iCyPhy

A Generalized Mechanism for Distributed Time-Sensitive Systems

Edward A. Lee

Professor of the Graduate School
Distinguished Professor Emeritus

Invited Talk

Real-time And intelliGent Edge computing workshop (RAGE) (at CPS-IoT Week)

May 11th, 2026, Saint Malo, France

University of California at Berkeley

A Simple Challenge Problem

A software component on a microprocessor in an aircraft door provides two network services:

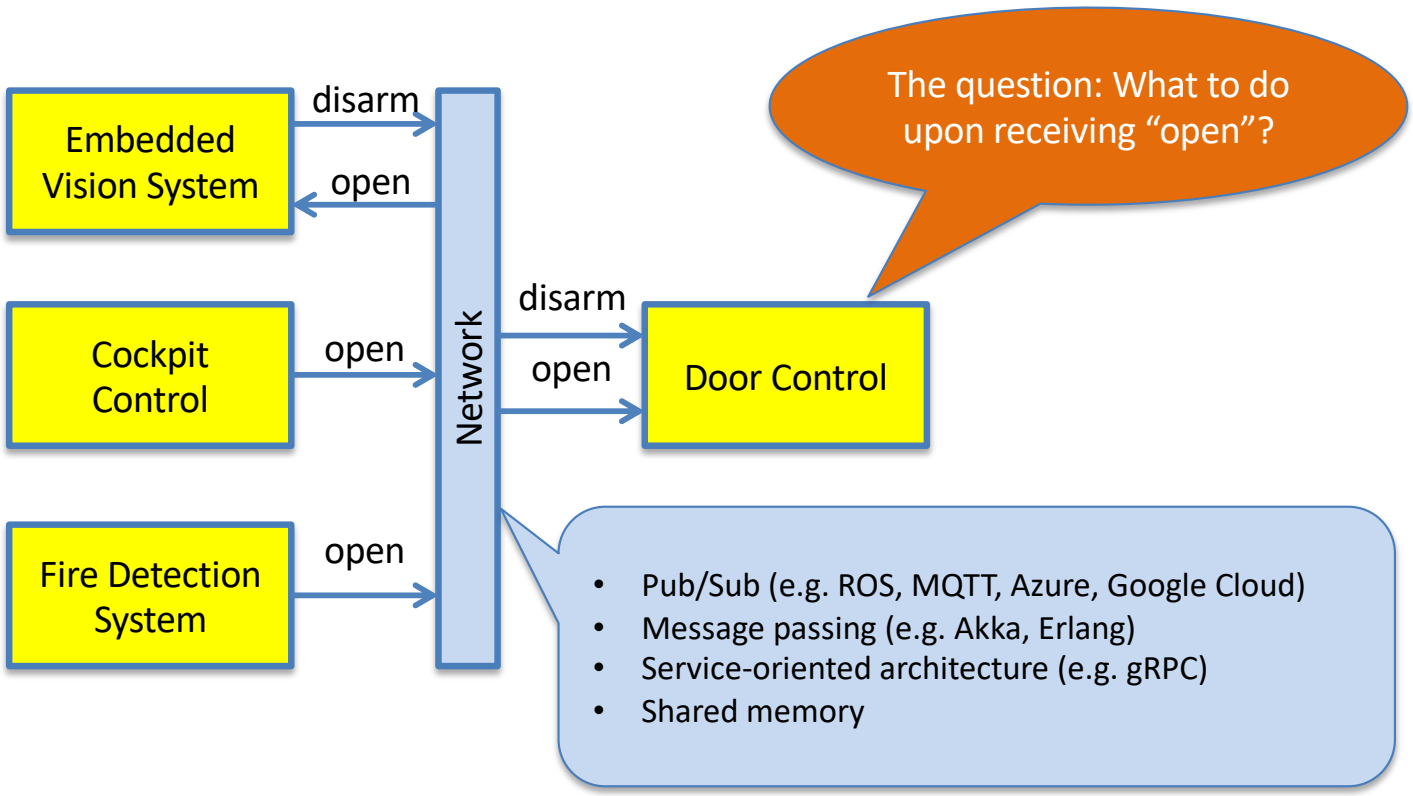
1. “open”
2. “disarm”

Assume state is closed and armed.

What should it do when it receives a request “open”?

Image by Christopher Doyle from Horley, United Kingdom - A321 Exit Door, CC BY-SA 2.0

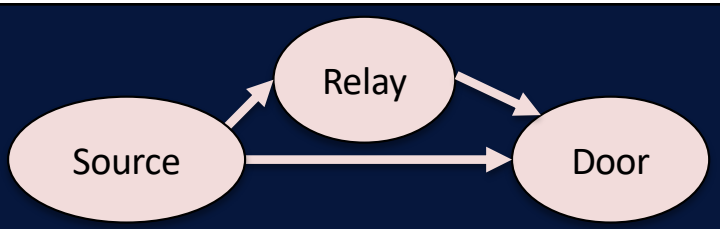
Possible Architectures



When order matters, even small discrepancies can be fatal

Image from *The Telegraph*, Sept. 9, 2015

Naïve Aircraft Door Using Actors or Pub-Sub




```
Actor Source {
     handler main(){
          x = new Door();
          r = new Relay();
          r.check();
          x.open_door();
     }
}
```

The application designer now has to figure out how develop a distributed consensus.

```
Actor Relay {
     handler check(Door x){
          x.disarm_door();
     }
}
```

```
Actor Door {
     handler open_door(){
          …
     }
     handler disarm_door(){
          …
     }
}
```

The Reactor Model of Computation

Reactors:
- Concurrent
- Deterministic
- Distributed
- Time sensitive

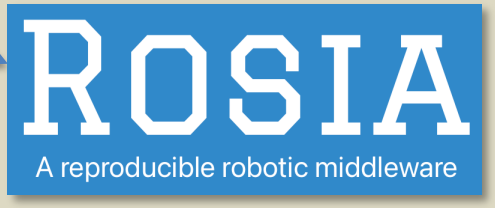
https://reactor-model.org

Coordination languages

Programming APIs

https://lf-lang.org

https://lf-lang.org/reactor-uc



https://rosia.dev

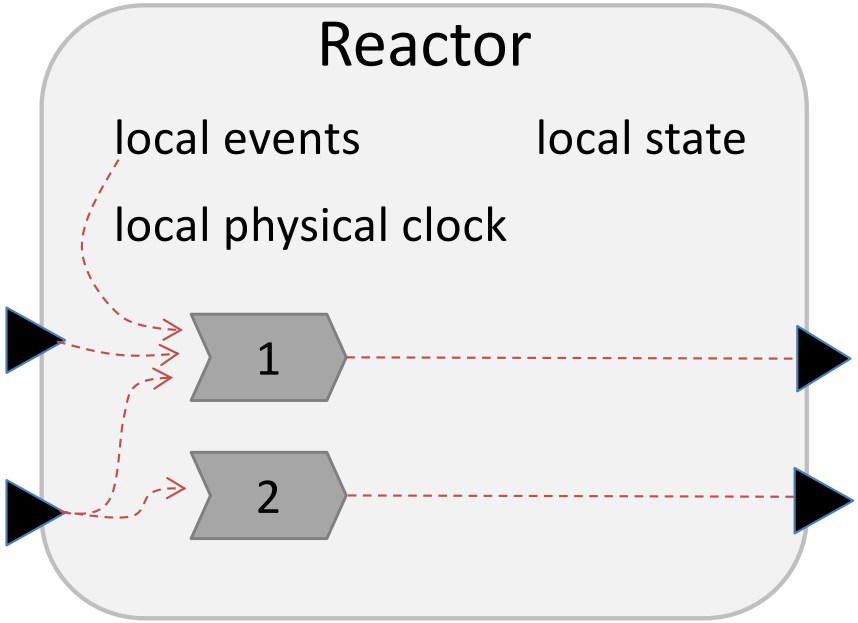
https://xronos.com

The Reactor Model

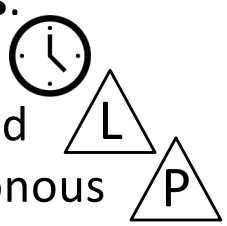
Input ports:
Timestamped events are received in timestamp order at each port.

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



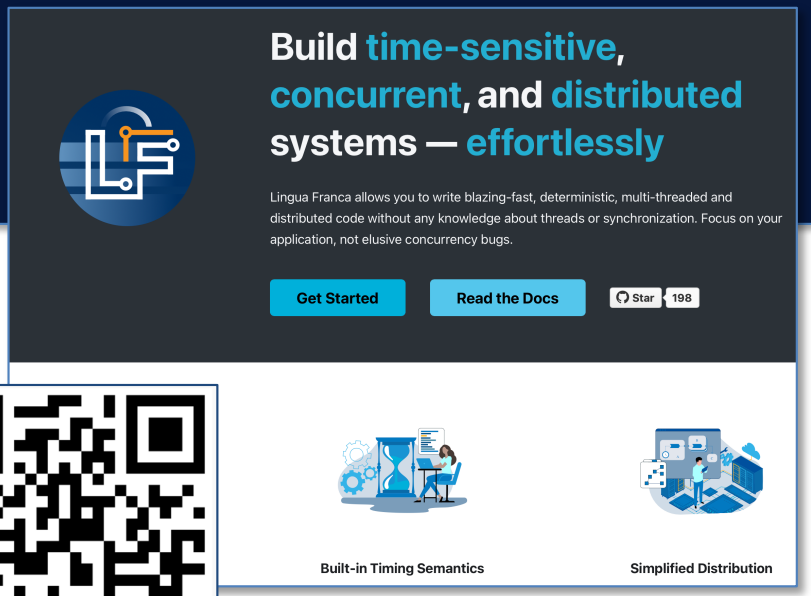
Local physical clock:
Timestamps of local events align with this clock.

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

Lingua Franca

Concurrent, distributed, timed, & deterministic by default

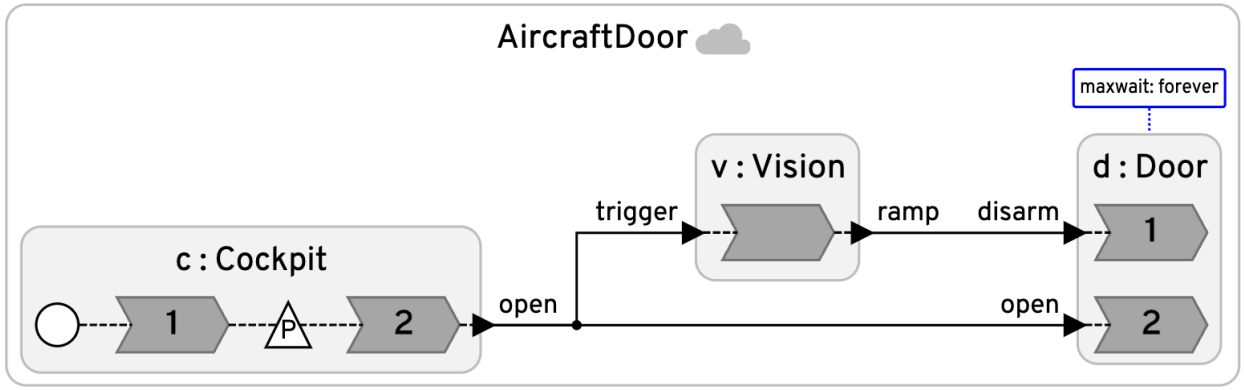
- A polyglot coordination language for high performance, secure, distributed, real-time, safety-critical software.
- An open source tool chain supporting development in C, C++, Python, Rust, and TypeScript: https://github.com/icyphy/lingua-franca
- Shows that time-stamped deterministic discrete-event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



https://lf-lang.org

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

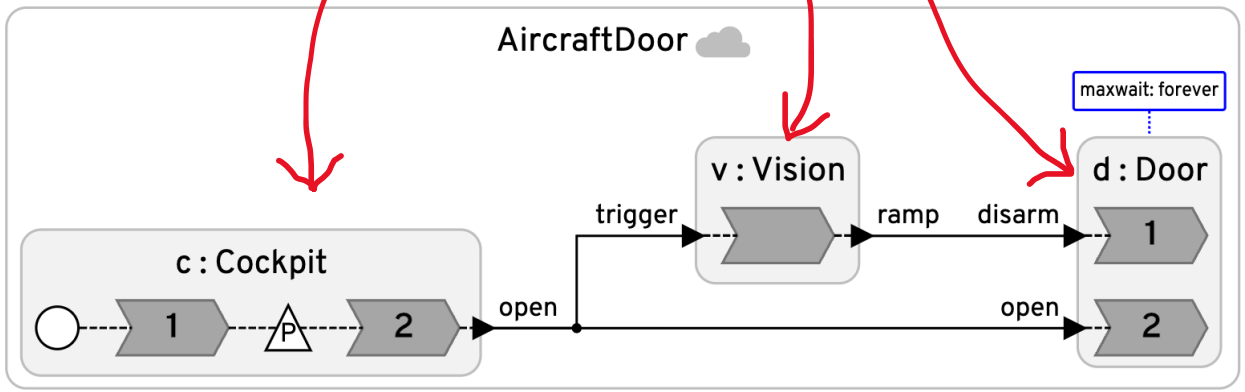
Visualization of architecture is automatically generated from source.



Federated Programs in Lingua Franca

```
federated reactor {
  c = new Cockpit()
  v = new Vision()
  @maxwait(forever)
  d = new Door()

  c.open -> d.open
  c.open -> v.trigger
  v.ramp -> d.disarm
}
```

Federates execute in separate processes, possibly on different machines.

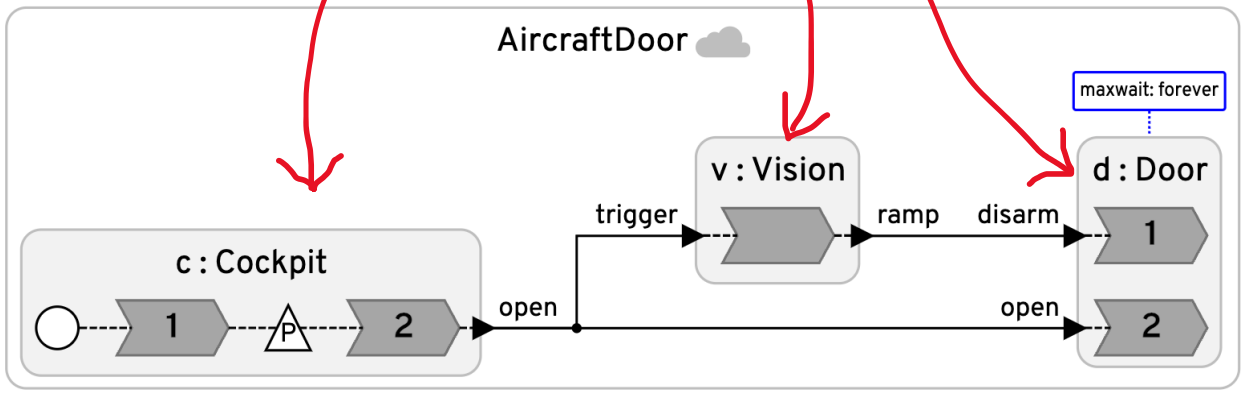


Federated Programs

```
federated reactor {
  c = new Cockpit()
  v = new Vision()
  @maxwait(forever)
  d = new Door()

  c.open -> d.open
  c.open -> v.trigger
  v.ramp -> d.disarm
}
```

A Runtime Infrastructure (RTI) helps coordinate the federates:

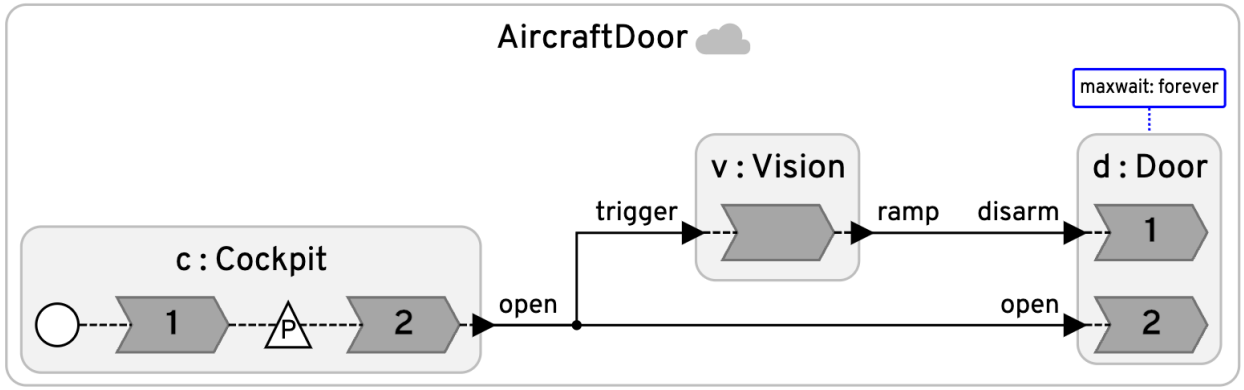
- Provides software clock synchronization (turn this off if you use TSN)
- Coordinates starting logical time
- Coordinates arrivals and departures of federates



Two RTI Coordination Modes: Centralized and Decentralized

```
target C {
  coordination: decentralized
}
federated reactor {
  c = new Cockpit()
  v = new Vision()
  @maxwait(forever)
  d = new Door()

  c.open -> d.open
  c.open -> v.trigger
  v.ramp -> d.disarm
}
```

Today: Focus on Decentralized coordination.
The RTI does not get involved during runtime.

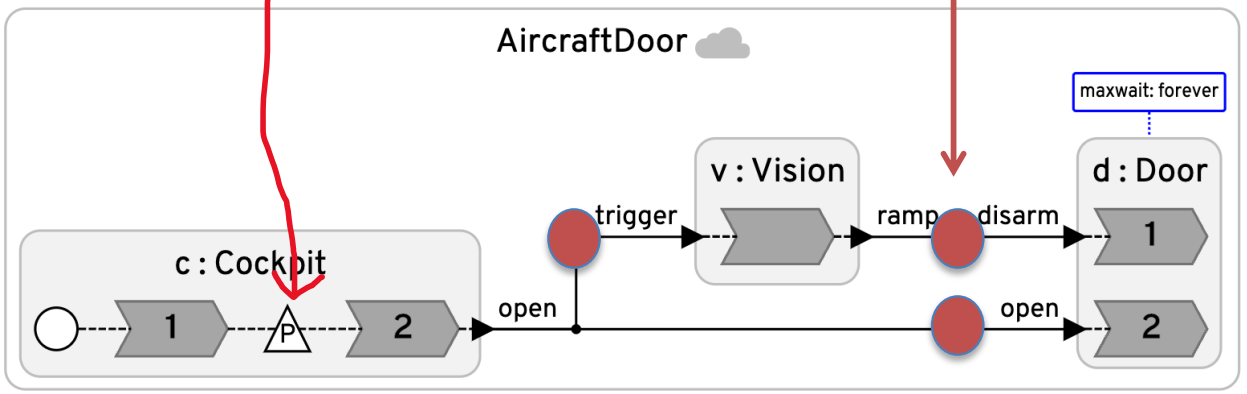


Handling Messages in Timestamp Order

```
reactor Cockpit {
  physical action command
  output open: bool

  reaction(startup) -> command {=
    // Set up cockpit UI controls.
  =}

  reaction(command) -> open {=
    // Send open message.
    lf_set(open, true);
  =}
}
```

Timestamp assigned using local physical clock.

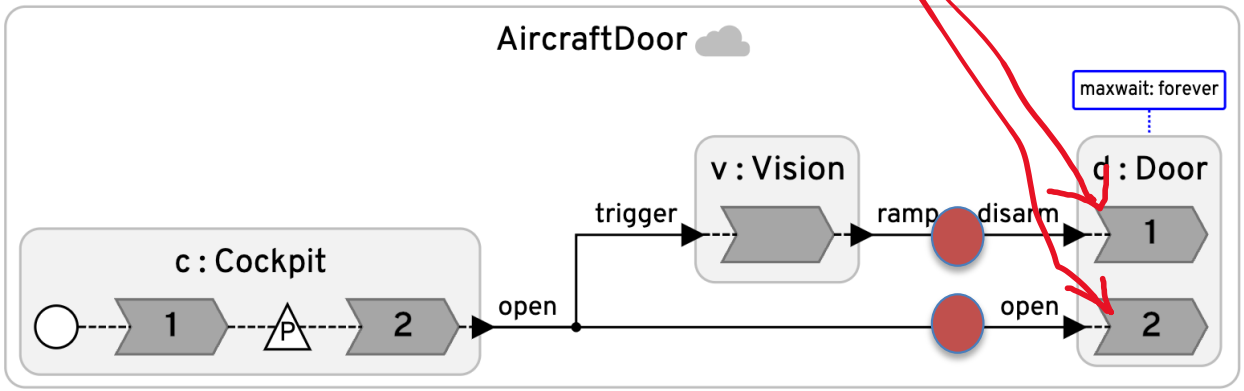
Same timestamp means messages are logically simultaneous and should be handled together.



Logically Simultaneous Events

```
reactor Door {
  input open: bool
  input disarm: bool
  // 0: closed_armed, 1: closed_disarmed, 2: open, 3: open_deployed
  state status: int = 0

  // The reaction to disarm should be first.
  reaction(disarm) {=
    // Disarm the door.
    if (self->status == 0)
      self->status = 1;
  =}

  reaction(open) {=
    // Actuate door opener.
    if (self->status == 0)
      self->status = 3;
    else if (self->status == 1)
      self->status = 2;
  =}
}
```

Reactions to (logically) simultaneous events are handled in lexical order. How?



Decentralized Coordination: Coordination Rule #1

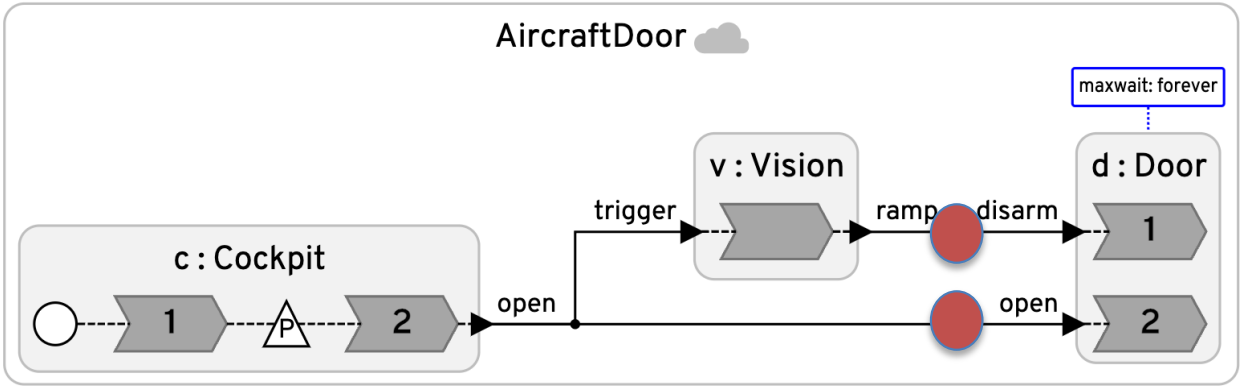
```
federated reactor {
  c = new Cockpit()
  v = new Vision()
  @maxwait(forever)
  d = new Door()

  c.open -> d.open
  c.open -> v.trigger
  v.ramp -> d.disarm
}
```

Each federate advances its logical time locally to *t* when either:

1) Local physical clock reaches *t* and all inputs are known up to and including *t* (all inputs have received messages with tags >= *t*) or
2) Local physical clock >= *t* + maxwait.

With maxwait == forever, the coordination strategy is the same as Chandy and Misra [1988].

With reliable, in-order message delivery, delivers deterministic semantics.

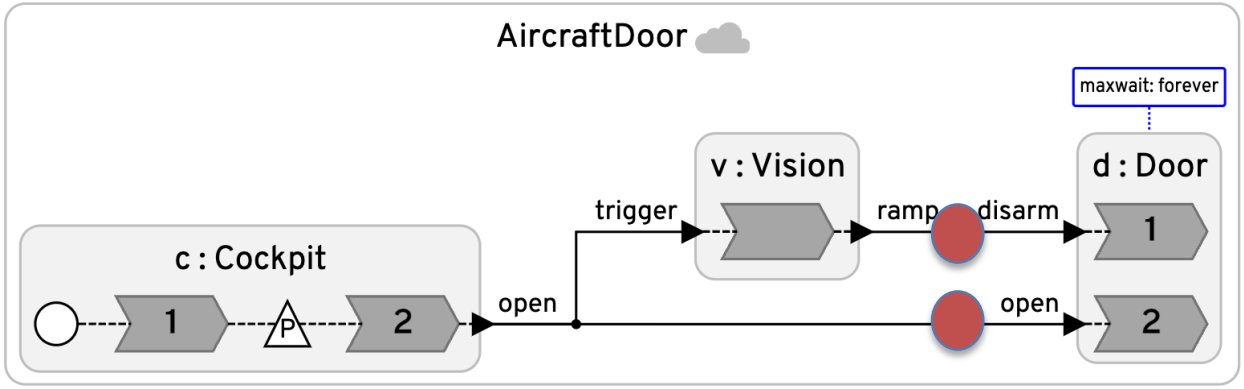


Handling Failure Modes

Choosing maxwait == forever is equivalent to asserting that it is not safe to respond to a network open command without input from the Vision component.

An alternative is a finite maxwait.

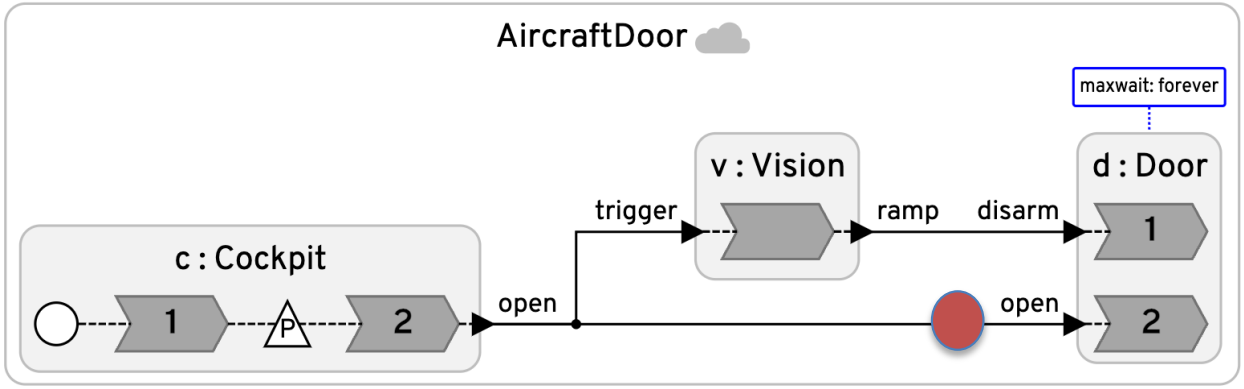
```
federated reactor {
  c = new Cockpit()
  v = new Vision()
  @maxwait(forever)
  d = new Door()

  c.open -> d.open
  c.open -> v.trigger
  v.ramp -> d.disarm
}
```



The Finite Maxwait Alternative

```
reactor Door {
  input open: bool
  input disarm: bool
  // 0: closed_armed, 1: closed_disarmed, 2: open, 3: open_deployed
  state status: int = 0

  // The reaction to disarm should be first.
  reaction(disarm) {=
    // Disarm the door.
    if (self->status == 0)
      self->status = 1;
  =} tardy {=
    // Handle late arrival of disarm signal.
  =}

  reaction(open, disarm) {=
    // Actuate door opener.
    if (!open->is_present || !disarm->is_present
      // Handle failure to receive both inputs.
    }
  =}
}
```

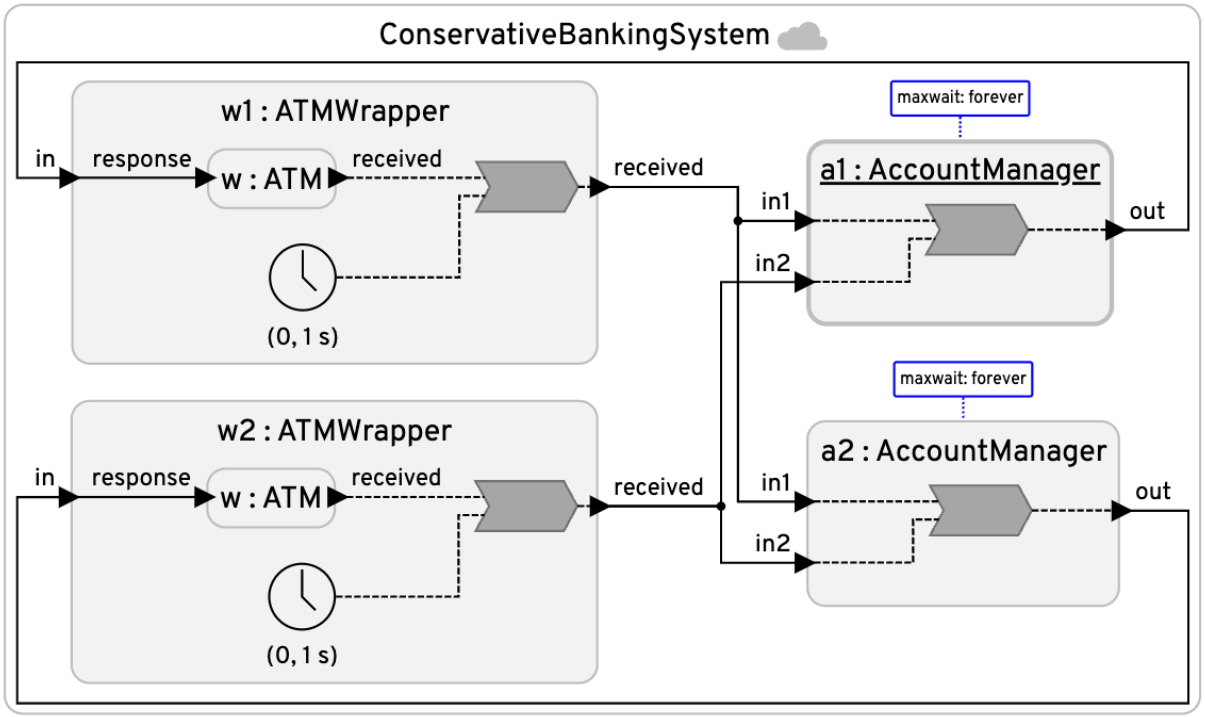
```
federated reactor {
  c = new Cockpit()
  v = new Vision()
  @maxwait(100 ms)
  d = new Door()

  c.open -> d.open
  c.open -> v.trigger
  v.ramp -> d.disarm
}
```



Consistency with Bounded Wait

```
import ATM from "lib/ATM.lf"

reactor ATMWrapper(null_message_period: time = 1 s) {
  input in: int
  output received: int
  timer t(0, null_message_period)

  w = new ATM()
  in -> w.response

  reaction (w.received, t) -> received {=
    if (w.received->is_present) {
      lf_set(received, w.received->value);
    } else {
      // Send a null message.
      lf_set(received, 0);
    }
  =}
}
```

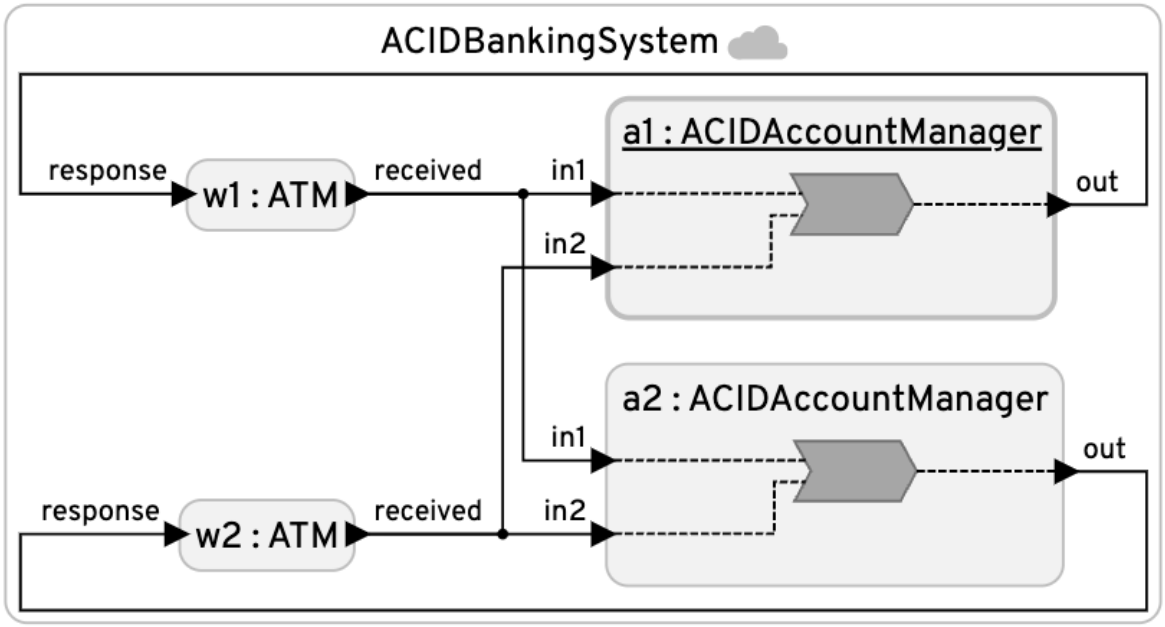
Chandy and Misra [1988] with NULL messages.



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

Eventual Consistency Without Coordination

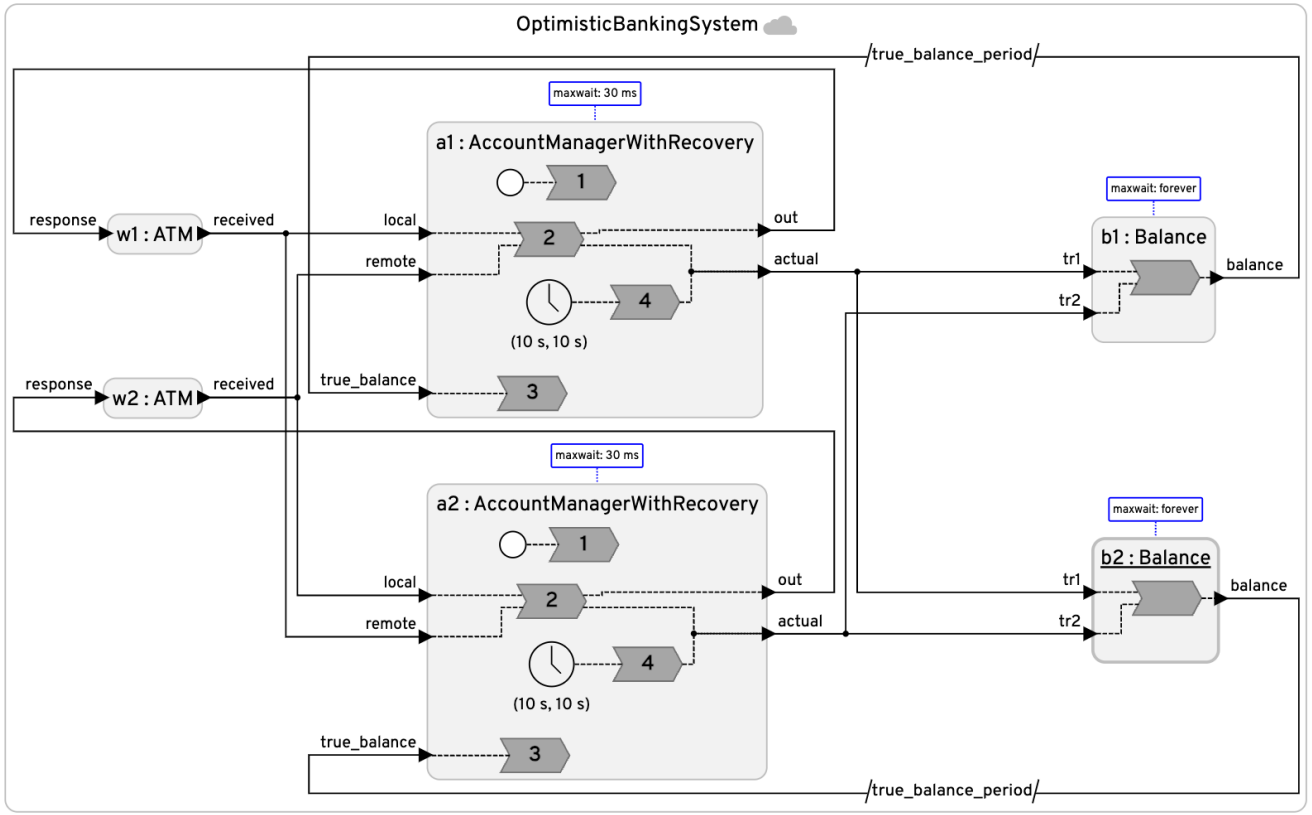
```
reactor ACIDAccountManager {
  input in1: int
  input in2: int
  output out: int
  state balance: int = 0
  reaction(in1, in2) -> out {=
    if (in1->is_present)
      self->balance += in1->value;
    if (in2->is_present)
      self->balance += in2->value;
    lf_set(out, self->balance);
  =} tardy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




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

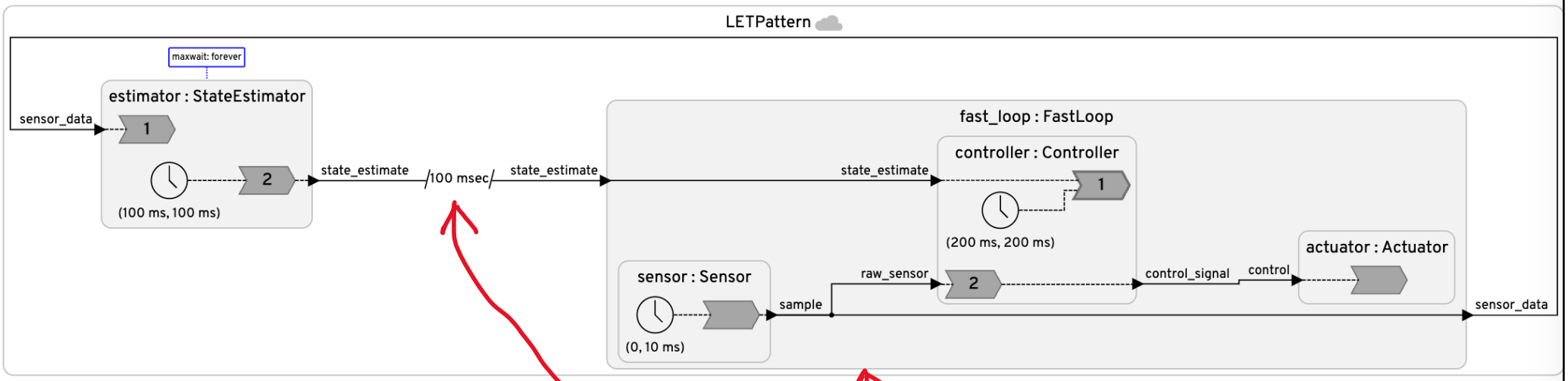
ACID 2.0 [Helland, 2021], CRDTs [Shapiro, et al., 2011], CALM [Hellerstein & Alvaro, 2021]

Optimistic Execution with Rollback



TimeWarp [Jefferson, 1985]

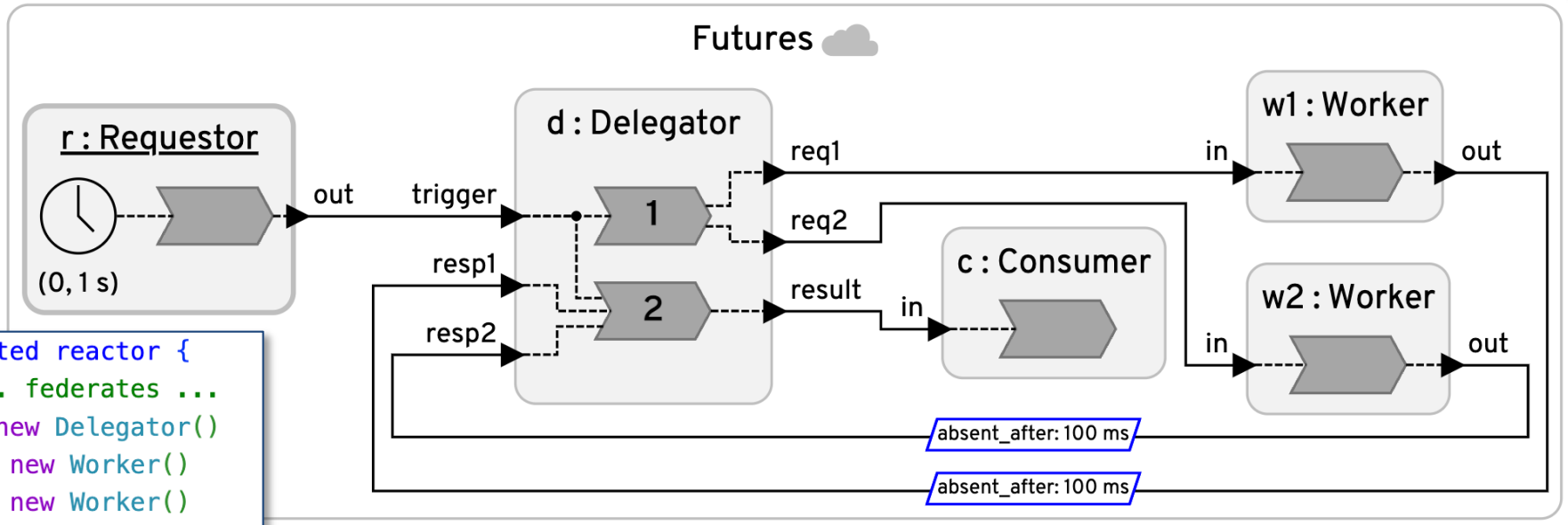
Logical Execution Time (LET)




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

  // Fast loop sends sensor data to estimator
  fast_loop.sensor_data -> estimator.sensor_data

  // State estimator feeds back to fast loop with 100ms logical delay
  estimator.state_estimate -> fast_loop.state_estimate after 100 msec
}
```

Maxwait == 0.
If StateEstimator $X + E + L > 100$ ms, a fault condition will occur.
Controller will detect this fault.

Remote Procedure Calls with Futures



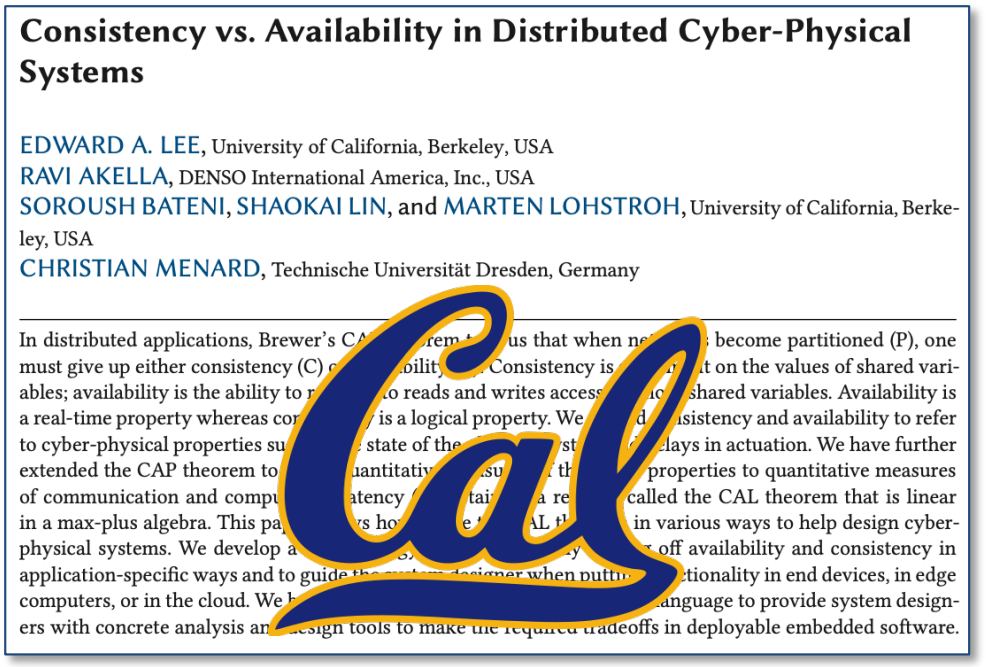

```
federated reactor {
  //... federates ...
  d = new Delegator()
  w1 = new Worker()
  w2 = new Worker()
  d.req1 -> w1.in
  d.req2 -> w2.in
  @absent_after(100 ms)
  w1.out -> d.resp1
  @absent_after(100 ms)
  w2.out -> d.resp2
}
```

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

Fundamental Limits

Consistency fundamentally comes with a cost in availability, where the cost is proportional to latency.

Maxwait (and absent_after) enable explicitly managing this tradeoff and detecting faults as early as possible.



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

In distributed applications, Brewer's CA ...rem ...us that when ne... become partitioned (P), one must give up either consistency (C) ... ...bilit... Consistency is ... ...t on the values of shared variables; availability is the ability to r... ...o reads and writes access ... ...shared variables. Availability is a real-time property whereas cor... ... is a logical property. We ... d ...sistency and availability to refer to cyber-physical properties su... ... state of the ... st... ...lays in actuation. We have further extended the CAP theorem to ... ...uantitativ... ...su... ...t... properties to quantitative measures of communication and compu... ...atency ... ...tai... ...a re... called the CAL theorem that is linear in a max-plus algebra. This pa... ...ys ho... ...e ... ...AL th... in various ways to help design cyber-physical systems. We develop a... ...S... ...y ... ...off availability and consistency in application-specific ways and to guide the ... ...designer when puttin... ...ctionality in end devices, in edge computers, or in the cloud. We h... ...anguage to provide system designers with concrete analysis an... ...esign tools to make the required tradeoffs in deployable embedded software.

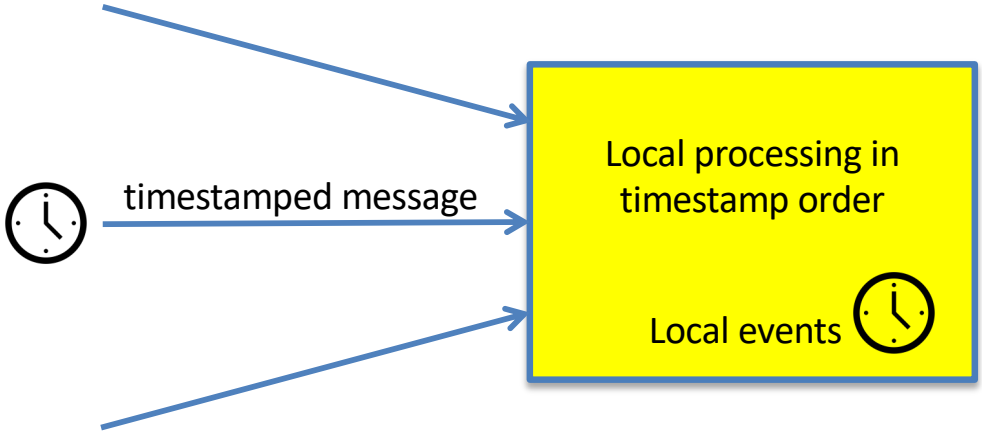
Lee, E. A., et al. (2023). "Consistency vs. Availability in Distributed Cyber-Physical Systems." ACM Transactions on Embedded Computing Systems (TECS) **22**(5s): 1-24.

If you don’t care about consistency,

- **Actors**: Set maxwait == 0 and ignore timestamp order violations.
- **Pub-Sub**: An output port is a topic, an input port a subscription. maxwait == 0.

In both cases, the use of ports makes the code more modular.

The Problem We Solve



When should you process an incoming event and still be sure you process all events in timestamp order (ensuring **Consistency**) and how to define and handle faults?

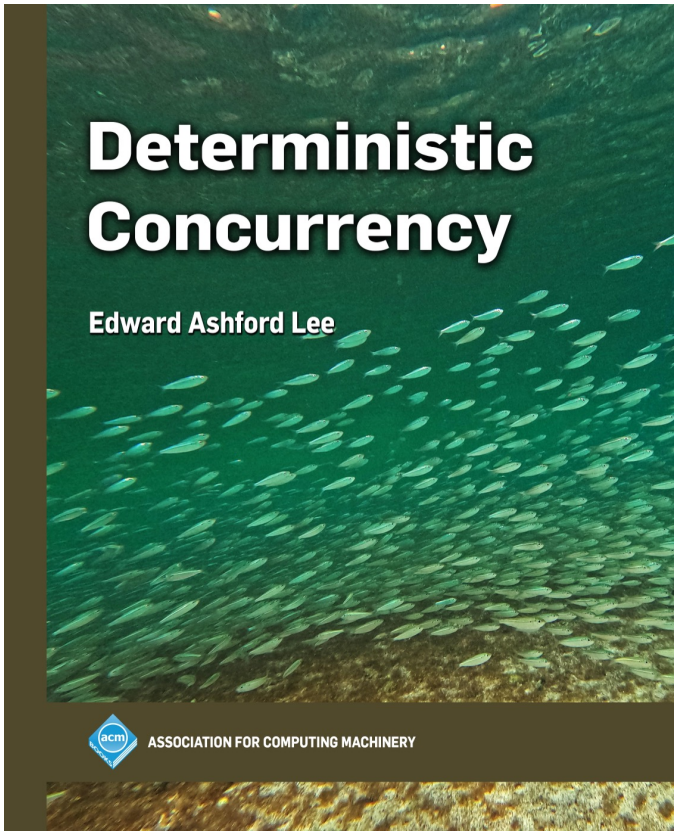
The CAL theorem says that you must wait (**Availability** cost). Penalty depends on $L + E + X$ (apparent **Latency**), not on L or E or X alone.

Conclusion

https://lf-lang.org

Maxwait handles many distributed computing patterns:

- Chandy-and-Misra, with/without NULL messages
- Fault tolerant Chandy-and-Misra
- ACID/CRDT/CALM
- Optimistic with or without rollback
- Logical Execution Time (LET)
- Remote procedure calls (RPC) with futures
- Pub-Sub
- Actors



Book in progress, est.Spring 2027.