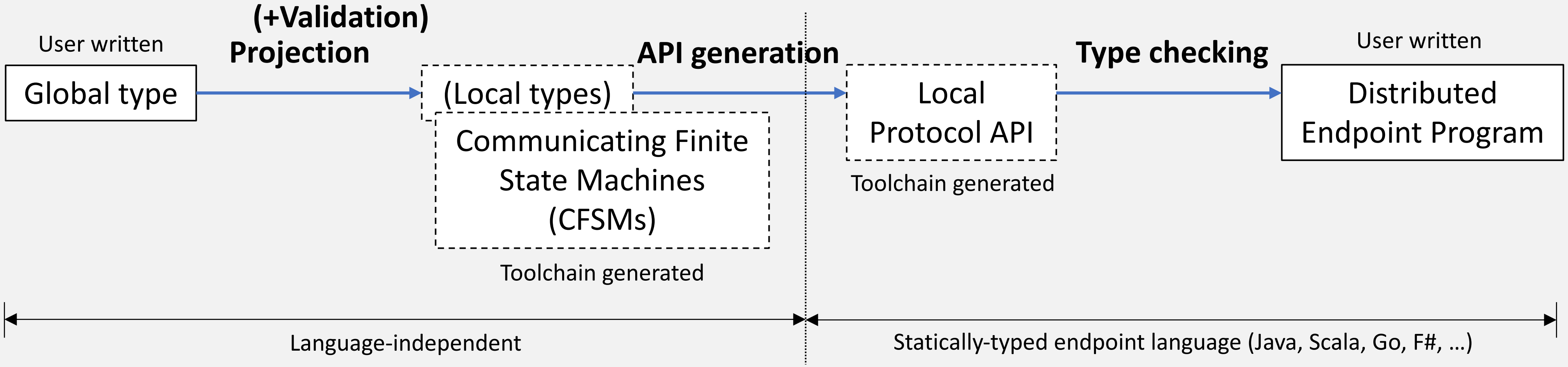
Programming model in practice:
- Concurrent subtasks
- Asynchronous I/O
- **Event-driven** concurrency

MSTs for **Fault-tolerant Event-driven** Distributed Programming

⇒ Unify “regular” I/O and failure event handling
- Integration of range of MPST features needed for fault-tolerance
- Target real-world programming model for DS

[Spark-Cluster-Mode] Spark Cluster Mode overview. https://spark.apache.org/docs/latest/cluster-overview.html
[Spark-Master] Spark Master source code.
https://github.com/apache/spark/blob/1c3bdabc03117494ffbf8fd6863ea82d4961379b/core/src/main/scala/org/apache/spark/deploy/master/Master.scala

A (Scribble-style) MST-based toolchain for practical distributed programming



User written

Global type

(+Validation)
Projection

(Local types)

Communicating Finite State Machines (CFSMs)

Toolchain generated

API generation

Local Protocol API

Toolchain generated

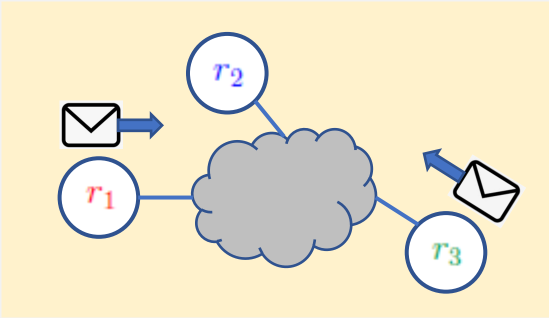
Type checking

User written

Distributed Endpoint Program

Language-independent

Statically-typed endpoint language (Java, Scala, Go, F#, …)



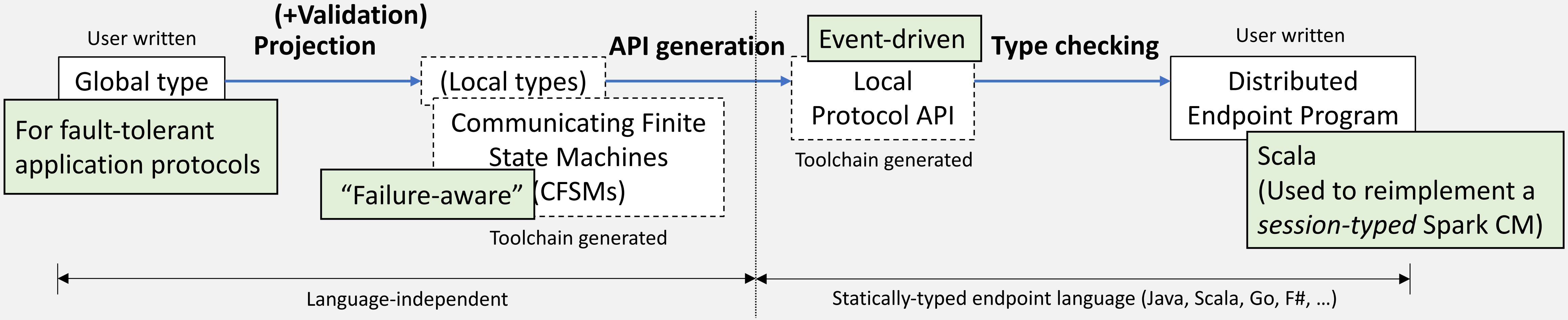
Scribble **[ECOOP17,FASE17,FASE16,TGC13]**

- Refinements for multiparty protocols **[OOPLSA20a, CC18]**
- Role-parametric protocols **[POPL19]**

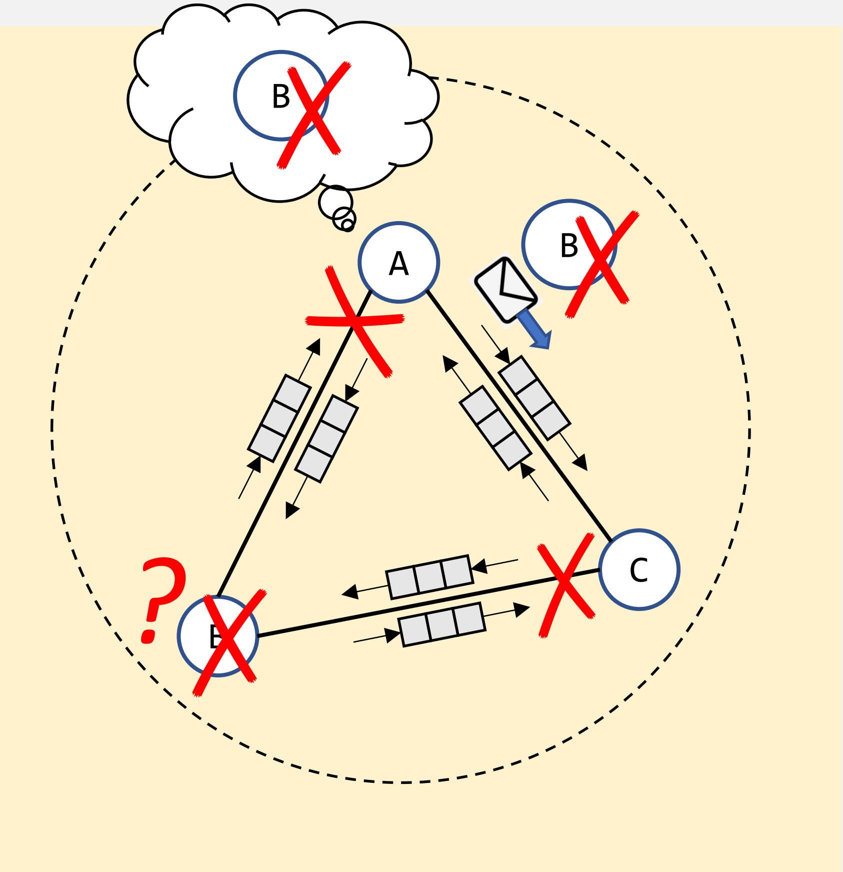
- Exceptions, failure handling and **fault-tolerant** MSTs **[OOPSLA21, ESOP18, FMSD15]**

[Scribble-Tutorial] Scribble-Java tutorial. http://www.scribble.org/docs/scribble-java.html
[Scribble-Java] Scribble-Java Github. https://github.com/scribble/scribble-java

A (Scribble-style) MST-based toolchain for practical distributed programming



A failure model for multiparty sessions



Communication model: Communicating FSMs

- Message FIFO in each direction between each pair of endpoints
- Messaging is **asynchronous** but **ordered** and **reliable** (e.g., TCP)

Failure model

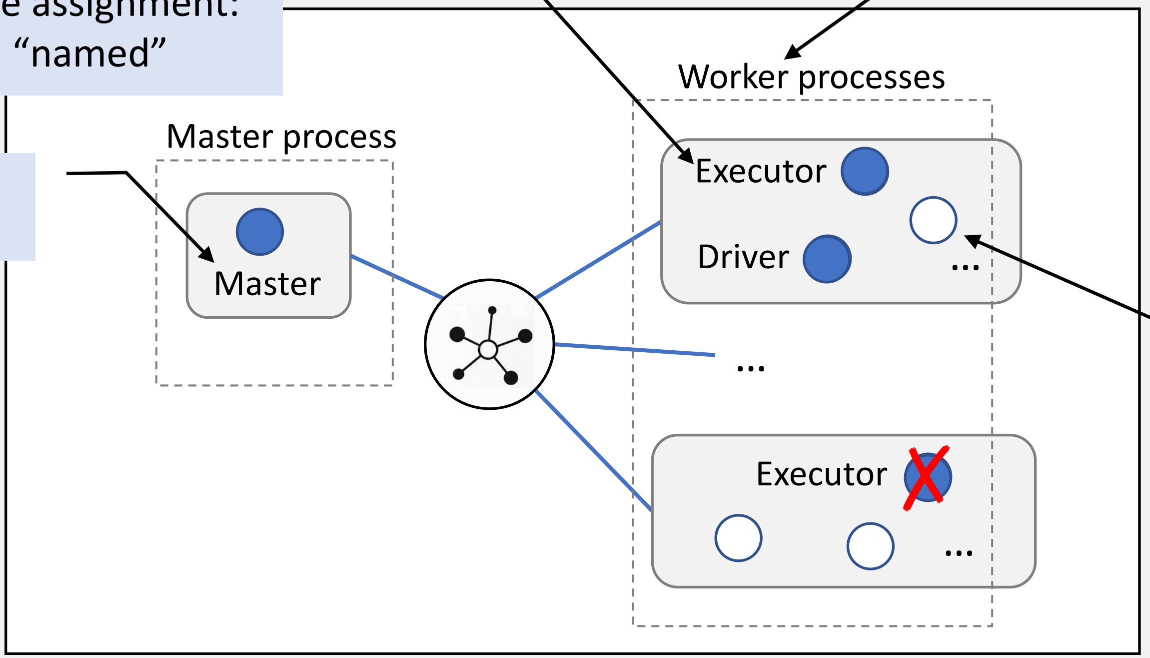
- Non-deterministic process failures – **crash-stop**
  - Minimum one **robust** role
- ***Peer*-based failure monitoring**
  - **Explicit** failure notifications to others – communication model as above
- No further assumptions ⇒ **im**perfect failure detection!
  - E.g., “false suspicions”

[ICALP13] *Multiparty Compatibility in Communicating Automata*. Deniélou and Yoshida.
[JACM83] *On Communicating Finite-State Machines*. Brand and Zafiropulo.

“Rolesets” – Practical, lightweight participant parameterisation

Dynamic role assignment:
“generic” ⇒ “named”

~~Standard MST~~ role
“Named”



Roleset – a set of participants of the same *overall* “*kind*”. Each *capable* of all the same behaviour *specialisations*.

“Generic” process
(not specialised so far)

Rolesets

- Participant parameterisation – arbitrary number of “generic” processes of the same “kind”
- Assume only some sufficient number at runtime (for role assignment)
  - Processes could be dynamically created
- Subsumes standard MSTs (each roleset is a singleton, roles assigned on session initiation)

Global type for a fault-tolerant application protocol – Spark Cluster Manager (simplified)

```
root g_Dr (roles m: M; assign w_Dr: W; rosets W) {
  m   → w_Dr: Init_Dr(Info_Dr).
  w_Dr → m:   Ack(Int).
  μ t. m → W {
     Add_Ex:  spawn g_Ex(m, w_Dr; W; W). t,
     Ok:      end
  }
with w_Dr @ m.  // m suspects w_Dr has failed: replace w_Dr and retry
  m → W: Fail_Dr(Int).
  spawn g_Dr(m; W; W). end
}

g_Ex (roles m: M, w_Dr: W; assign w_Ex: W; rosets W ) {
  m   → w_Ex: Init_Ex(Info_Ex).
  w_Ex → m:   Done_Ex(Int, Int).
  m   → w_Dr: Fin_Ex(Int, Int). end
with w_Ex @ m.  // m suspects w_Ex has failed: replace w_Ex and retry
  m → W: Fail_Ex (Int, Int).
  spawn g_Ex(m, w_Dr; W; W). end
}
```

Concurrent *sub*sessions

(Sub)sessions also involve *rolesets*

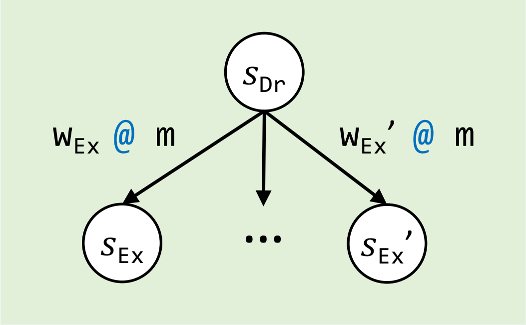
Dynamic role assignment

Peer-based failure monitoring

Explicit failure coordination

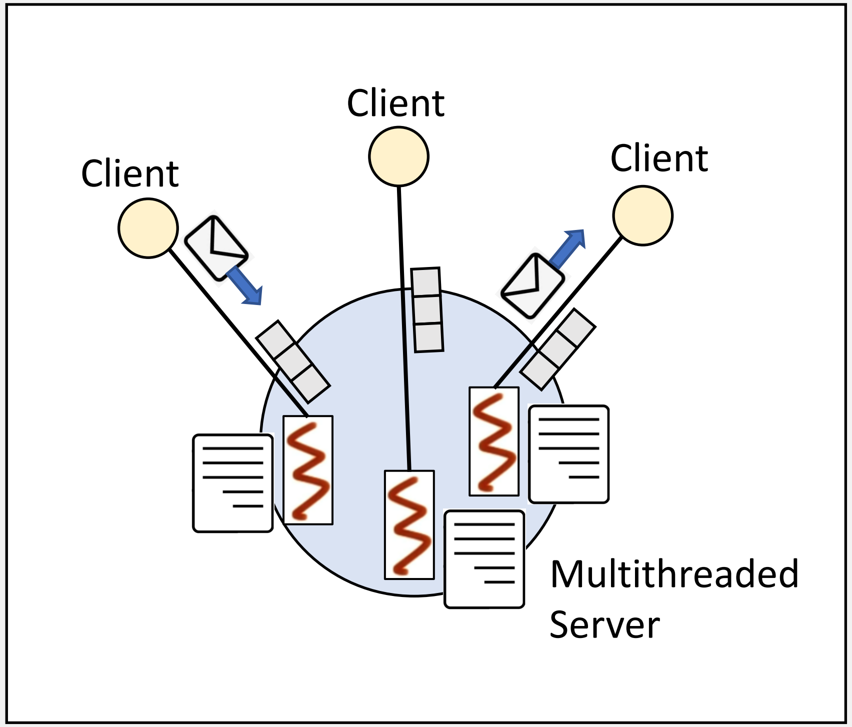
(Syntax slightly abridged)

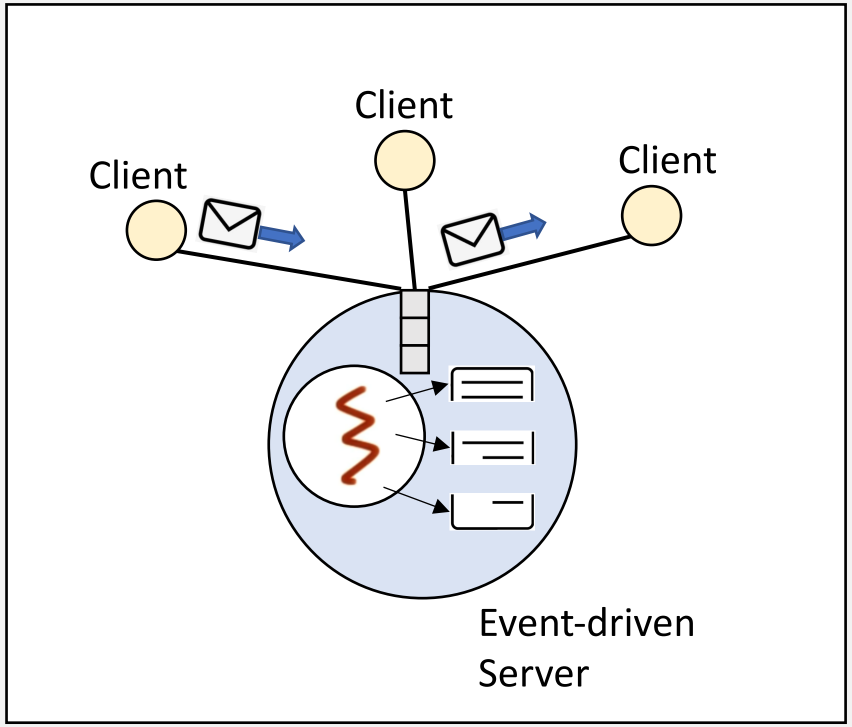
- Protocol "manually" derived from the Spark source code
- Start from a model of **concurrent subsessions**
  - Leverage session abstraction to manage I/O complexity
- Generalise the notion of each multiparty (sub)session to include
  - Interactions with **rolesets**
  - **Dynamic role assignment**
  - **Failure monitoring** of named roles
- Subsession spawning forms a parent-child tree relation
  - Leverage as a *supervision tree* for *peer-based* failure monitoring!



[CONCUR12] *Nested Protocols in Session Types*. Demangeon and Honda.

Multithreading vs. Event-driven concurrency – (e.g.) A basic Client-Server scenario





Multithreading –

Parallel composition in a typical (session) π-calculus

- Multiple threads: each is an independent unit of control flow, running a "whole program"
- Threads may block waiting on inputs

$$C_1 \mid\mid C_2 \mid\mid C_3 \mid\mid \ldots \mid\mid \underleftrightarrow{S_1 \mid\mid S_2 \mid\mid S_3} \mid\mid \ \ldots$$

Event-driven processing –

Reactive handling of event occurrences

- Single **event loop** thread: fires "program fragments" to handle event occurrences one-by-one
- Control flow (i.e., handler firing) *externally* driven by event occurrences (inversion of control)
- Event loop (should) *never* blocks

[OSR79] *On the Duality of Operating System Structures*. Lauer and Needham.
[ECOOP10] *Type-Safe Eventful Sessions in Java*. Hu, Kouzapas, Pernet and Yoshida.
[OOPSLA20] *Statically verified refinements for multiparty protocols.* Zhou, Ferreira, Hu, Neykova and Yoshida.

Event-driven multiparty sessions in Scala

```
root g_Dr (roles m: M; assign w_Dr: W; rosets W) {
  m    → w_Dr: Init_Dr(Info_Dr).
  w_Dr → m:    Ack(Int).
  μ t. m → W {
    Add_Ex: spawn RunEx(m, w_Dr; W; W). t
    Ok:     end
  }
with w_Dr @ m.
  m → W: Fail_Dr(Int).
  spawn g_Dr(m; W; W). end
}

g_Ex (roles m: M, w_Dr: W; assign w_Dr: W; rosets W) {
  m    → w_Ex: Init_Ex(Info_Ex).
  w_Ex → m:    Done_Ex(Int, Int).
  m    → w_Dr: Fin_Ex(Int, Int). end
with w_Ex @ m.
  m → W: Fail_Ex (Int, Int).
  spawn g_Ex(m, w_Dr; W; W). end
}
```

Output

Input

Subsession initiation

Failure
(suspicion)

Session-typed event loop

- Tracks the "*current protocol state*" at run-time
- Dispatches events based on the pair *(current state, event occurrence)*
- Branching/selection enacted by handler dispatch
- Recursion driven by repeat *(state+event)* occurrences

```
λ (e) { case (d, c) => ( …d'…, …c'… ) }
```

```
// I/O event(s)          // Event handler functions
λ(SndInit_Dr)           { case (s, c: M1) => (s, c ! Init_Dr(…)) }
λ(RcvAck)               { case (s, c: M2) => (s, (c ? ())._2) }
λ(SndAdd_Ex)            { case (s, c: M3) if s.workRemaining() => (s, c ! Add_Ex()) }
λ(SndOk)                { case (s, c: M3) => (s, c ! Ok()) }
λ(Spw_Ex)               { case (s, c: M4) => (s, c.init(…)) }
λ(Sus_Dr, SndFail_Dr)   { case (s, c: M6) => (s, c.failure() ! Fail_Dr(s.appId)) }
λ(Spw_Dr)               { case (s, c: M8) => (s, c.init(…)) }
```

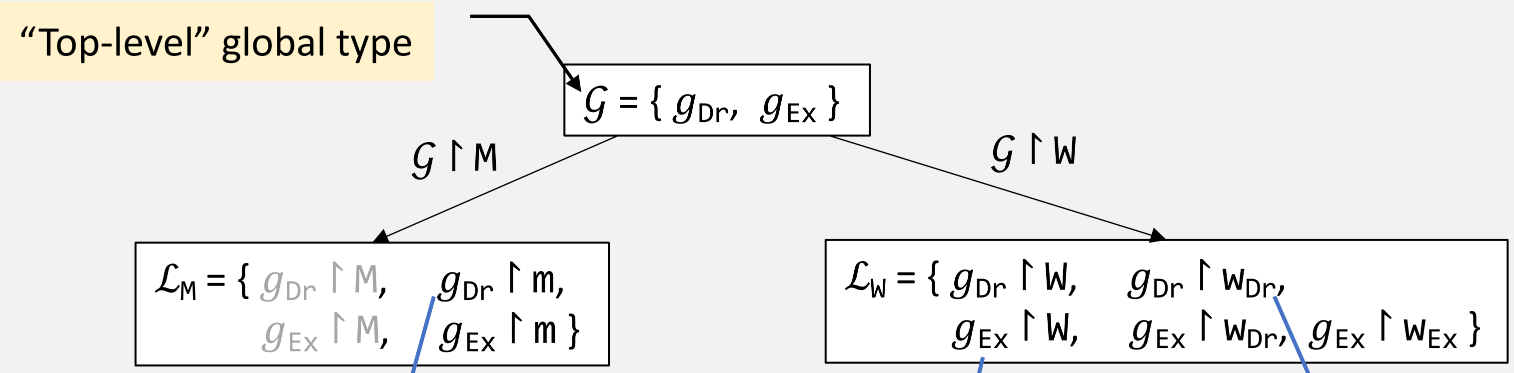
Output

Input

Subsession initiation

Failure
(suspicion)

Projection on to *rolesets* – Representation as “failure-aware” CFSMs



“Top-level” global type

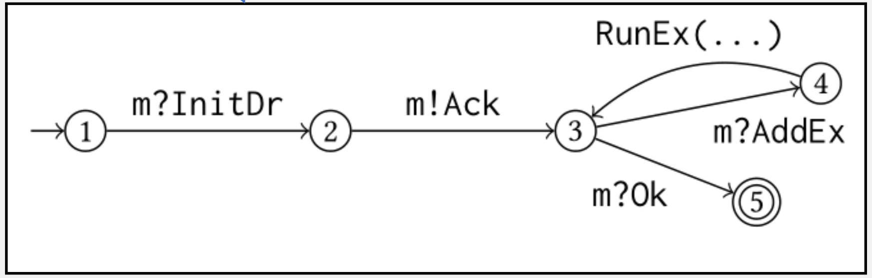
$\mathcal{G} = \{ g_{\mathtt{Dr}},\ g_{\mathtt{Ex}} \}$

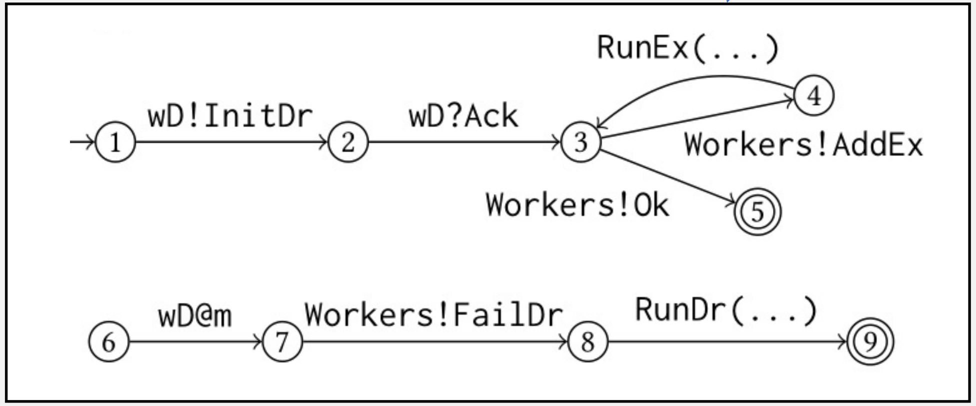
$\mathcal{G} \restriction \mathtt{M}$

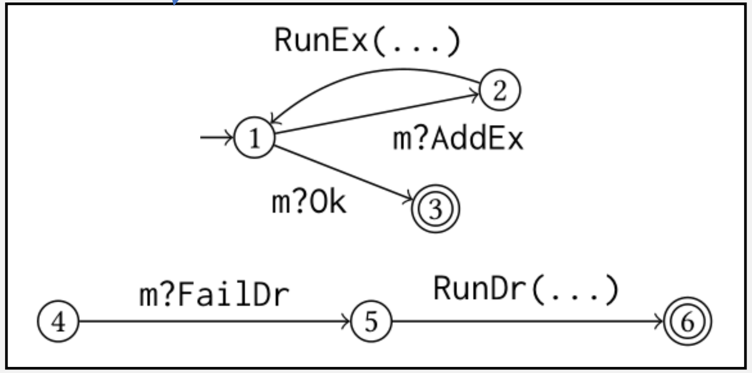
$\mathcal{G} \restriction \mathtt{W}$

$\mathcal{L}_{\mathtt{M}} = \{ g_{\mathtt{Dr}} \restriction \mathtt{M},\ g_{\mathtt{Dr}} \restriction \mathtt{m},\ g_{\mathtt{Ex}} \restriction \mathtt{M},\ g_{\mathtt{Ex}} \restriction \mathtt{m} \}$

$\mathcal{L}_{\mathtt{W}} = \{ g_{\mathtt{Dr}} \restriction \mathtt{W},\ g_{\mathtt{Dr}} \restriction \mathtt{w}_{\mathtt{Dr}},\ g_{\mathtt{Ex}} \restriction \mathtt{W},\ g_{\mathtt{Ex}} \restriction \mathtt{w}_{\mathtt{Dr}},\ g_{\mathtt{Ex}} \restriction \mathtt{w}_{\mathtt{Ex}} \}$



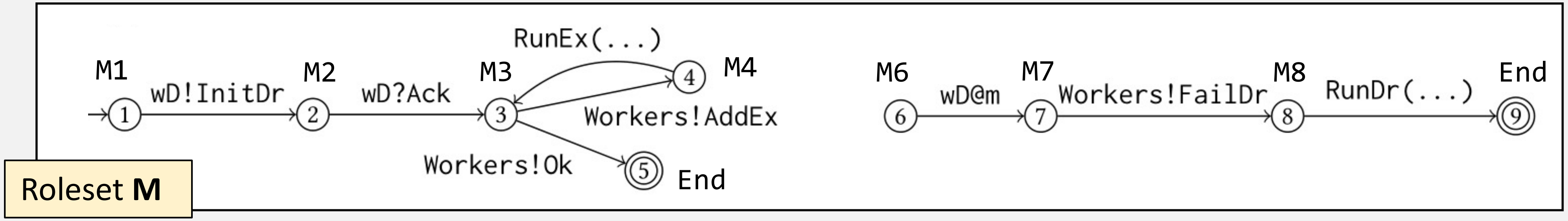




`r!l` Output `r?l` Input `SubProto(...)` Spawn `r@r'` Suspicion

◎ End (Performs a final sync. between all subsess. participants)

(Scala) API generation for Event-driven session programming



State	Chan. type	I/O methods	Return type
1	M1	`!(InitDr)`	M2
2	M2	`?()`	`(Ack, M3)`
3	M3	`!(AddEx)`	M4
		`!(Ok)`	End
4	M4	`init(…)`	M3

State	Chan. type	I/O methods	Return type
5, 9	End		
6	M6	`failure()`	M7
7	M7	`!(FailDr)`	M8
8	M8	`init(…)`	End

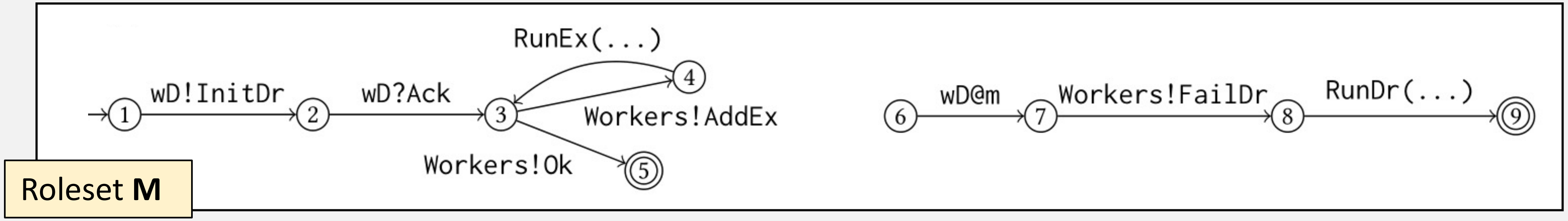
(N.B. ! And ? are method names)

```
def runNormalM(Data d, M1 m1): End {
  val m2 = m1 ! InitDr(…)
  var m3 = (m2 ? ())._2
  while (…d.workRemaining()…) {
    m3 = (m3 ! AddEx(…)). …init(…)…
  }
  m3 ! Ok(…)
}
```

(For safety, this basic approach assumes *dynamic* checking of **linear** usages of session channels – specifically, no channel instance used *more* than once… more on linearity later!)

[FASE16] *Hybrid Session Verification through Endpoint API Generation*. Hu and Yoshida.
[CONCUR04] *Session Types for Functional Multithreading*. Vasconcelos, Ravara, and Gay.

(Scala) API generation for Event-driven session programming



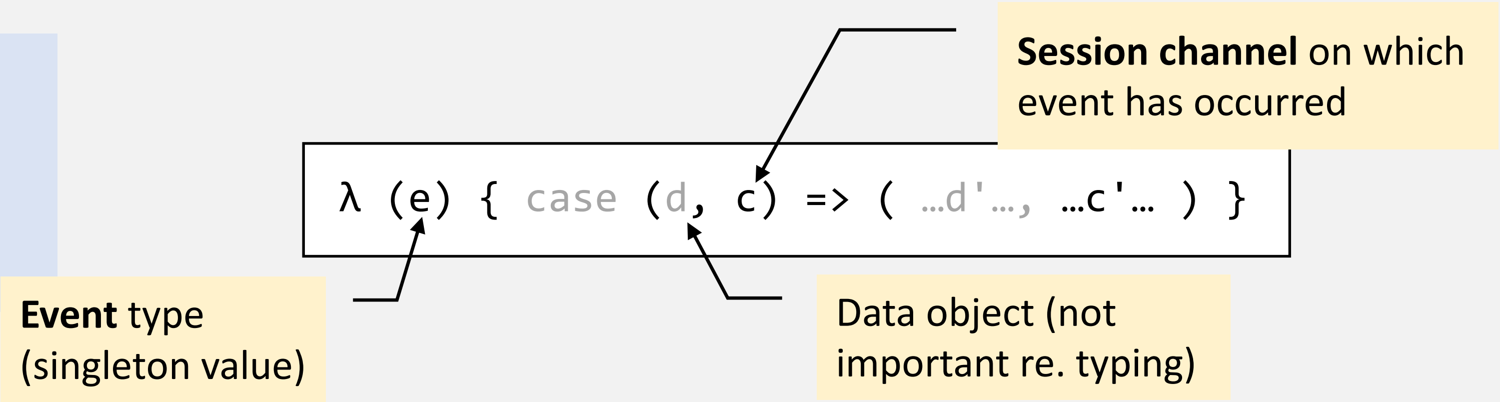
Roleset **M**

State	Chan. type	I/O methods	Return type	Event type
1	M1	`!(InitDr)`	M2	SndInitDr
2	M2	`?()`	(Ack, M3)	RcvAck
3	M3	`!(AddEx)`	M4	SndAddEx
		`!(Ok)`	End	SndOk
4	M4	`init(…)`	M3	SpwRunEx

State	Chan. type	I/O methods	Return type	Event type
5, 9	End			
6	M6	`failure()`	M7	SuswD
7	M7	`!(FailDr)`	M8	SndFailDr
8	M8	`init(…)`	End	SpwRunDr

(N.B. ! And ? are method names)

For each session I/O event, provide a **callback** function to handle occurrences of that event



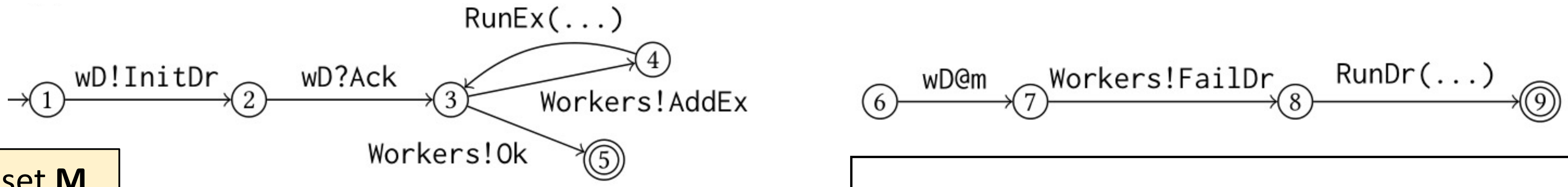
`λ (e) { case (d, c) => ( …d'…, …c'… ) }`

Session channel on which event has occurred

Event type (singleton value)

Data object (not important re. typing)

Event-driven endpoint implementation in Scala



Roleset **M**

```
λ (e) { case (d, c) => ( …d'…, …c'… ) }
```

```
    // I/O event(s)     // Event handler functions
1   λ(SndInitDr)        { case (s, c: M1) => (s, c ! InitDr(…)) }
2   λ(RcvAck)           { case (s, c: M2) => (s, (c ? ())._2) }
3   λ(SndAddEx)         { case (s, c: M3) if s.workRemaining() => (s, c ! AddEx()) }
3   λ(SndOk)            { case (s, c: M3) => (s, c ! Ok()) }
4   λ(SpwEx)            { case (s, c: M4) => (s, c.init(…)) }
6   λ(SusDr, SndFailDr) { case (s, c: M6) => (s, c.failure() ! FailDr(s.appId)) }
8   λ(SpwDr)            { case (s, c: M8) => (s, c.init(…)) }
```

Output

Input

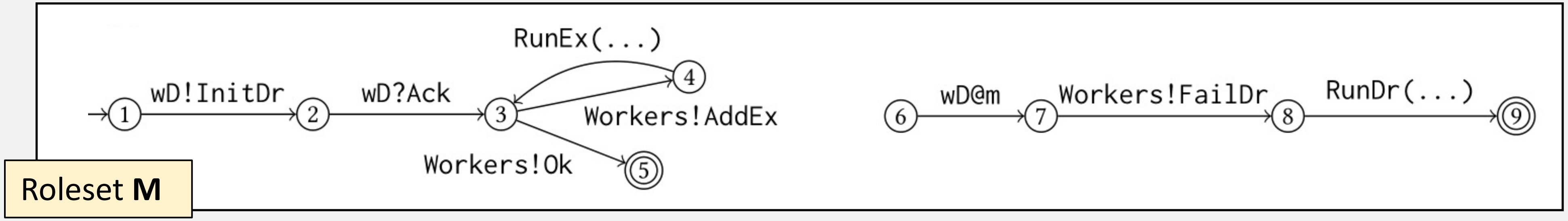
Failure (suspicion)

Subsession initiation

Session-typed event loop

- Tracks the "*current protocol state*" at run-time
- Dispatches events based on the pair *(current state, event occurrence)*
- Branching/selection enacted by handler dispatch
- Recursion driven by repeat *(state+event)* occurrences

Event-driven endpoint implementation in Scala



Roleset **M**

…alternatively: don't expose the channels!
⇒ linearity violations *impossible*

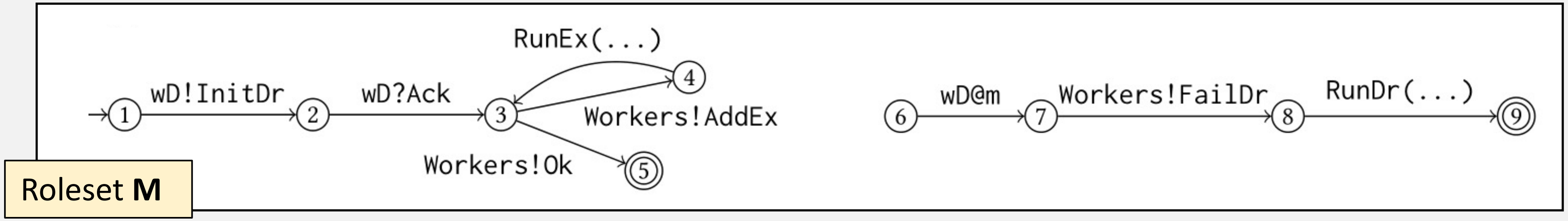
```
    // I/O event(s)     // Event handler functions
1   λ(SndInitDr)        { case (s        ) => (s,     InitDr(…)) }
2   λ(RcvAck)           { case (s, x: Ack) => (s, …) }
3   λ(SndAddEx)         { case (s        ) if s.workRemaining() => (s,     AddEx()) }
3   λ(SndOk)            { case (s        ) => (s,     Ok()) }
4   λ(SpwEx)            { case (s        ) => (s,        … ) }
6   λ(SusDr, SndFailDr) { case (s        ) => (s,                FailDr(s.appId)) }
8   λ(SpwDr)            { case (s        ) => (s,        … ) }
```

Session-typed event loop

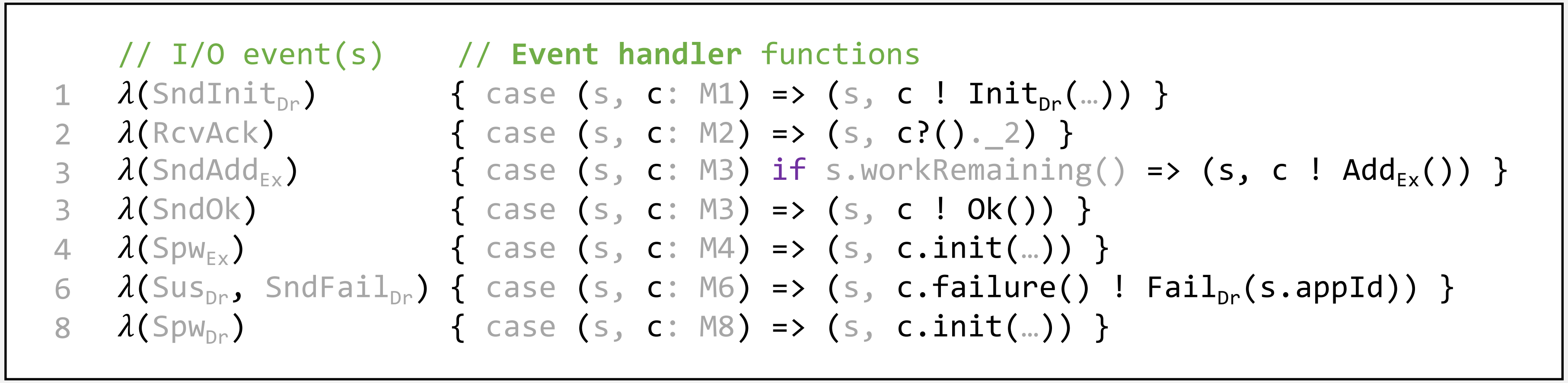
- Tracks the "*current protocol state*" at run-time
- Dispatches events based on the pair *(current state, event occurrence)*
- Branching/selection enacted by handler dispatch
- Recursion driven by repeat *(state+event)* occurrences

[OOPSLA20] *Statically verified refinements for multiparty protocols.* Zhou, Ferreira, Hu, Neykova and Yoshida.

Properties of failure-prone, event-driven sessions



Roleset **M**



```
    // I/O event(s)       // Event handler functions
1   λ(SndInitDr)          { case (s, c: M1) => (s, c ! InitDr(…)) }
2   λ(RcvAck)             { case (s, c: M2) => (s, c?()._2) }
3   λ(SndAddEx)           { case (s, c: M3) if s.workRemaining() => (s, c ! AddEx()) }
3   λ(SndOk)              { case (s, c: M3) => (s, c ! Ok()) }
4   λ(SpwEx)              { case (s, c: M4) => (s, c.init(…)) }
6   λ(SusDr, SndFailDr)   { case (s, c: M6) => (s, c.failure() ! FailDr(s.appId)) }
8   λ(SpwDr)              { case (s, c: M8) => (s, c.init(…)) }
```

THEOREM 6.3 (SUBJECT REDUCTION). *Let* $\vdash (\Theta_1, \mathcal{F}_1, (vs : \mathcal{G})\ N_1)$ *such that* $(\Theta_1, \mathcal{F}_1, (vs : \mathcal{G})\ N_1) \to (\Theta_2, \mathcal{F}_2, (vs : \mathcal{G})\ N_2)$. *Then* $\vdash (\Theta_2, \mathcal{F}_2, (vs : \mathcal{G})\ N_2)$.

COROLLARY 6.4 (COMMUNICATION SAFETY). *Let* $\vdash (\Theta, \mathcal{F}, (vs : \mathcal{G})\ N))$. *For every session s in* Θ *and unsuspected p in s, p has the following properties: (i) p does not have a reception error; (ii) p is not stuck; and (iii) p does not have a non-covered failure.*

(Perhaps not *actually* failed!)

No unknown messages

No WFC between *unsuspected*

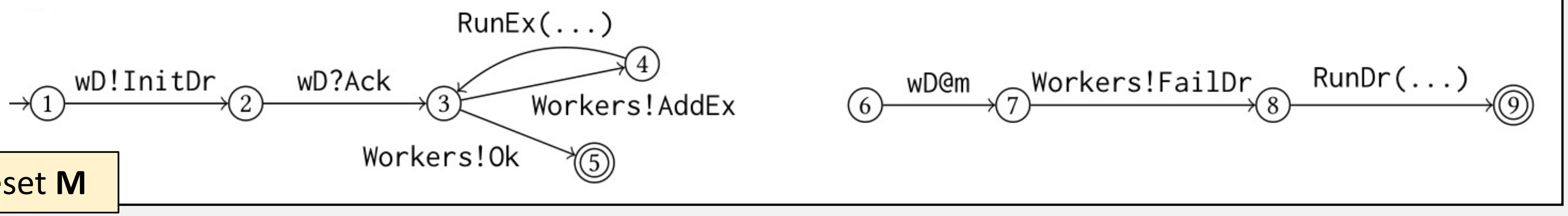
Never left permanently hanging due to a failure

- Session typing within handlers: a *single* session channel and "*flat*" – much simpler than standard session processes!

Instead:

- **Non-blocking** handlers
- **Coverage** (and global type structure/WF)

Properties of failure-prone, event-driven sessions



Roleset M

```
    // I/O event(s)      // Event handler functions
1   λ(SndInitDr)         { case (s, c: M1) => (s, c ! InitDr(…)) }
2   λ(RcvAck)            { case (s, c: M2) => (s, c?()._2) }
3   λ(SndAddEx)          { case (s, c: M3) if s.workRemaining() => (s, c ! AddEx()) }
3   λ(SndOk)             { case (s, c: M3) => (s, c ! Ok()) }
4   λ(SpwEx)             { case (s, c: M4) => (s, c.init(…)) }
6   λ(SusDr, SndFailDr)  { case (s, c: M6) => (s, c.failure() ! FailDr(s.appId)) }
8   λ(SpwDr)             { case (s, c: M8) => (s, c.init(…)) }
```

Every (sub)session is completed

THEOREM 6.6 (GLOBAL PROGRESS). *Assume an initial system* $\vdash (\Theta_1, \mathcal{F}_1, (vs : \mathcal{G})\ N_1)$ *and a reduction* $(\Theta_1, \mathcal{F}_1, (vs : \mathcal{G})\ N_1) \to^* (\Theta_2, \mathcal{F}_2, (vs : \mathcal{G})\ N_2)$. *Then either* Θ_2 *is empty, or without using* SUSP *we have* $(\Theta_2, \mathcal{F}_2, (vs : \mathcal{G})\ N_2) \to (\Theta_3, \mathcal{F}_3, (vs : \mathcal{G})\ N_3)$.

Can make a step…

…without "cheating" by just failing (if stuck)

- A process *never* engages in I/O unless event is **ready**
- Progress of *individual* (sub)sessions is **independent**

Thus, a (sub)session action is:

- Itself never blocked if fired, i.e., when actually executed
- Never blocked from firing by actions of *another* (sub)session

(Caveat: data object guards…)

[MSCS16] *Global progress for dynamically interleaved multiparty sessions*. Coppo, Dezani-Ciancaglini, Yoshida and Padovani.

Prototype implementation – Practical evaluation

Failure-aware extension of examples from MST literature

- Subsumes classical MSTs
- Rolesets support patterns involving parameterised numbers of participants
- Can encode "application-level failures" (exceptions/interrupts)

(1) *Core MPSTs (MP interactions, choice, recursion)*

2-Buyers, Streaming [Honda et al. 2016]
Sutherland-Hodgman [Neykova et al. 2018]

(2) *Dynamic/parameterized participants*

3-Buyers [Coppo et al. 2016]
N-stage Pipe [Castro-Perez et al. 2019]
N-stage Ring [Castro-Perez et al. 2019]

(3) *Application-level exceptions/interrupts*

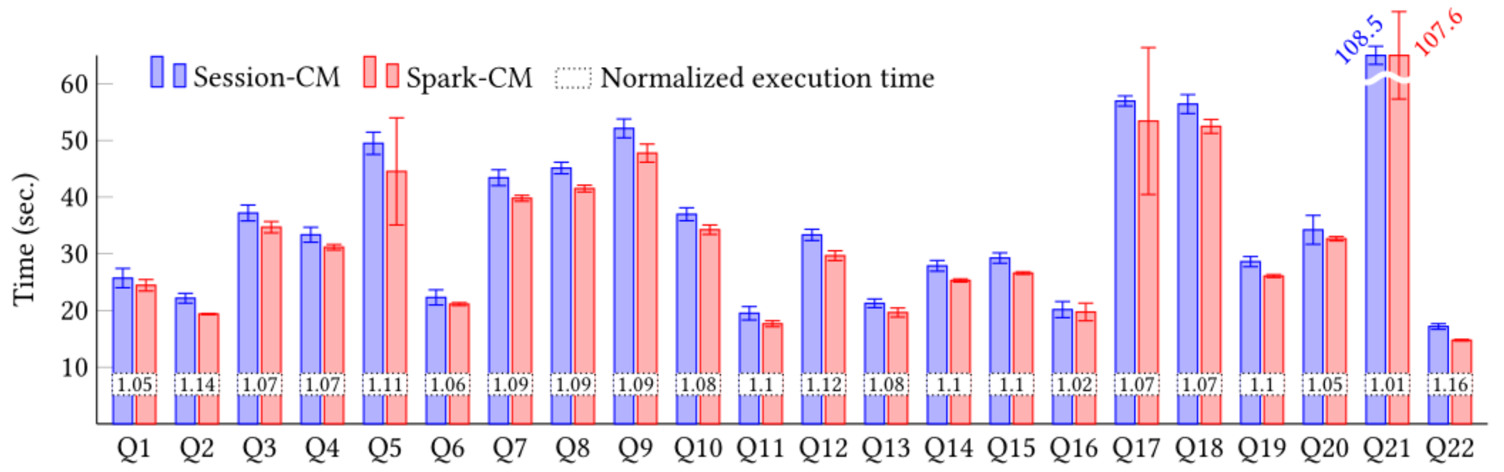
Two Factor [Fowler et al. 2019]
Resource Control [Demangeon et al. 2015]
WebCrawler [Neykova and Yoshida 2017]
Interruptible 3-Buyers [Capecchi et al. 2016]

Basic failure handling (cf. Fig. 12)

Failure-Aware Streaming [Viering et al. 2018]

Session-CM: *Session-typed* Spark cluster manager

- Executes third-party Spark applications *without* code modification
- E.g., TPC-H benchmark suite
  Average overhead <10%
  Max. overhead <16.5%
- Failure scenario
  (Q18, Executor killed after 20s):
  overhead ~10%



TPC-H Spark (database ~10GB).
Each query as a separate application.
Three servers.

[TPCH] TPC-H benchmark suite. http://www.tpc.org/tpch/
[TPCH-Spark] TPC-H Spark. Savvides. https://github.com/ssavvides/tpch-spark