iCyPhy

Distributed Real-Time Computing

Edward A. Lee

Professor of the Graduate School
Distinguished Professor Emeritus

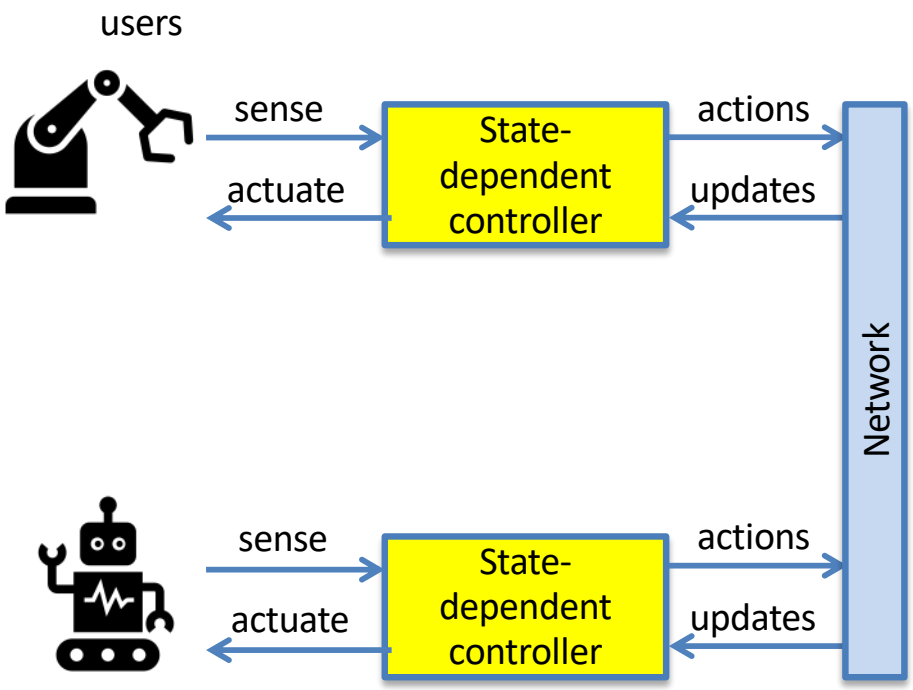
Keynote

38-th European Conference on Real-Time Systems (ECRTS)

July 8, 2026, Lund, Sweden

University of California at Berkeley

Distributed Cyber-Physical Systems



Examples

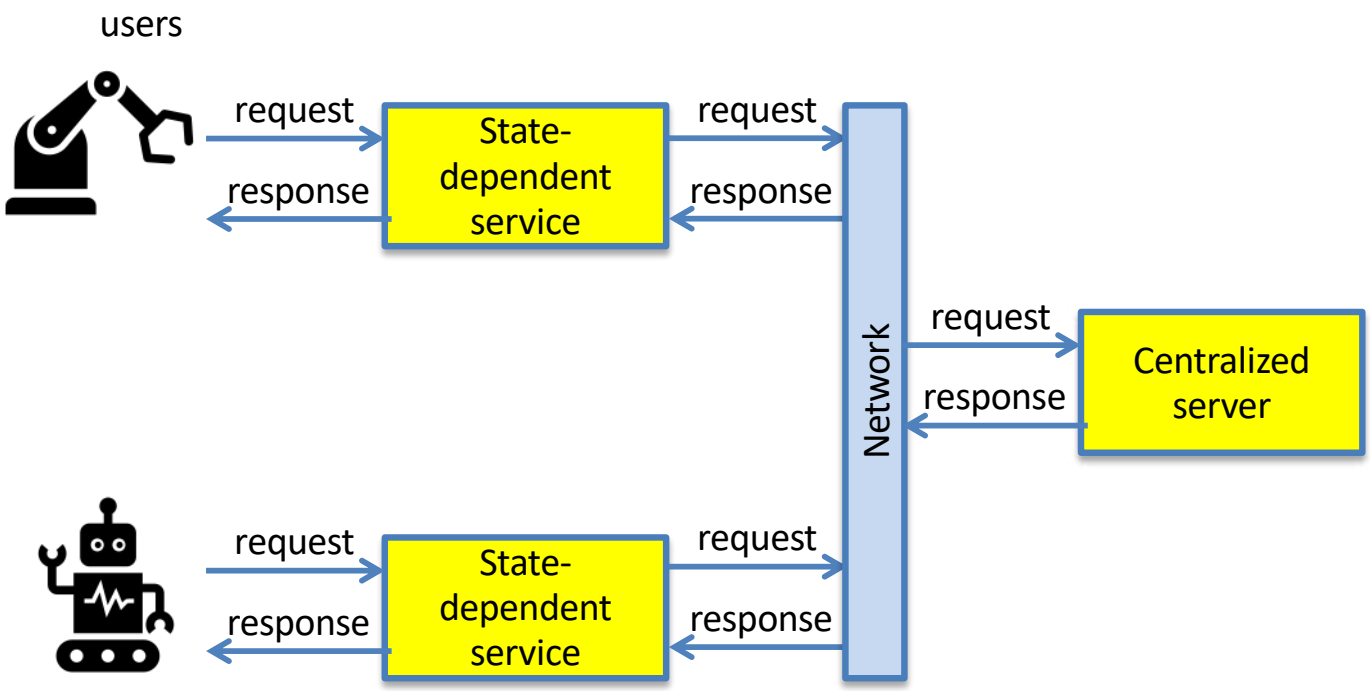
- Factory floor
- Traffic control
- Power systems

Requires a *consistent* view of system state

Requirements:

- Distributed control
- Mutual exclusion
- Safe resource sharing
- Operating limits
- Fault management

Classic Centralized Solution



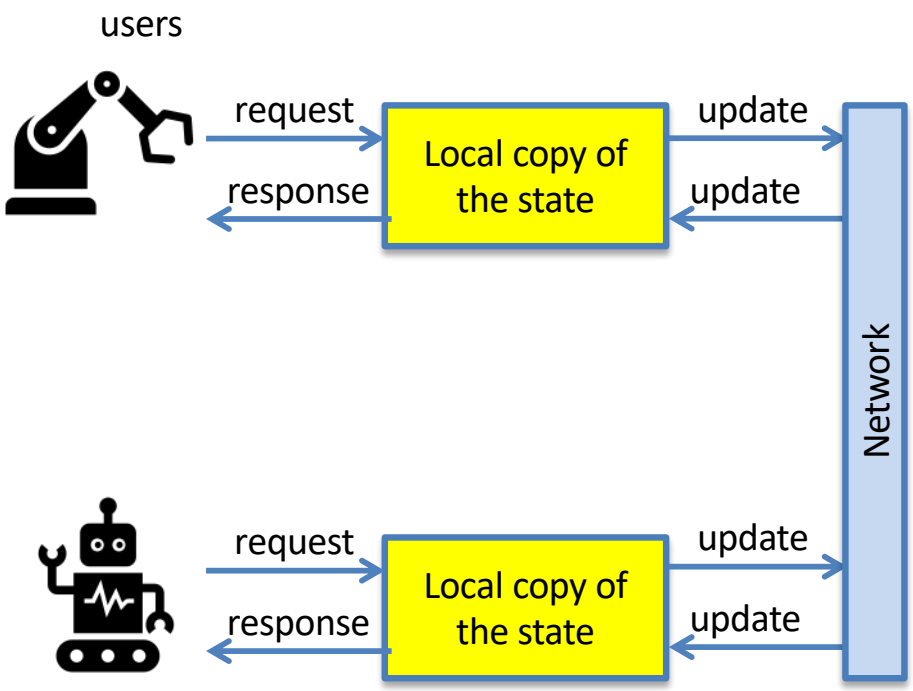
Problems

- Single point of failure, or
- Complex leader election
- High latency
- Bottleneck

CAP Theorem: Cannot have

- Consistency
- Availability
- Partition tolerance

Decentralized Solutions



Today's approaches:

- Publish-and-subscribe
  - ROS, MQTT, DDS, Azure, Google Cloud, …
- Actors
  - Erlang, Akka, Orleans, Scala, …
- Remote procedure calls
  - gRPC, Bond, Thrift, …
- Shared memory
  - Threads, Linda, pSpaces, …

Problems:

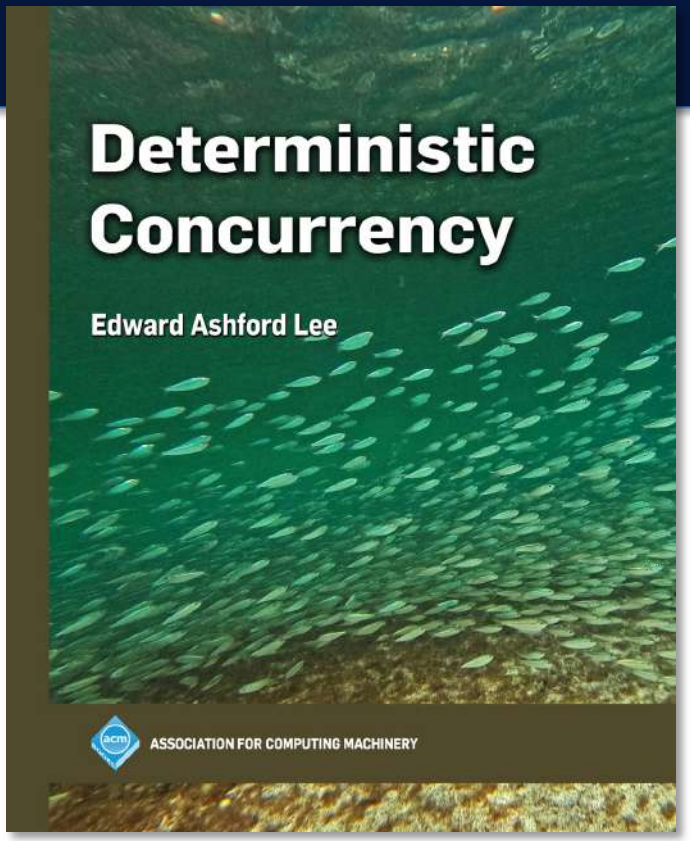
- Inconsistency
- Nondeterminism
- Untestable

Deterministic Concurrency

Not a new topic:

- Dataflow (1970s - …)
- Kahn networks (1970s - …)
- Discrete-event systems (1980s - …)
- Synchronous languages (1980s - …)
- Logical execution time (2000s - …)
- Reactors (2020s - …)

And yet, nondeterministic methods dominate…



Forthcoming book

Our Principle: Logical Time

Separate “logical time” from “physical time.”

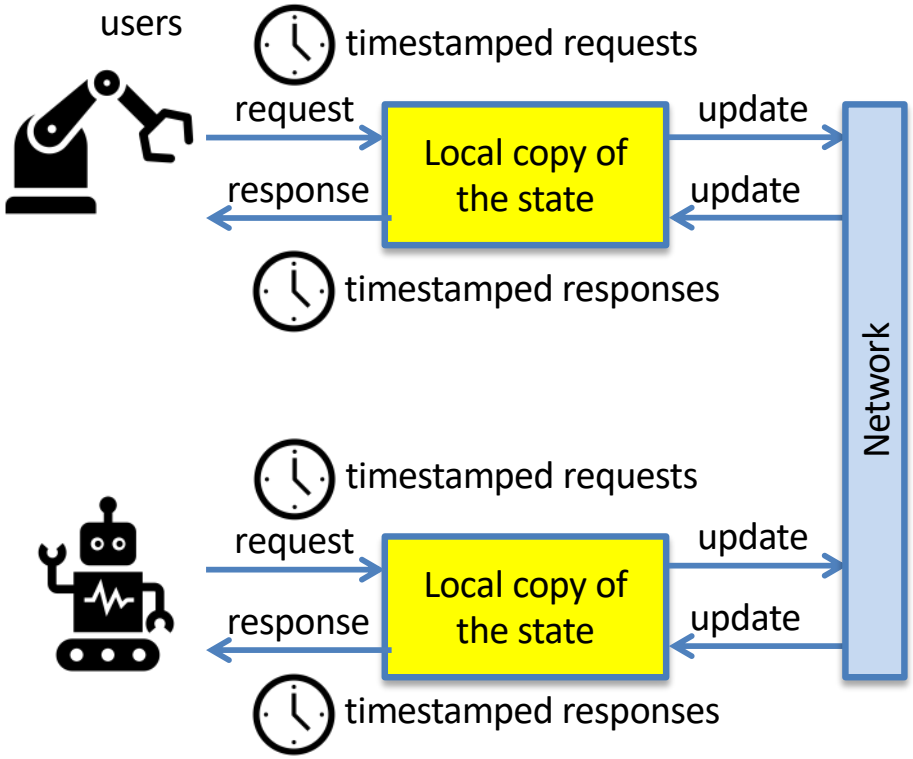
- An **event** occurs instantaneously at a **logical time** *t* in a (superdense) time set.
- Events are *delivered* to components in **logical time order**, regardless of when they are *received.*
- **Physical time** determines whether deadlines are met and messages are received on time and to invoke **fault handlers** if not.

A Deterministic Model

- Given **timestamped inputs**, absent **faults**, events are handled in timestamp order. If event handling is deterministic, then overall **behavior** is deterministic (generated timestamped events are a function of the inputs).

- Faults can occur, of course (in *any* system).
  - The conditions that lead to a fault should be clear.
  - Faults should be detectable so that they can be handled.
  - These requirements are not met by today's solutions.

Principled Decentralized Solution

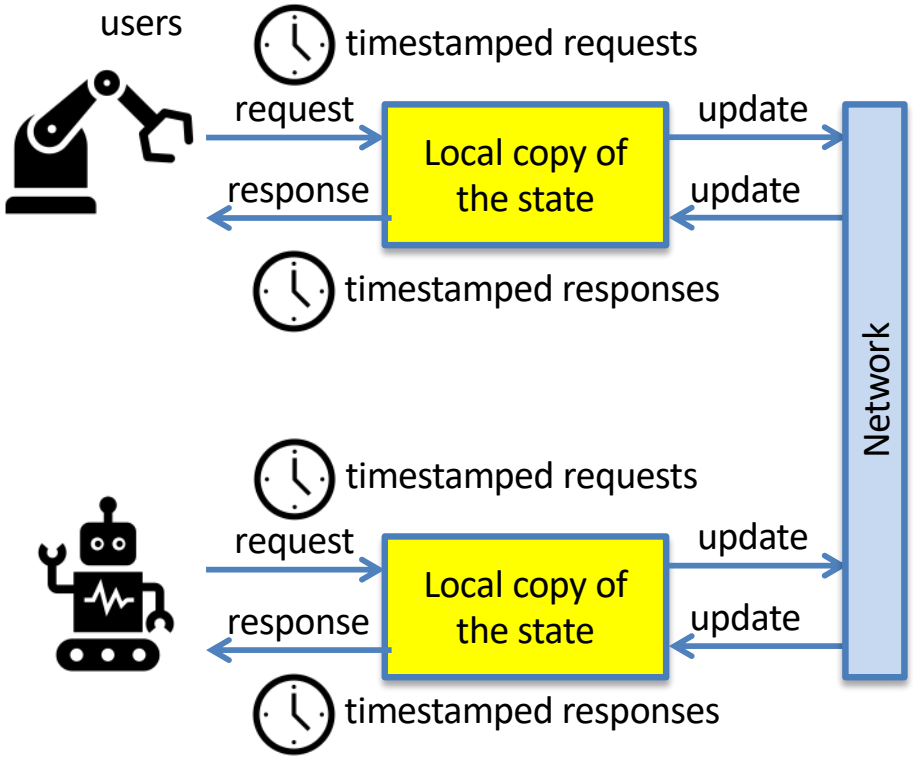


Approach:

- Handle events in timestamp order
- **Consistency**: agreement *at a timestamp*
- **Availability**: quick responses
- **Latency**: cost of communication

Fundamental limit: **CAL Theorem**:
Either **C**onsistency or **A**vailability or both must decrease as **L**atency increases.
[Lee, et al., '21, '23]

Determinism

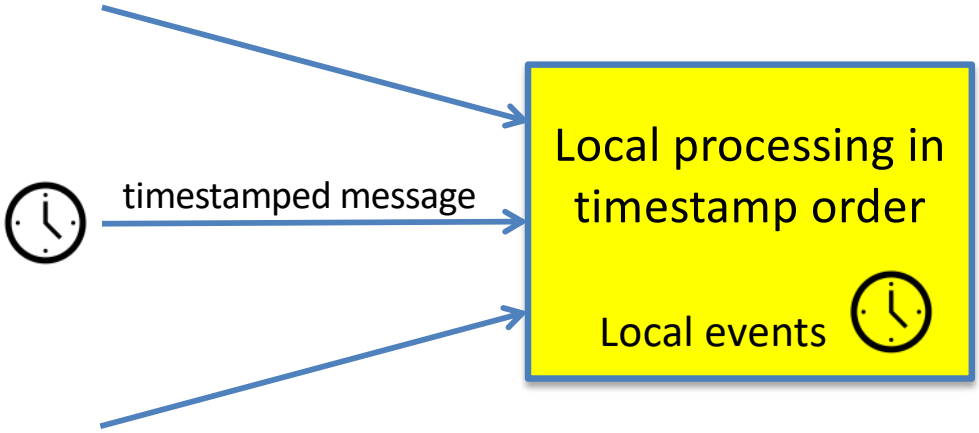


- If you can tolerate unbounded unavailability, consistency can be assured.
- If you bound ***LEX:***
  - network latency ***L,***
  - clock synchronization error ***E***, and
  - execution time ***X***

  then you can bound unavailability while preserving consistency.

Fundamental limit: **CAL Theorem**:
Either **C**onsistency or **A**vailability or both must decrease as **L**atency increases.
[Lee, et al., '21, '23]

The Problem We Solve



When can you process an incoming event and still be reasonably sure you process all events in timestamp order?

The Reactor Model of Computation

Reactors:
- Concurrent
- Deterministic
- Distributed
- Time sensitive

https://reactor-model.org

Coordination languages

Programming APIs

https://lf-lang.org

https://micro-lf.org

https://rosia.dev

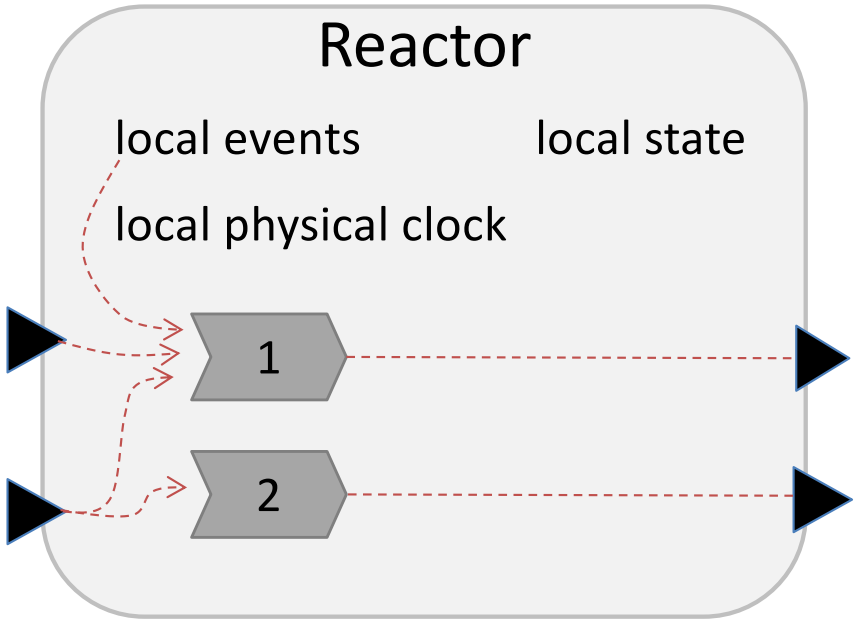
https://xronos.com

The Reactor Model

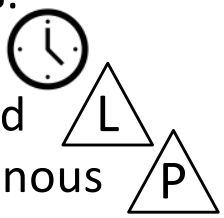
Input ports:
Timestamped events are received in timestamp order at each port.

Reactions:
Triggered by local and/or input events. Manipulate state.

Output ports:
Written to by reactions. Timestamp matches the trigger.



Local events:
- Periodic
- Scheduled
- Asynchronous



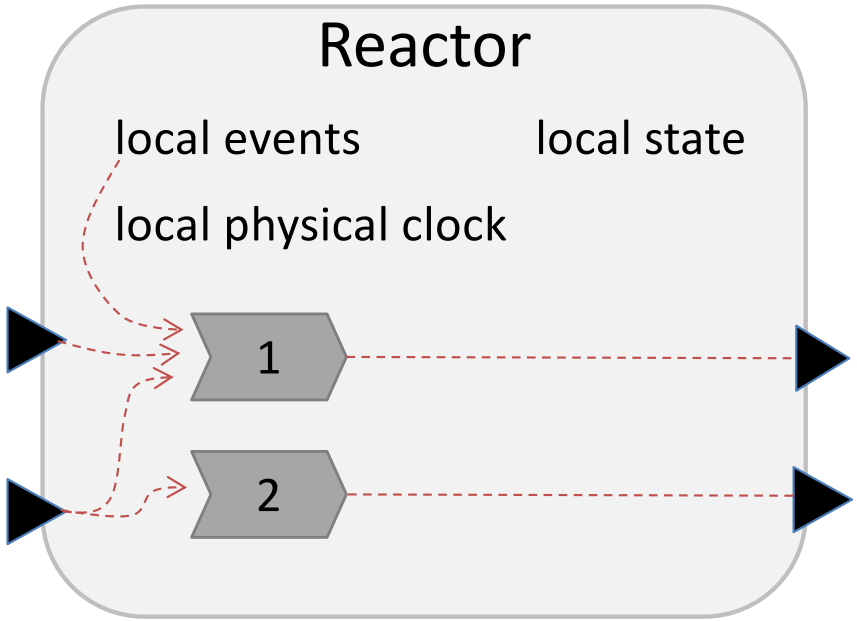
Local physical clock:
Local events are timestamped using this clock.

Goal is to process events in timestamp order.
When events are simultaneous, use reaction order.

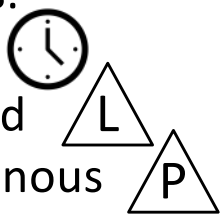
Distributed Execution

Input ports:
Timestamped events are received in timestamp order at each port.

Reactions:
Triggered by local and/or input events. Manipulate state.



Local events:
- Periodic
- Scheduled
- Asynchronous



Local physical clock:
Local events are timestamped using this clock.

Output ports:
Written to by reactions.
Timestamp matches the trigger.

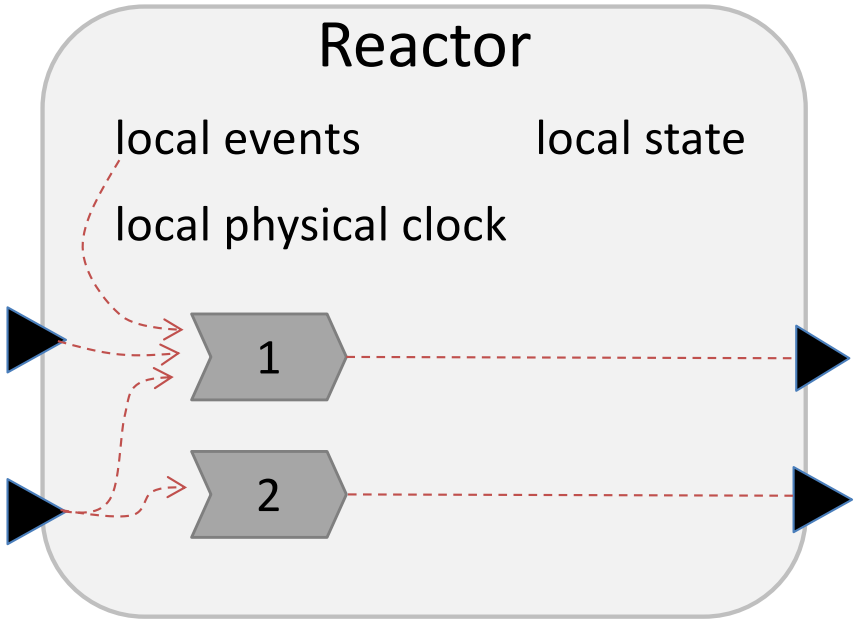
Advance local logical time to the timestamp *t* of the earliest next event. Execute triggered reactions in precedence order.

Logical Time Chases Physical Time

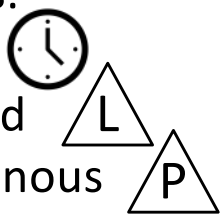
Input ports:
Timestamped events are received in timestamp order at each port.

Reactions:
Triggered by local and/or input events. Manipulate state.

Output ports:
Written to by reactions. Timestamp matches the trigger.



Local events:
- Periodic
- Scheduled
- Asynchronous



Local physical clock:
Local events are timestamped using this clock.

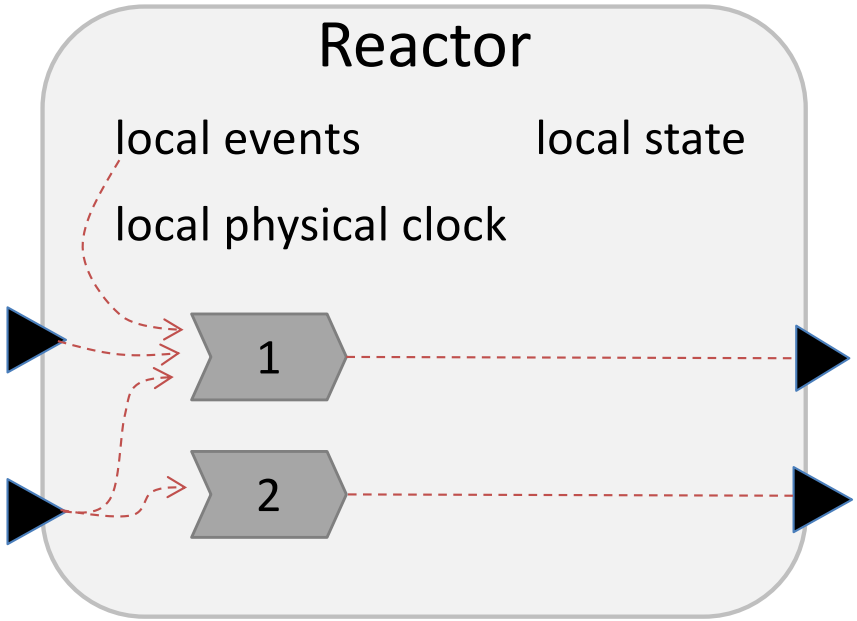
Advance to logical time *t* after local physical time exceeds *t*.

Conservative Decentralized Coordination with In-Order Network Delivery

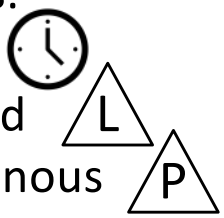
Input ports:
Timestamped events are received in timestamp order at each port.

Reactions:
Triggered by local and/or input events. Manipulate state.

Output ports:
Written to by reactions. Timestamp matches the trigger.



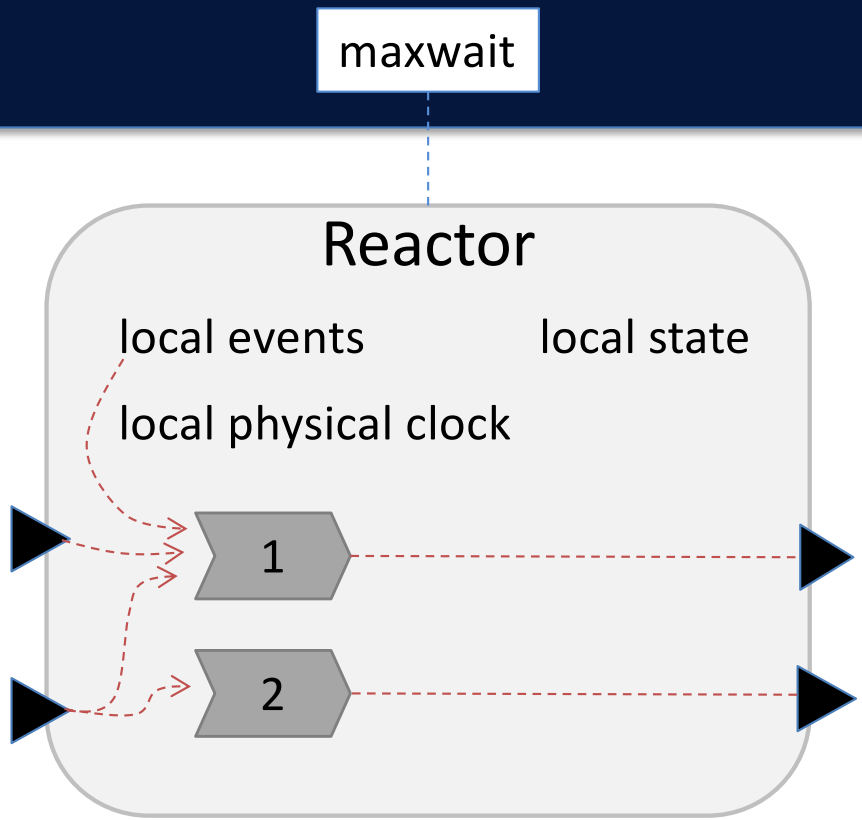
Local events:
- Periodic
- Scheduled
- Asynchronous



Local physical clock:
Local events are timestamped using this clock.

Advance to logical time *t* when local physical time exceeds *t* and inputs have been received on each input port with timestamp *t* or greater.

Maxwait



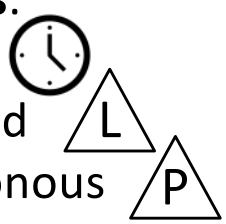
Input ports:
Timestamped events are received in timestamp order at each port.

Reactions:
Triggered by local and/or input events. Manipulate state.

Output ports:
Written to by reactions. Timestamp matches the trigger.

Local events:

- Periodic
- Scheduled
- Asynchronous



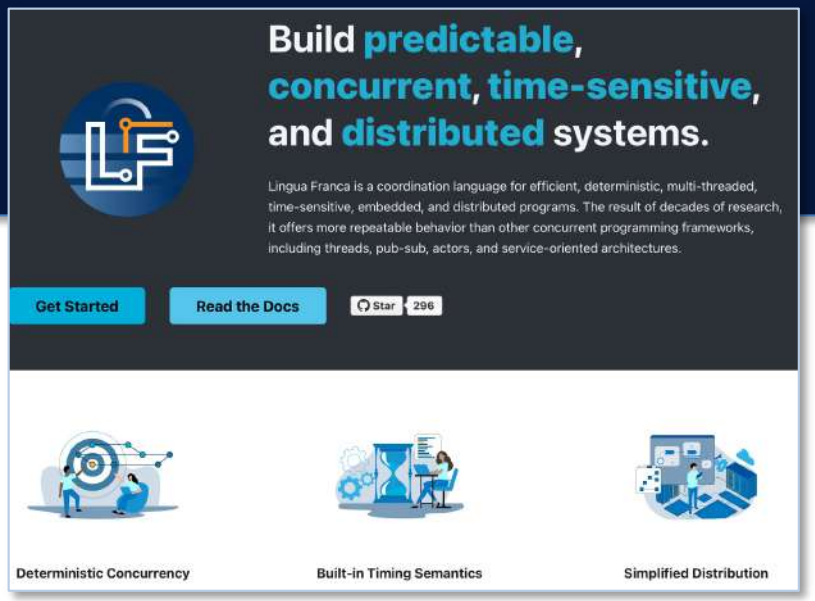
Local physical clock:
Local events are timestamped using this clock.

Advance to logical time *t* when either:

1. **Inputs are known up to *t* and *T* > *t***
2. ***T* > *t* + maxwait.** **(*T* = local physical time)**

Lingua Franca

Concurrent, distributed, timed, & deterministic by default

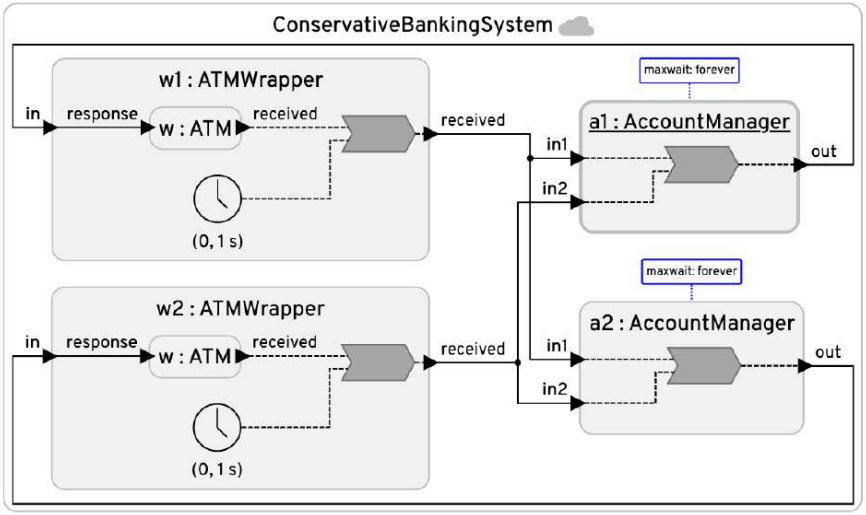


- A polyglot coordination language for high performance, secure, distributed, real-time, safety-critical software.
- An open source tool chain supporting development in C, C++, Python, Rust, and TypeScript: https://github.com/icyphy/lingua-franca
- Shows that time-stamped deterministic discrete-event models can execute efficiently.

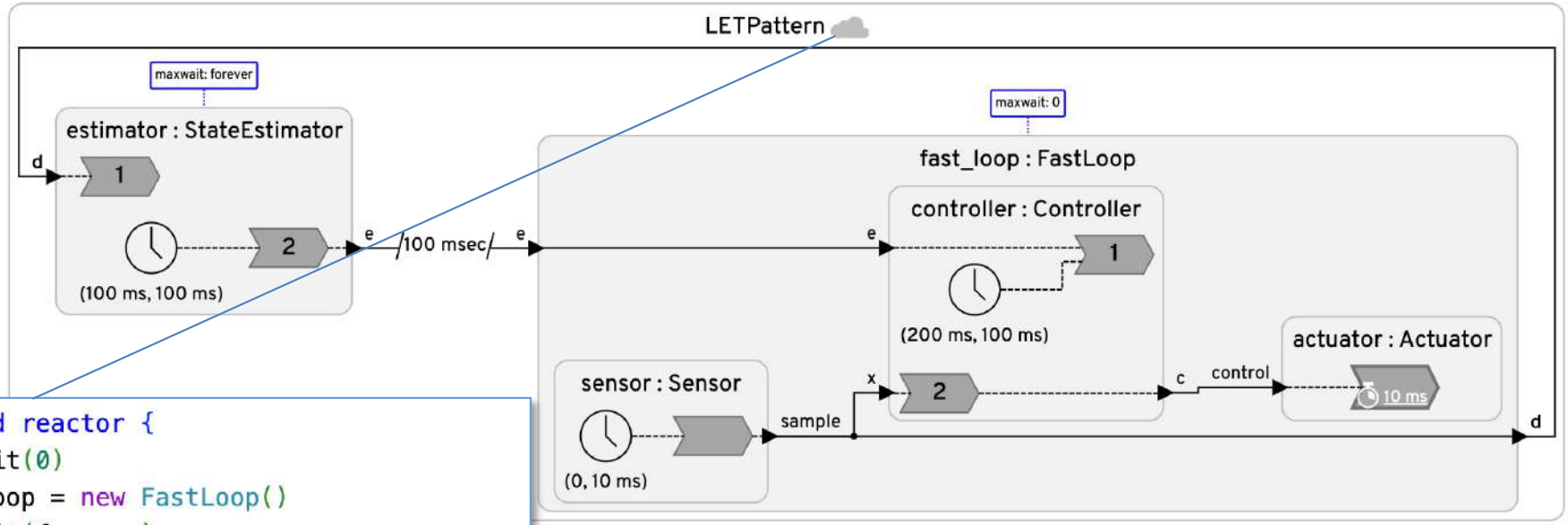
- *Deterministic concurrency*
- Automatic multicore utilization
- Federated, distributed execution
- Real-time scheduling

Visualization of architecture is automatically generated from source.

https://lf-lang.org



Federated Lingua Franca

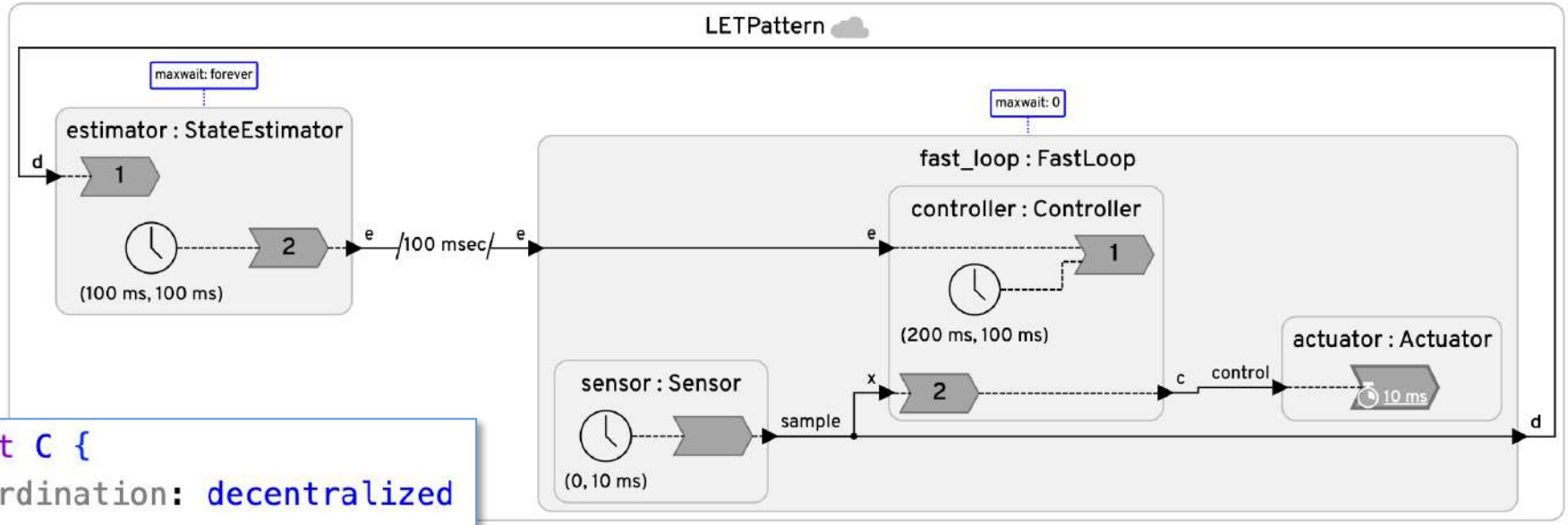



```
federated reactor {
  @maxwait(0)
  fast_loop = new FastLoop()
  @maxwait(forever)
  estimator = new StateEstimator()

  fast_loop.d -> estimator.d
  estimator.e -> fast_loop.e after 100 msec
}
```

A top-level “federated” reactor is code generated into separate programs that can be deployed on the same machine or across machines.

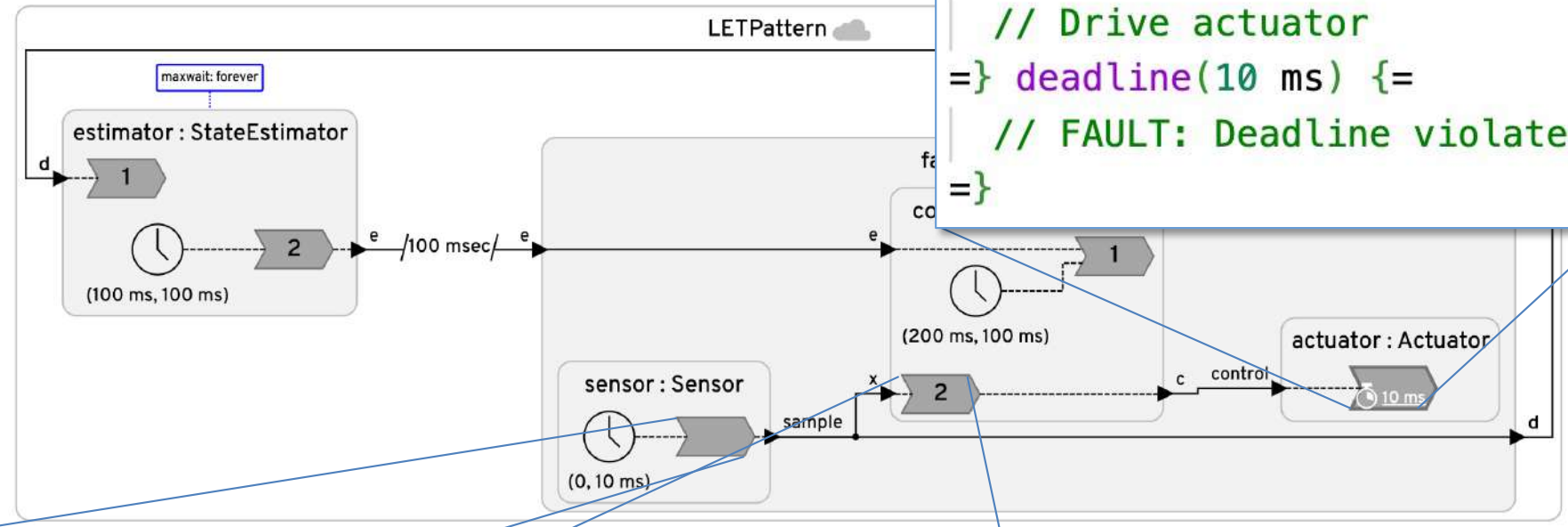
Decentralized Coordination




```
target C {
  coordination: decentralized
}
```

The decentralized coordination option enables trading off consistency and availability and fault detection and handling.

Classic Sense-Control-Actuate Chain



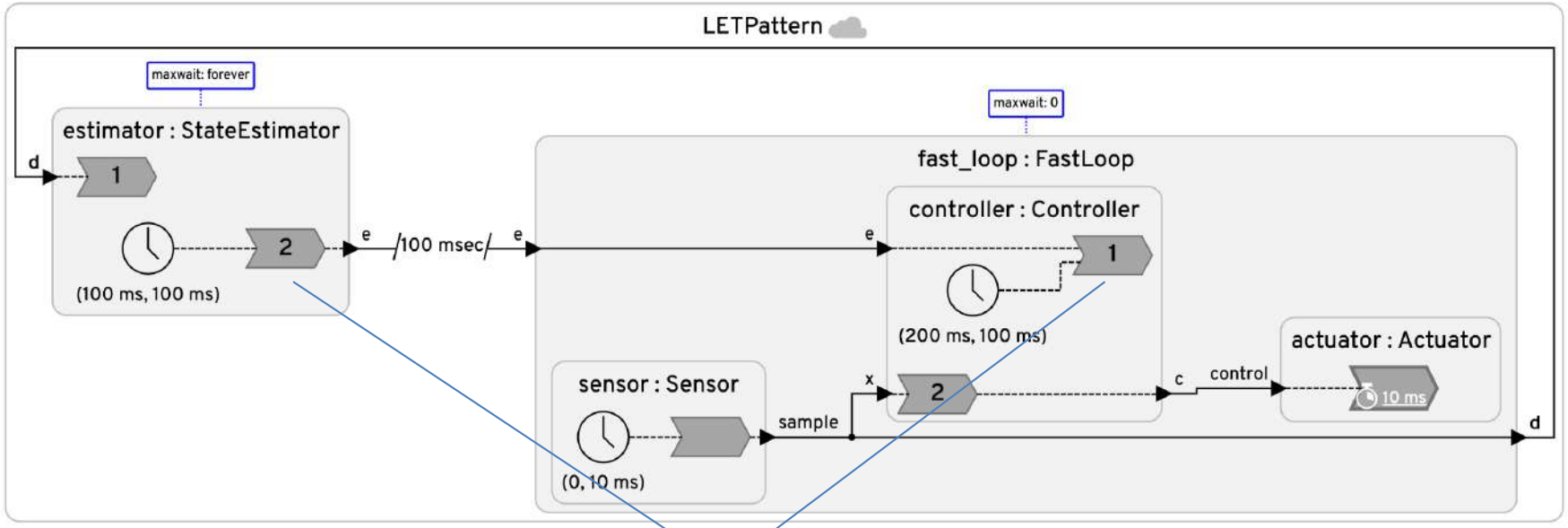

```
reaction(control) {=
  // Drive actuator
=} deadline(10 ms) {=
  // FAULT: Deadline violated.
=}
```

```
reaction(t) -> sample {=
  // Sample the sensor.
=}
```

```
reaction(x) -> c {=
  // Control (e.g. PID)
=}
```

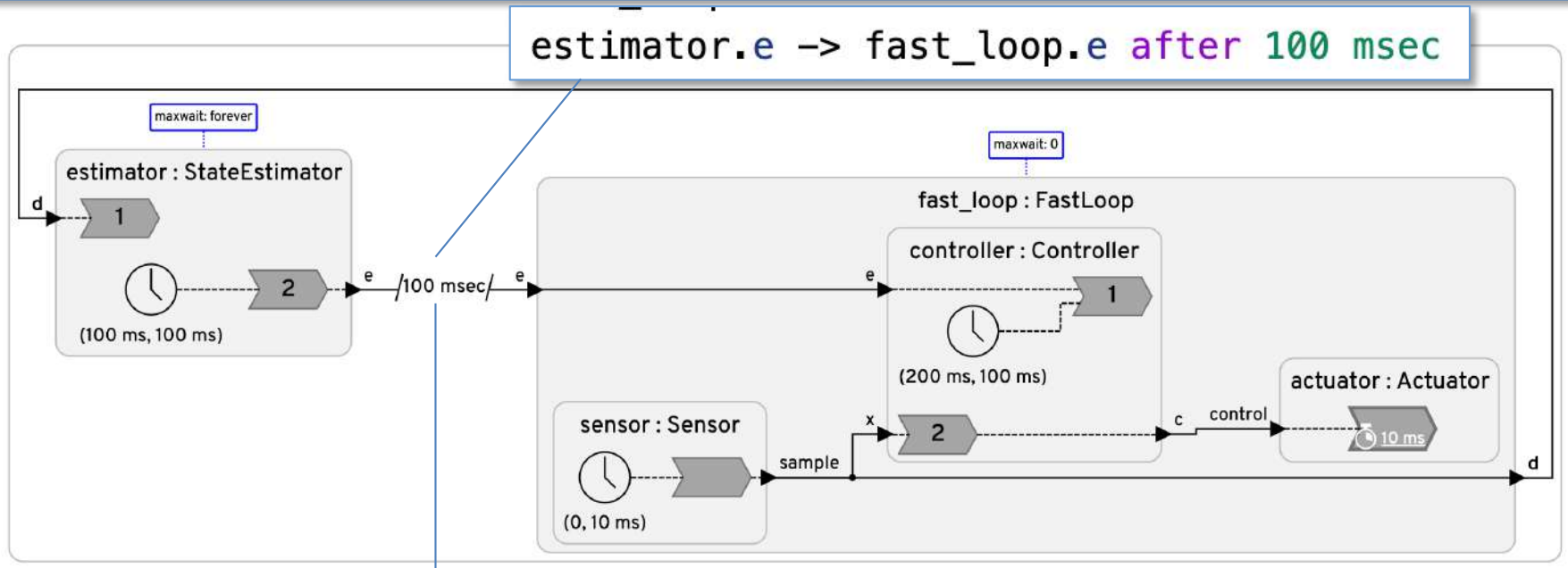
Deadline guides scheduling and provides a handler for late inputs.

Concurrent Tasks



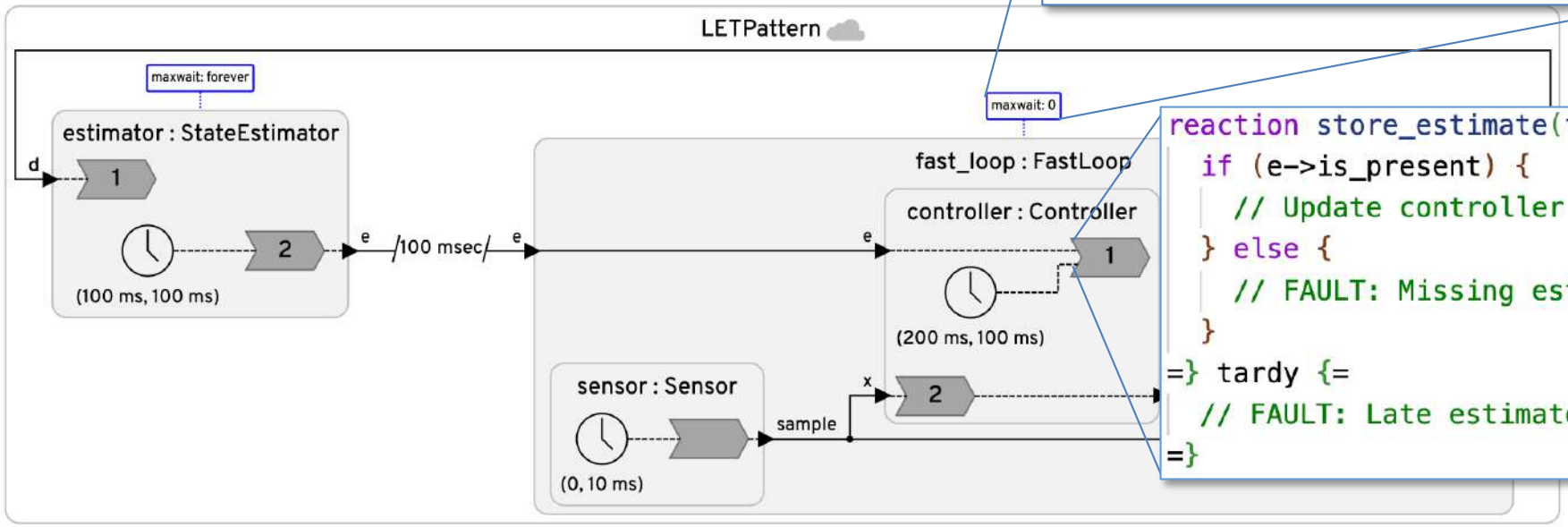
Periodically, use recorded sensor data to estimate the state of the plant and update the controller.

Logical Execution Time (LET) but with Fault Handling



Logical time delay (“after” delay) allows time for computation and communication to complete. It explicitly relaxes consistency to improve availability.

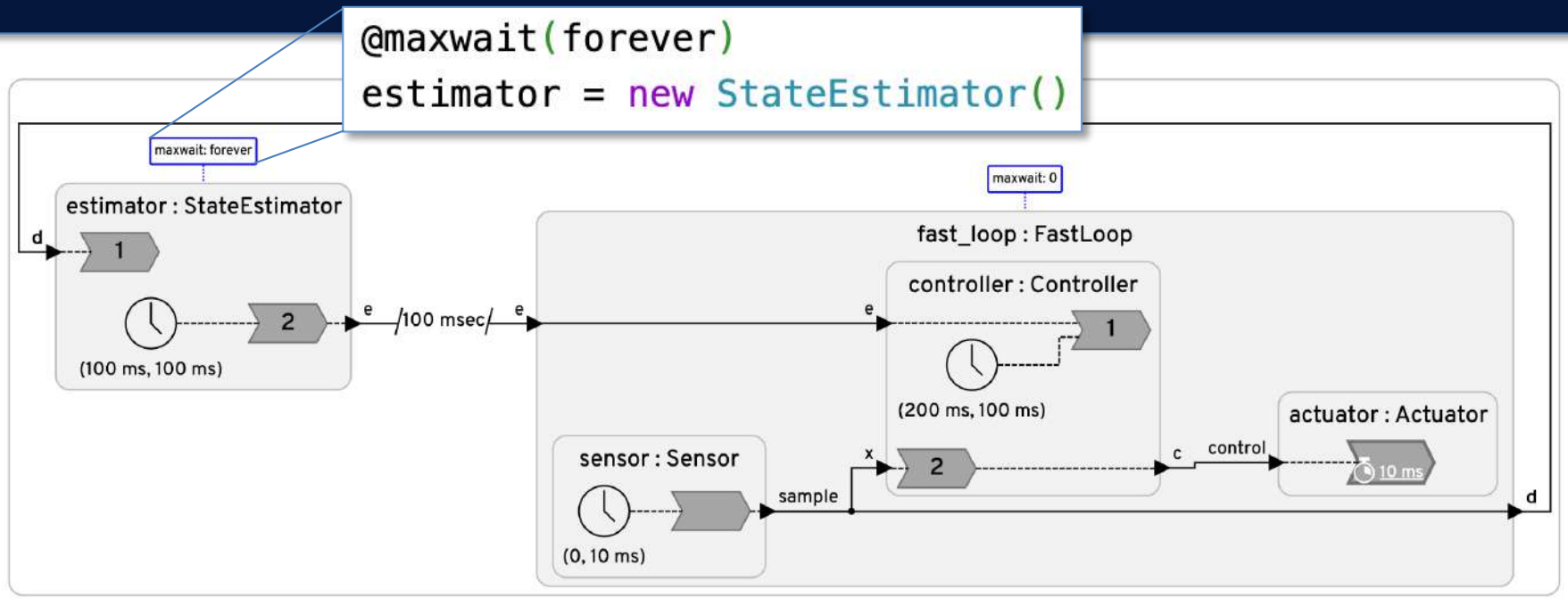
Fast Responses



A maxwait of zero means the federate can react to events without waiting.

- Minimizes sensor-actuator latency.
- Ensures quick detection of missing updates from estimator.

Enforcing Required Data



A maxwait of forever means the federate waits indefinitely for inputs.

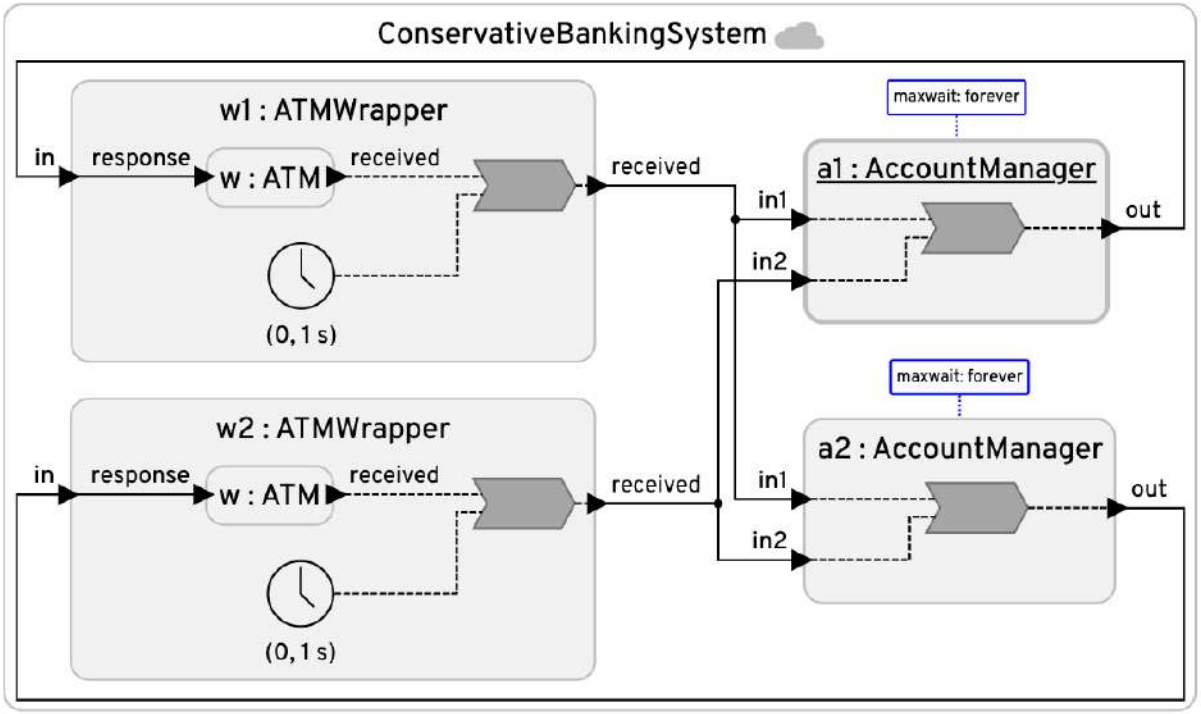
- Input data is not optional.
- Reaction order is assured.

Ex. 2: Consistency: Agreement at a *Timestamp* (not at a time)

```
federated reactor {
  w1 = new ATMWrapper()
  w2 = new ATMWrapper()
  @maxwait(forever)
  a1 = new AccountManager()
  @maxwait(forever)
  a2 = new AccountManager()

  w1.received -> a1.in1
  w2.received -> a2.in2
  w1.received -> a2.in1
  w2.received -> a1.in2

  a1.out -> w1.in
  a2.out -> w2.in
}
```

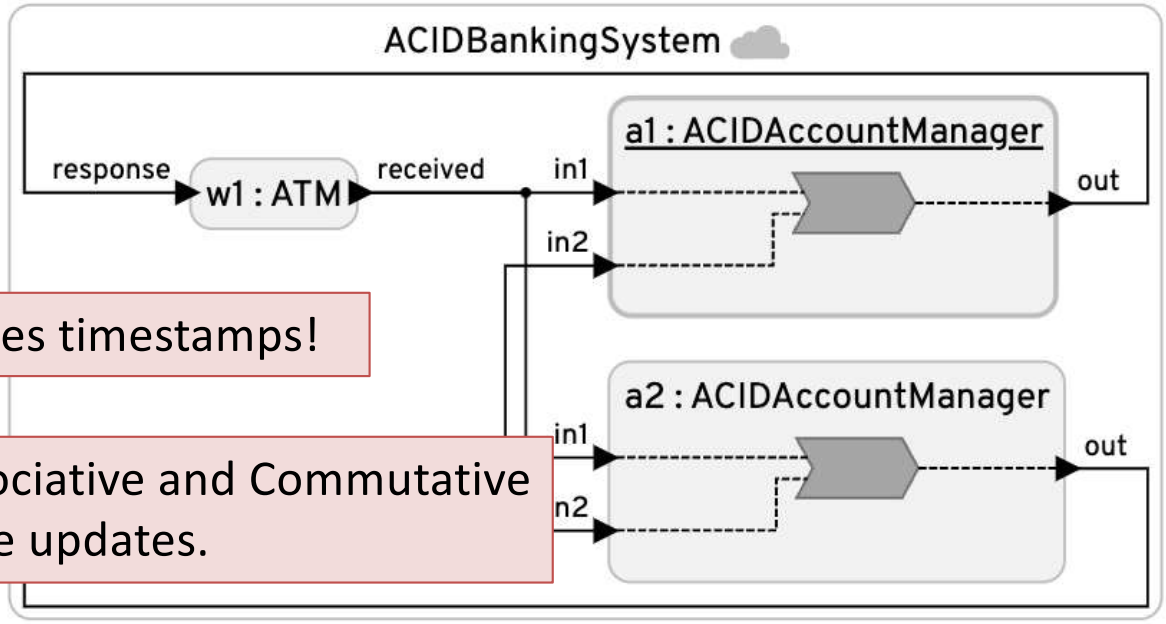


Chandy and Misra [1988] with NULL messages.

Alternative: Bounded maxwait and local timer to detect missing NULL messages (heartbeat).

Eventual Consistency Without Coordination
ACID 2.0 and CRDTs

```
reactor ACIDAccountManager {
  input in1: int
  input in2: int
  output out: int
  state balance: int = 0
  reaction(in1, in2) -> out {=
    if (in1->is_present)
      self->balance += in1->value;
    if (in2->is_present)
      self->balance += in2->value;
    lf_set(out, self->balance);
  =} tardy
}
```



Ignores timestamps!

Associative and Commutative state updates.

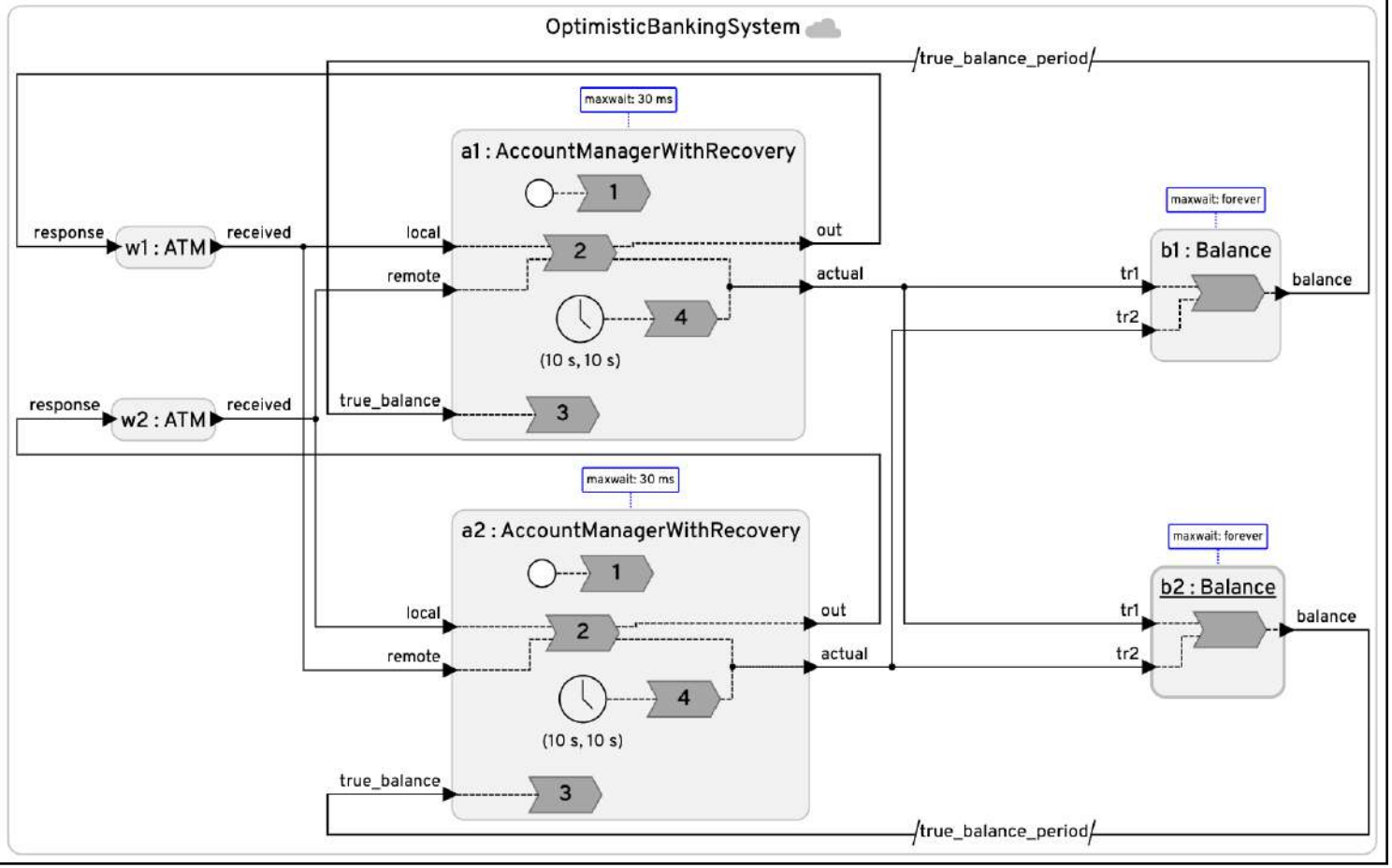
```
@maxwait(0)
a1 = new ACIDAccountManager()
@maxwait(0)
a2 = new ACIDAccountManager()
```

ACID 2.0 [Helland, 2021], CRDTs [Shapiro, et al., 2011], CALM [Hellerstein & Alvaro, 2021]

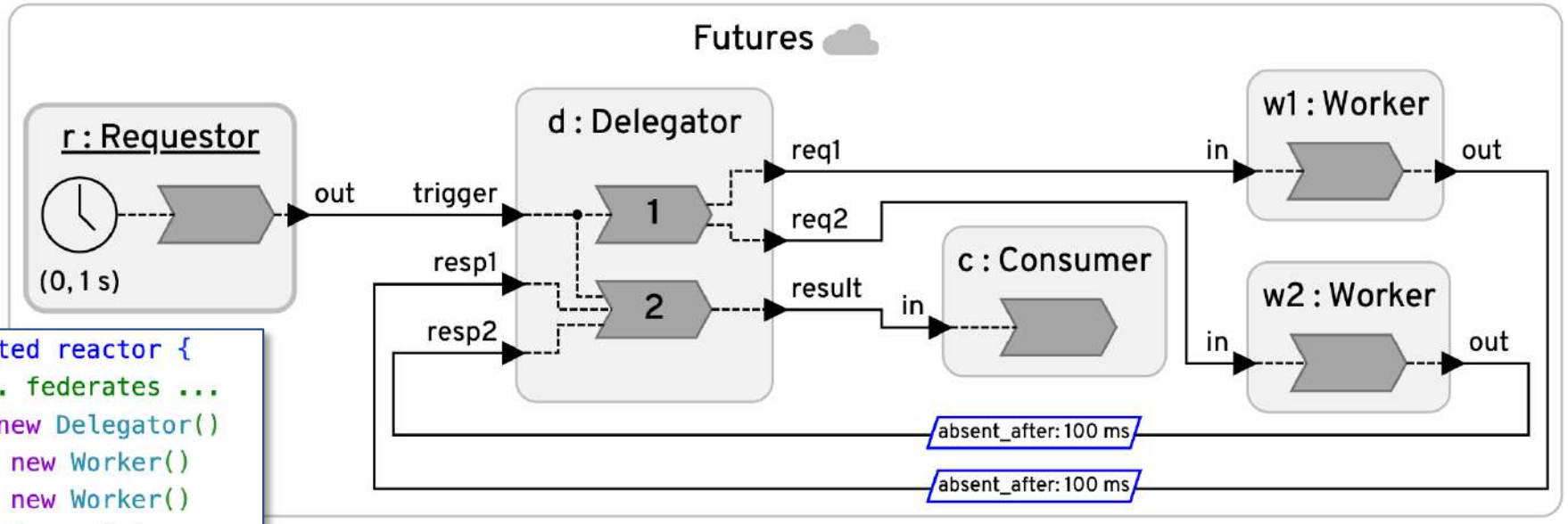
Working the CAL Tradeoff: Optimistic Execution with Rollback

Guarantee <= 30 ms unavailability, taking measured business risk, and ensure eventual consistency (assuming eventual delivery).

TimeWarp [Jefferson, 1985]



Ex. 3: Remote Procedure Calls with Futures, but with Fault Tolerance

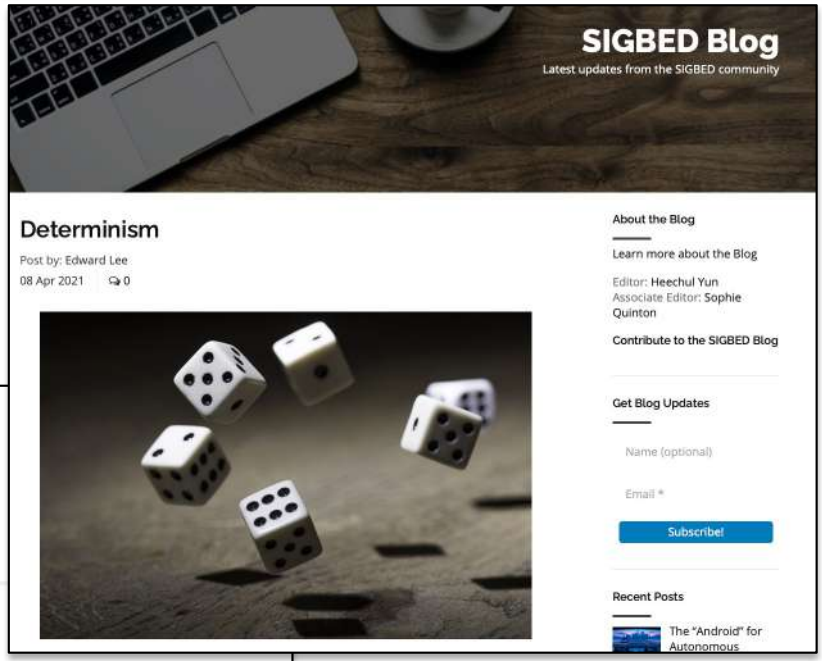



```
federated reactor {
  //... federates ...
  d = new Delegator()
  w1 = new Worker()
  w2 = new Worker()
  d.req1 -> w1.in
  d.req2 -> w2.in
  @absent_after(100 ms)
  w1.out -> d.resp1
  @absent_after(100 ms)
  w2.out -> d.resp2
}
```

Coordination rule #2:
After a federate advances to timestamp *t*, it waits an additional *absent_after* time before assuming the destination inputs are absent.

Finally: Some Comments about Determinism as a Property of ***Models***

A ***model*** is *deterministic* if, given the initial *state* and the *inputs*, the model defines exactly one *behavior*.



RESEARCH-ARTICLE OPEN ACCESS

Determinism

Author: Edward A. Lee Authors Info & Affiliations

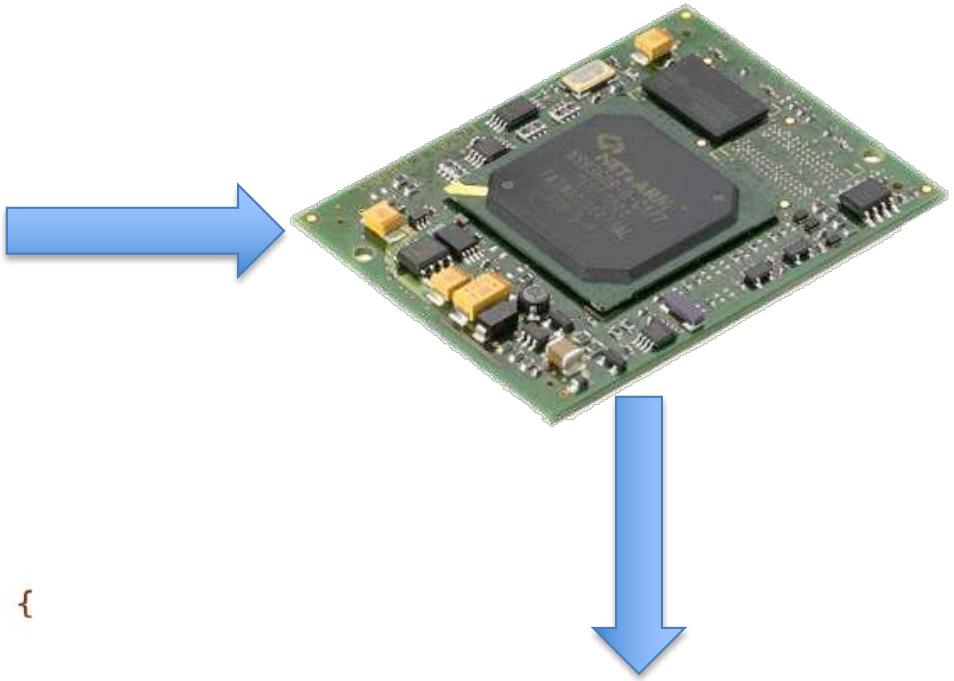
ACM Transactions on Embedded Computing Systems, Volume 20, Issue 5 • July 2021 • Article No.: 38, pp 1–34 • https://doi.org/10.1145/3453652

A Program is a Model

```c
// Initial state:
int sum = 0;
int a = 0;
int b = 1;
int count = 1;

// Input:
int stop;
printf("Input: ");
scanf("%d", &stop);

// Behavior:
printf("Output: ");
while (count++ <= stop) {
  printf("%d ", sum);
  a = b;
  b = sum;
  sum = a + b;
}
printf("\n");
```



```
Input: 10
Output: 0 1 1 2 3 5 8 13 21 34
```

But since the world is nondeterministic (sh** happens), shouldn't we just accept nondeterminism and design systems to tolerate it?

The Value of Models

- In *science*, the value of a *model* lies in how well its behavior matches that of the physical system.

- In *engineering*, the value of the *physical system* lies in how well its behavior matches that of the model.

A scientist asks, "Can I make a model for this thing?"
An engineer asks, "Can I make a thing for this model?"

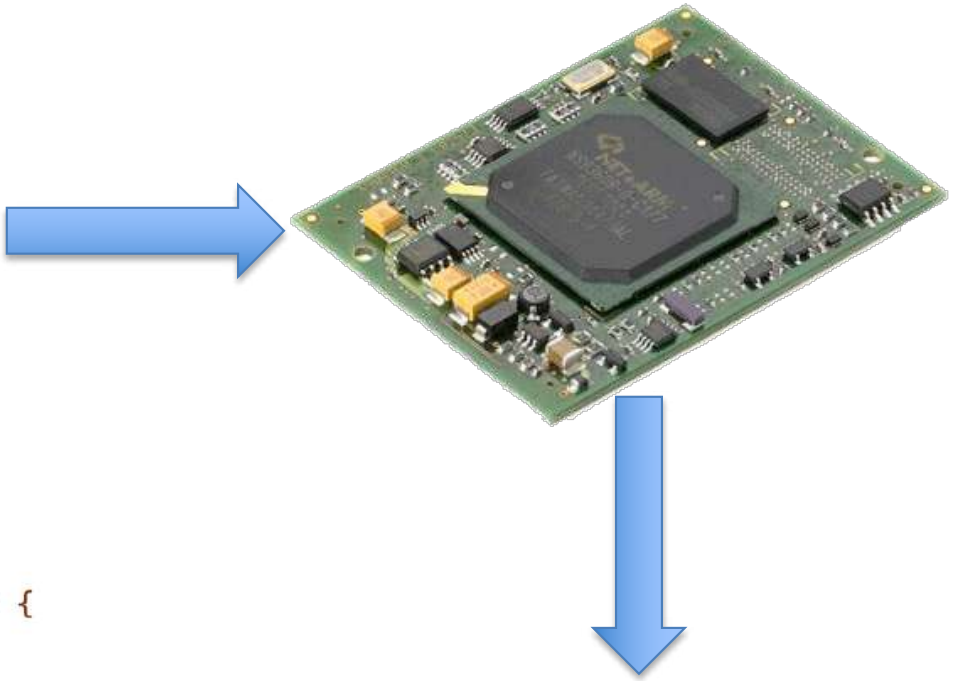
A scientist says, "All models are wrong but some are useful."
An engineer says, "All implementations are wrong but some are useful"

A Program is an *Engineering* Model

```c
// Initial state:
int sum = 0;
int a = 0;
int b = 1;
int count = 1;

// Input:
int stop;
printf("Input: ");
scanf("%d", &stop);

// Behavior:
printf("Output: ");
while (count++ <= stop) {
  printf("%d ", sum);
  a = b;
  b = sum;
  sum = a + b;
}
printf("\n");
```



```
Input: 10
Output: 0 1 1 2 3 5 8 13 21 34
```

Asking the Right Question

The question is *not* whether deterministic models can describe the behavior of cyber-physical systems (with high fidelity).

The question is whether we can build cyber-physical systems whose behavior matches that of a deterministic model (with high probability).

Advantages of Deterministic Models

- More predictable.
  - Known inputs => known behavior.
- Testing is more useful.
  - Known inputs => known outputs.
- Analysis is more tractable.
  - Math: Boolean algebra, calculus, etc.
- Simulation is more useful.
  - One input yields one trace.
- Verification scales better.
  - Much smaller state space.
- Detecting faults is easier.
  - Deviations from known behavior.
- More certifiable?

Advantages of Nondeterministic Models

- Abstraction
  - Abstractions may be easier to understand.
- Uncertainty
  - Model something not fully understood
- Deferred Design Decisions
  - Uncertain specification
- Security
  - Unpredictability can be good.
- Don’t Care
  - Many behaviors are OK for the same input.
- Surprising Behavior
  - Boring is bad.

Determinism?
What about resilience? Adaptability?

Deterministic models do not eliminate the need for robust, fault-tolerant designs.

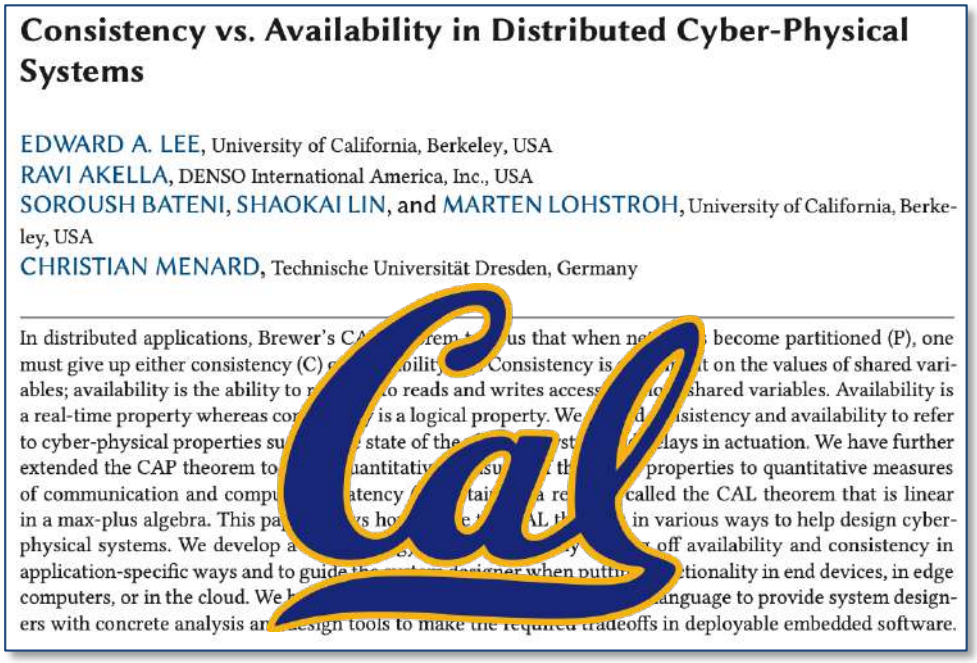
In fact, they *enable* such designs, because they make it much clearer what it means to have a fault!

Fault handlers are an important part of any design.

Fundamental Limits
Consistency Requires Determinism

Consistency fundamentally comes with a cost in **Availability**, where the cost is proportional to apparent **Latency** ($L + E + X$).

Reactors with maxwait enable explicitly managing this tradeoff and detecting faults early.



Consistency vs. Availability in Distributed Cyber-Physical Systems

EDWARD A. LEE, University of California, Berkeley, USA
RAVI AKELLA, DENSO International America, Inc., USA
SOROUSH BATENI, SHAOKAI LIN, and MARTEN LOHSTROH, University of California, Berkeley, USA
CHRISTIAN MENARD, Technische Universität Dresden, Germany

Lee, E. A., et al. (2023). "Consistency vs. Availability in Distributed Cyber-Physical Systems." ACM Transactions on Embedded Computing Systems (TECS) **22**(5s): 1-24.

Implementations of the Reactor Model

Reactors:
- Concurrent
- Deterministic
- Distributed
- Time sensitive

https://reactor-model.org

Coordination languages

https://lf-lang.org

https://micro-lf.org

Programming APIs

https://rosia.dev

https://xronos.com

One Program, One Behavior

```
reaction(in) Vzc, azf, Vzf, qf -> delta_ec {=
    double Ts_K2 = ((double)self->period)/1e9;
    double y = self->K2_intVz * self->integrator
            + self->K2_Vz * Vzf->value
            + self->K2_q * qf->value
            + self->K2_az * azf->value + delta_e_eq;
    self->integrator += Ts_K2 * (Vzc->value - Vzf->value);
    lf_set(delta_ec, y);
=}
```

Behavior is defined and identical across targets, absent faults.

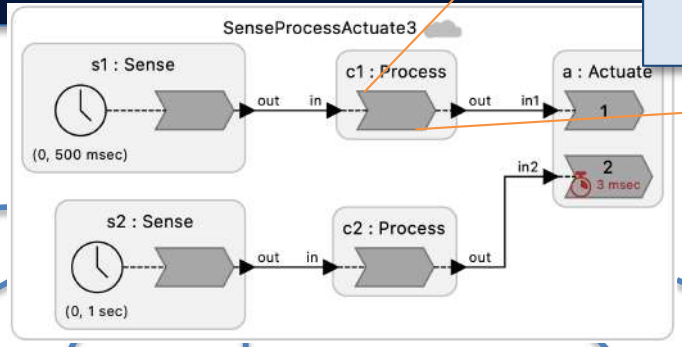


Diagram generated & updated as you write code.

Bare Metal

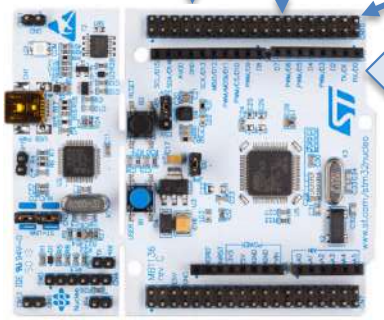
RIOT

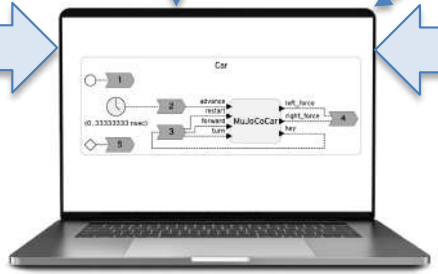
Zephyr

Microsoft Windows

Mac OS

Linux





Embedded processors

C, C++, Python, Rust, TypeScript

VMs on cloud servers

Major Contributors

- Soroush Bateni
  - UT Dallas, USA (LF)
- Jeronimo Castrillon
  - TU Dresden
- Jian-Jia Chen
  - Aachen, Germany (LF)
- Soren Domrös
  - Kiel Univ, Germany (LF)
- Peter Donovan
  - UC Berkeley, USA (LF, Xronos)
- Clément Fournier
  - TU Dresden, Germany (LF)
- Sebastiano Gaiardelli
  - U Verona/TU Munich, Italy/ Germany (micro-LF)
- Erling Jellum
  - NTNU/Berkeley, Norway/USA (LF, micro-LF)
- Dongha Kim
  - Arizona State Univ, USA (LF)
- Hokeun Kim
  - Arizona State Univ, USA (LF)
- Chadlia Jerad
  - Univ of Manouba, Tunisia (LF)
- Byeonggil Jun
  - Arizona State Univ, USA (LF)
- Edward A. Lee
  - UC Berkeley, USA (LF, micro-LF, Rosia)
- Shulu Li
  - UC Berkeley, USA (Rosia)
- Shaokai Lin
  - UC Berkeley, USA (LF)
- Marten Lohstroh
  - UC Berkeley/Xronos Inc., USA (LF, Xronos)
- Magnus Maehlum
  - NTNU, Norway (LF, micro-LF)
- Christian Menard
  - TU Dresden/Xronos Inc., Germany (LF, Xronos)
- Francesco Paladino
  - UC Berkeley, USA (LF, micro-LF)
- Anirudh Rengarajan
  - UC Berkeley, USA (LF)
- Lasse Rosenow
  - HAW Hamberg, Germany (LF)
- Alexander Schulz-Rosengarten
  - Kiel Univ, Germany (LF)
- Tassilo Tanneberger
  - U Dresden, Germany (micro-LF)
- Harun Teper
  - TU Dortmund, Germany (LF)
- Reinhard von Hanxleden
  - Kiel Univ, Germany (LF)
- Steven Wong
  - UC Berkeley, USA (LF)

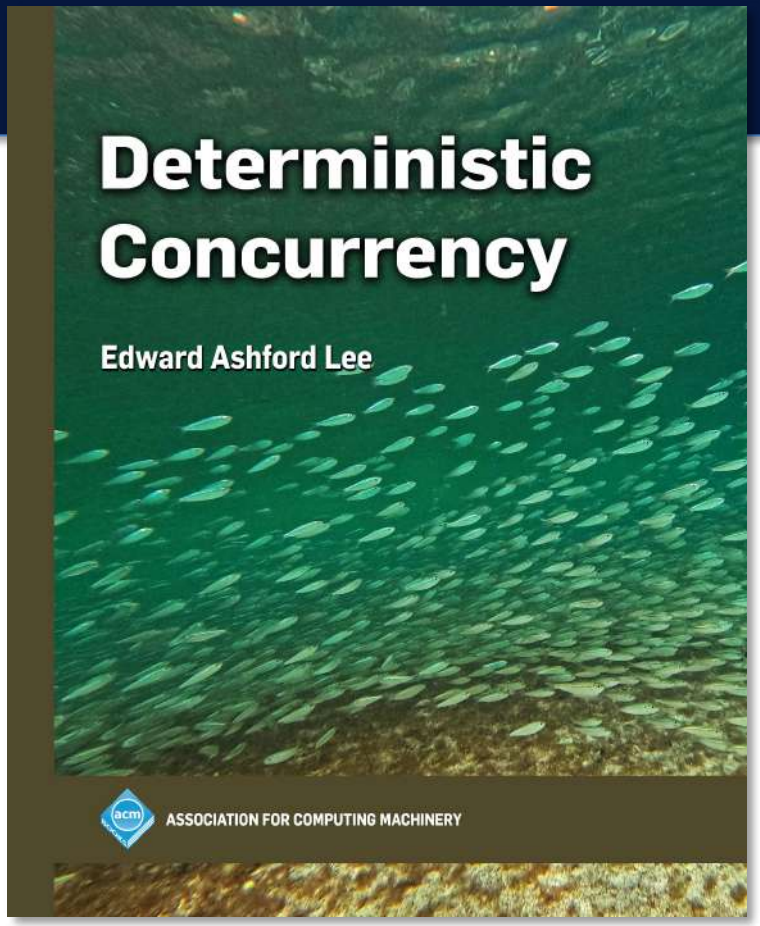
Conclusion

Distributed real-time computing practice today admits far more nondeterminism than necessary (e.g. using ROS 2) and can benefit enormously from more deterministic mechanisms.

But all designs involve tradeoffs.

The CAL tradeoff says that as $L + E + X$ (latency, clock sync error, execution time) increases, every application must pay a price in either Consistency or Availability.

Work the tradeoff!



Book in progress, expected in Spring 2027.