



iCyPhy

Distributed Time-Sensitive Systems

Edward A. Lee

Professor of the Graduate School
Distinguished Professor Emeritus

Keynote

IEEE/ACM Embedded Systems Week (ESWEEK)

October 6, 2026, Barcelona, Spain

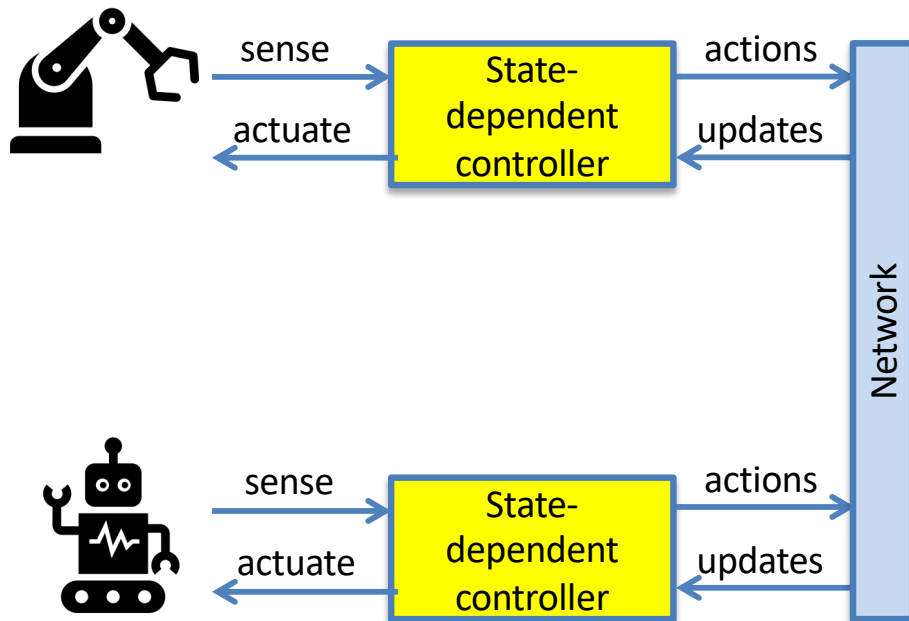


University of California at Berkeley





Distributed Time-Sensitive Systems



Examples

- Factory floor
- Traffic control
- Power systems
- Automotive

Requires a
*consistent view of
system state and
timely responses.*

Requirements:

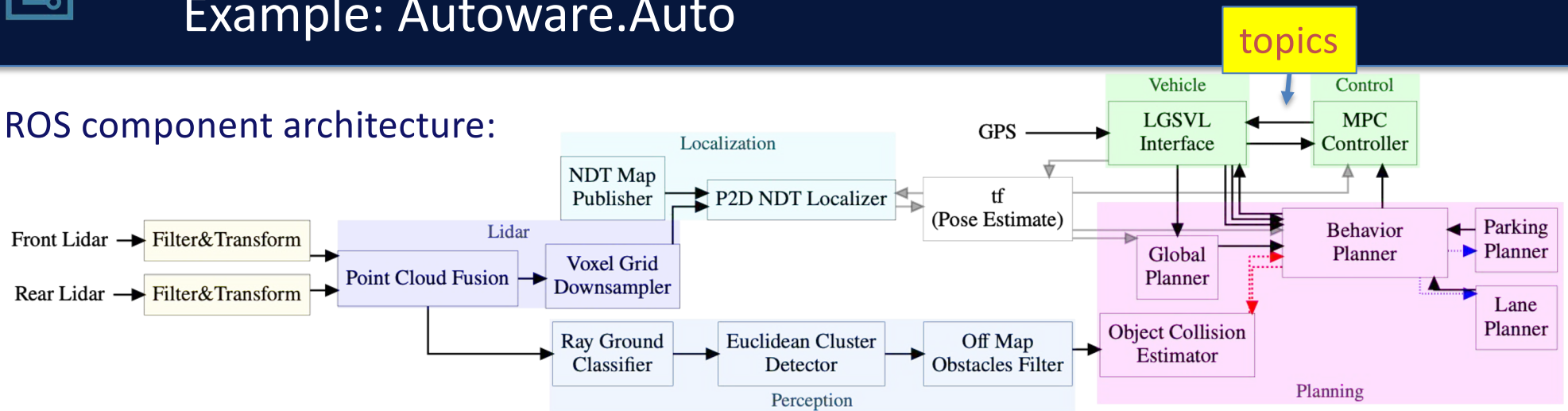
- Control over timing
- Distributed control
- Safe resource sharing
- Operating limits
- Fault management



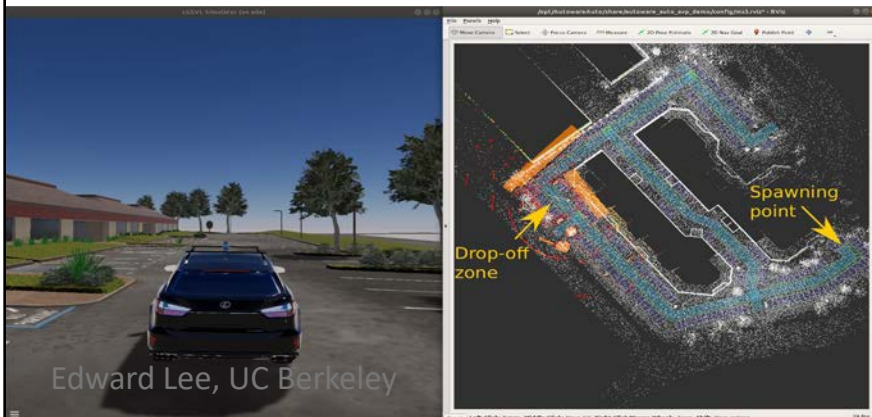
The Treachery of Inconsistency

Example: Autoware.Auto

ROS component architecture:

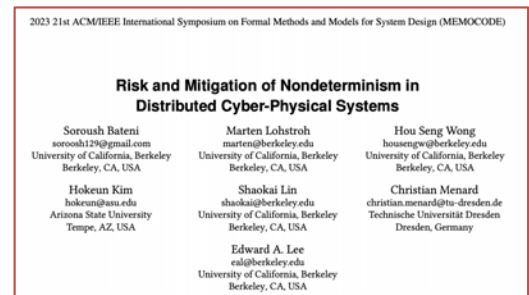


LGSVL simulation of the vehicle:



Edward Lee, UC Berkeley

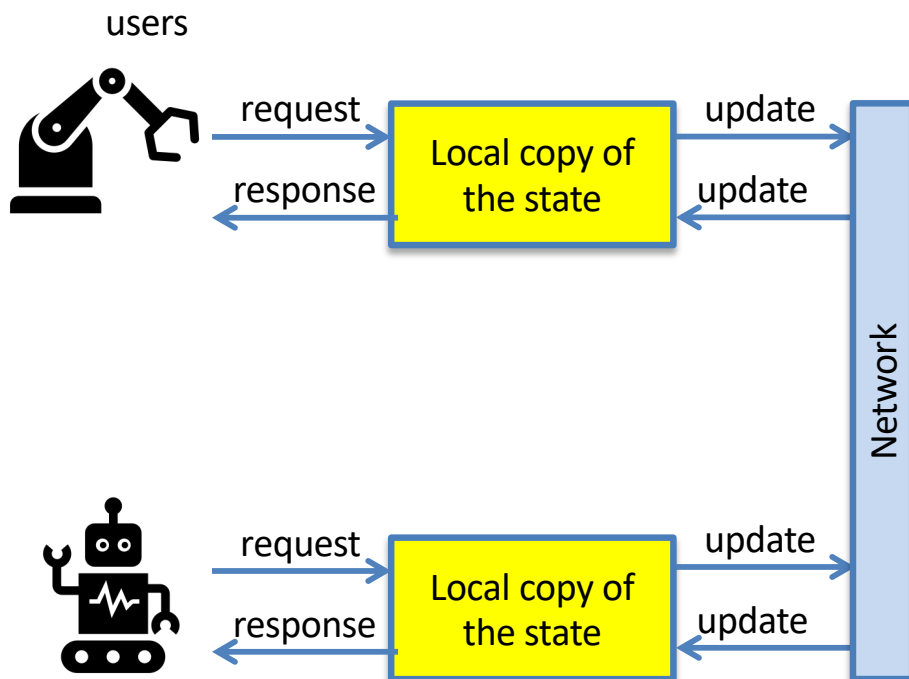
Occasionally, the planner gets into a **mode confusion** state where it thinks the car is moving forward in reverse or backward in drive.



Bateni, et al., MEMOCODE'23



Consistency Requires Determinism



Today's approaches are nondeterministic:

- 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:

- Difficult to ensure consistency
- Difficult to test
- Dangerous behaviors may be rare



Determinism as a Property of *Models*

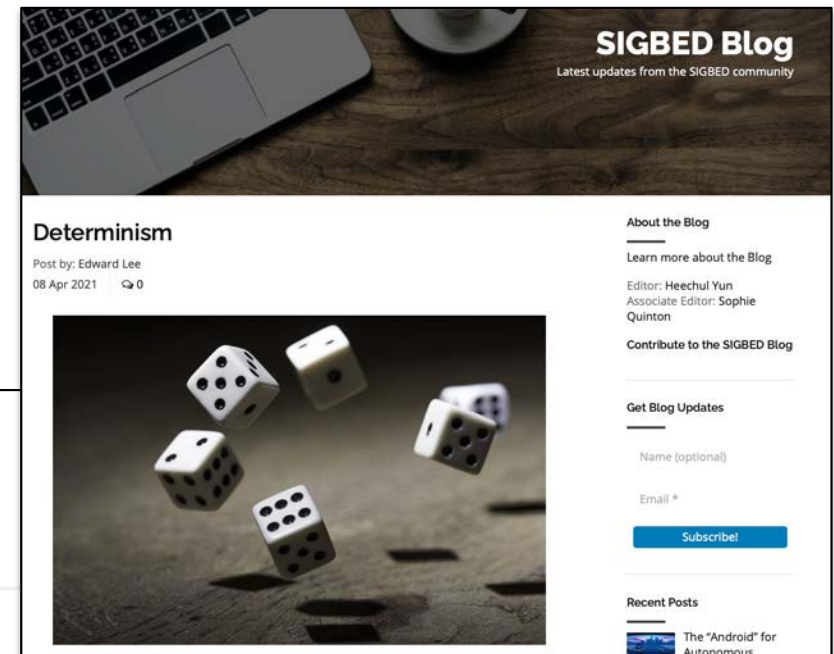
A *model* is *deterministic* if, given the initial *state* and the *inputs*, the model defines exactly one *behavior* as seen by an *observer*.

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>





Example 1

```
// 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");
```



Is this program deterministic?

It depends on how you define observable behavior.

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

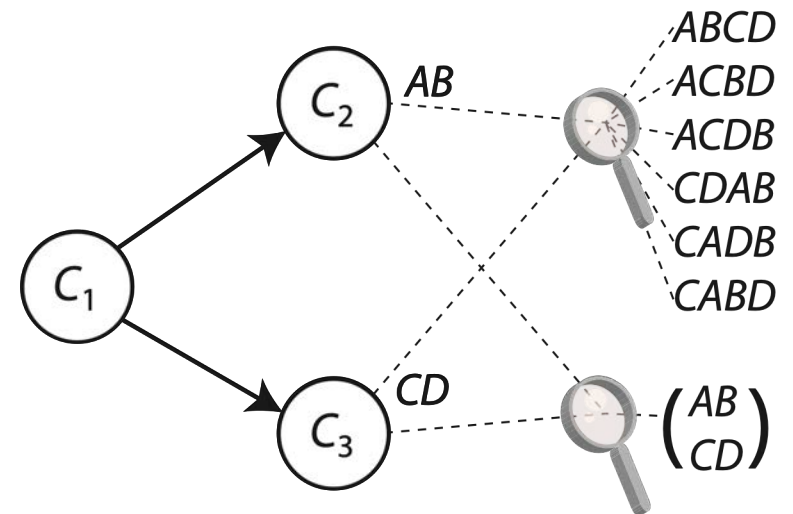
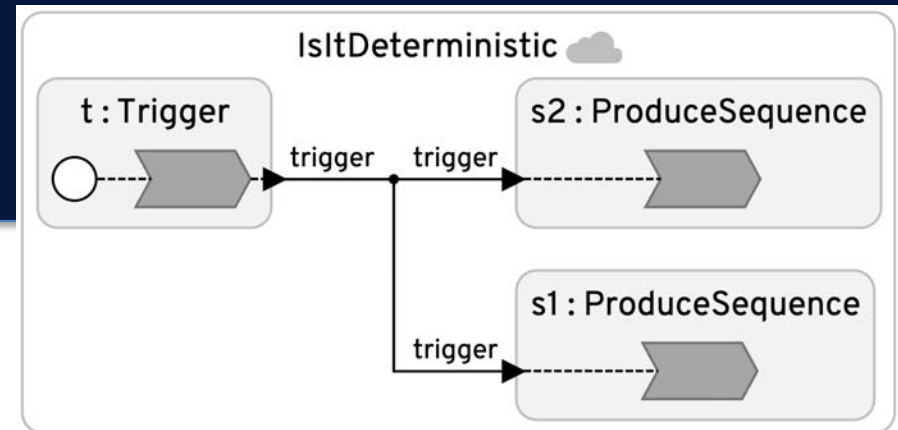


Example 2: With Concurrency

Consider three tasks. C_1 dispatches a request to C_2 and C_3 , which concurrently produce output sequences AB and CD respectively.

Is this program deterministic?

It depends on how you define observable behavior.



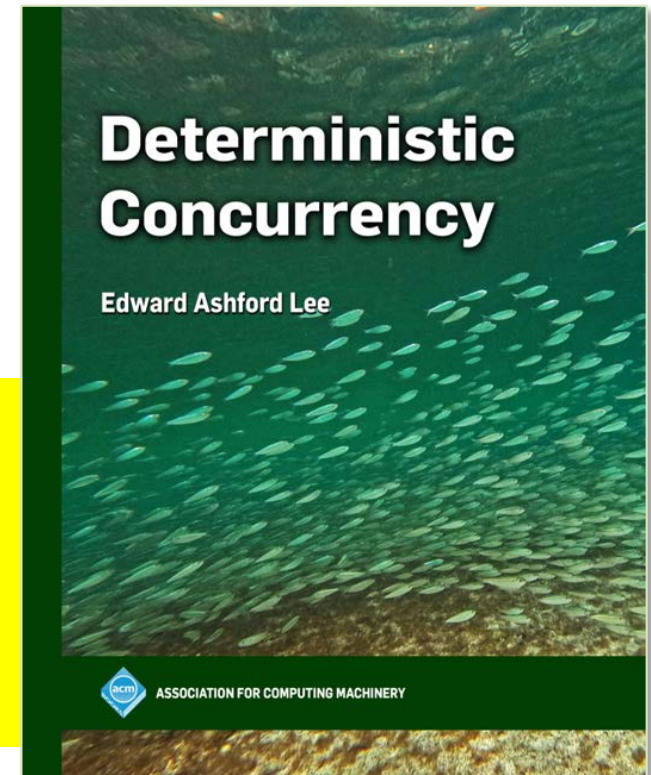


Deterministic Models for Software

Not a new topic:

- Imperative programs (1950s - ...)
- Dataflow (1970s - ...)
- Kahn networks (1970s - ...)
- Synchronous languages (1980s - ...)
- Discrete-event systems (1980s - ...)
- Logical execution time (2000s - ...)
- Reactors (2020s - ...)

Today: Focus on time-sensitive systems



Book Expected in Dec. 2026.
Draft at <https://ptolemy.org/~eal>



Reactor Principle: Logical Time vs. Physical Time

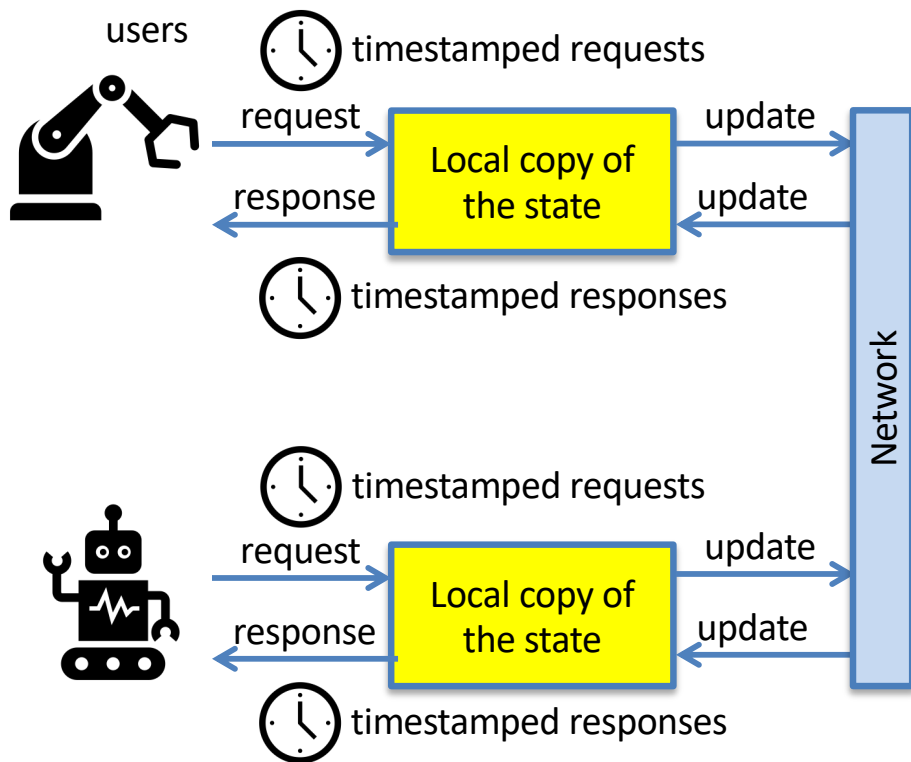


Separate “logical time” from “physical time.”

- 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.



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**:

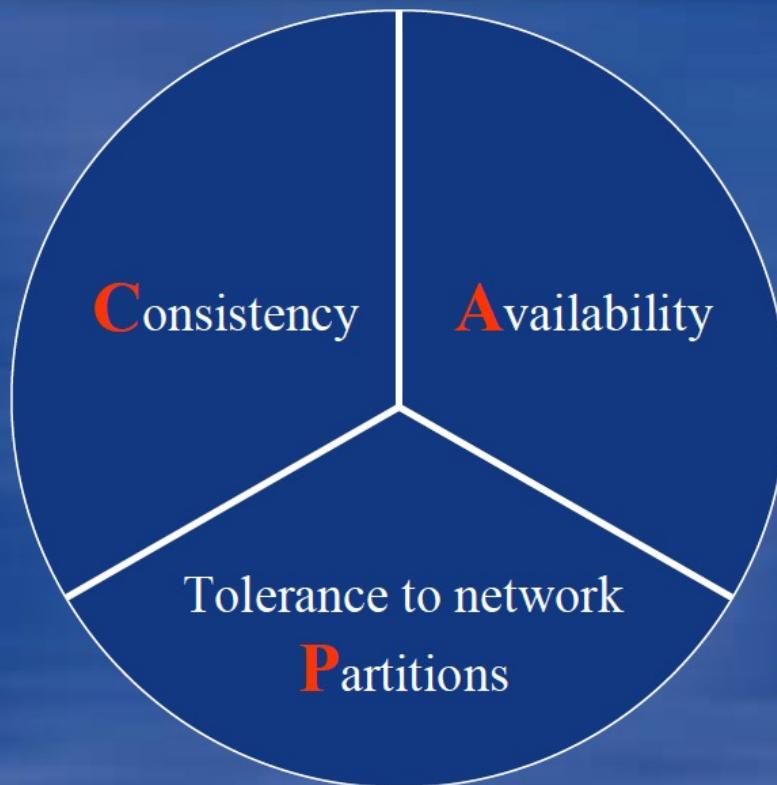
Either **Consistency** or **Availability** or both must decrease as **Latency** increases.

[Lee, et al., '21, '23, '26]



Inktomi

The CAP Theorem



Eric Brewer
Berkeley & Google

Theorem: You can have at most two of these properties for any shared-data system

PODC Keynote, July 19, 2000



Fundamental Limits: The CAL Theorem

Consistency fundamentally comes with a cost in **Availability** (timeliness), where the cost increases as apparent **Latency** ($L + E + X$) increases.

Generalizes and quantifies Brewer's classic CAP theorem.

Edward Lee, UC Berkeley

arXiv

Computer Science > Distributed, Parallel, and Cluster Computing

[Submitted on 16 Sep 2021]

Quantifying and Generalizing the CAP Theorem

Edward A. Lee, Soroush Bateni, Shaokai Lin, Marten Lohstroh, Christian Menard

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 Universitt Mnchen, Germany

In distributed applications, Brewer's CAP theorem must give up either consistency (C) or availability (A); availability is the ability to respond to reads in a real-time property whereas consistency is a logical property. This paper extends the CAP theorem to relate quantitative properties such as the state of the system to cyber-physical properties such as the state of the system. We develop a quantitative algebra. This paper shows how to use the algebra to reason about the system in application-specific ways and to design the system for computers, or in the cloud, or on the LINGU. The paper also discusses design tools to manage the complexity of the system.

CAL

Deterministic Concurrency

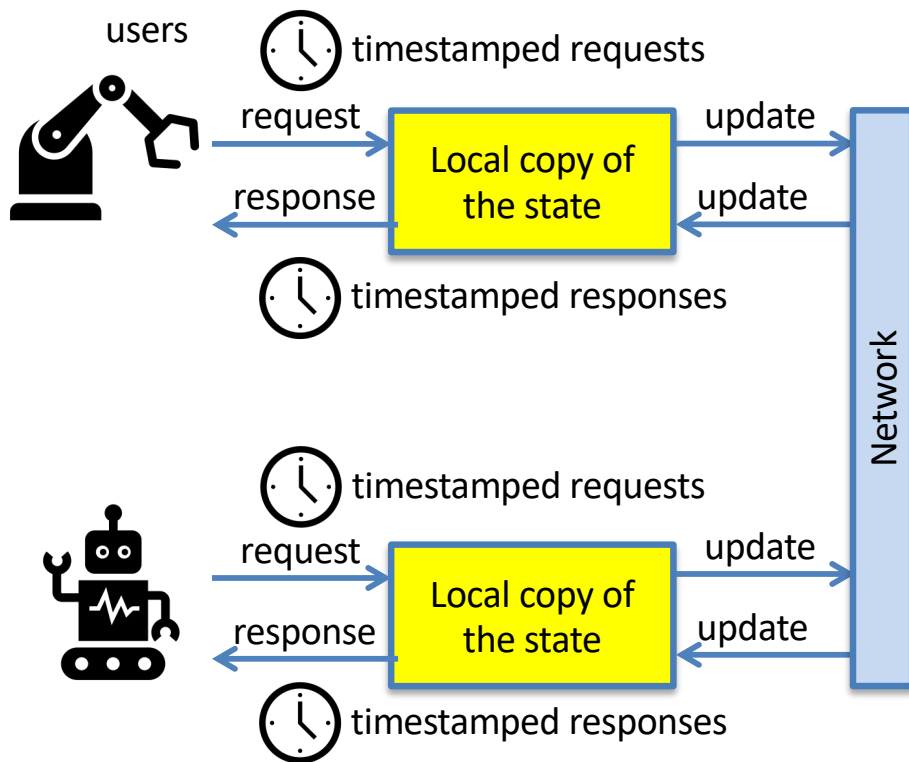
Edward Ashford Lee

ASSOCIATION FOR COMPUTING MACHINERY

<https://ptolemy.org/~eal>



Consistency, Availability, and Latency

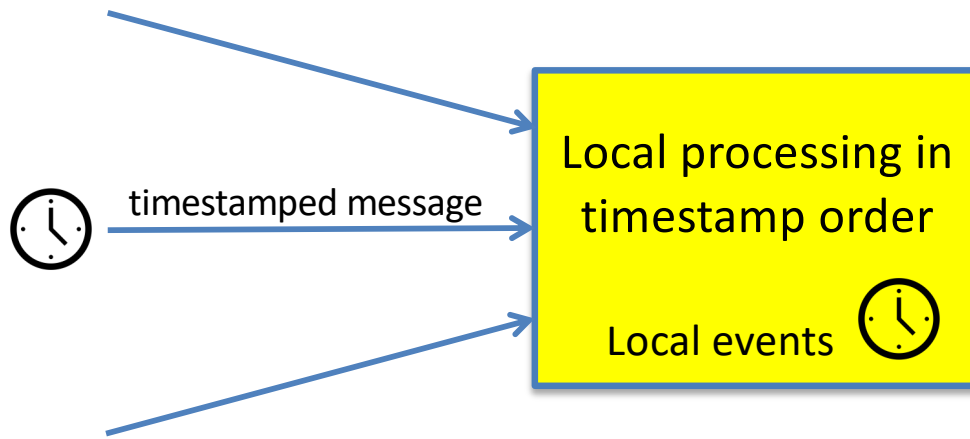


- If you can tolerate unbounded unavailability, consistency can be assured.
- If and only if you bound **LEX**:
 - network Latency L
 - clock synchronization Error E
 - eXecution time Xthen you can bound unavailability while preserving consistency.

Fundamental limit: **CAL Theorem**:
Either **C**onsistency or **A**vailability or both must decrease as **L**atency increases.
[Lee, et al., '21, '23, '26]



The Problem We Solve With Reactors



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



The Reactor Model of Computation

Input ports:

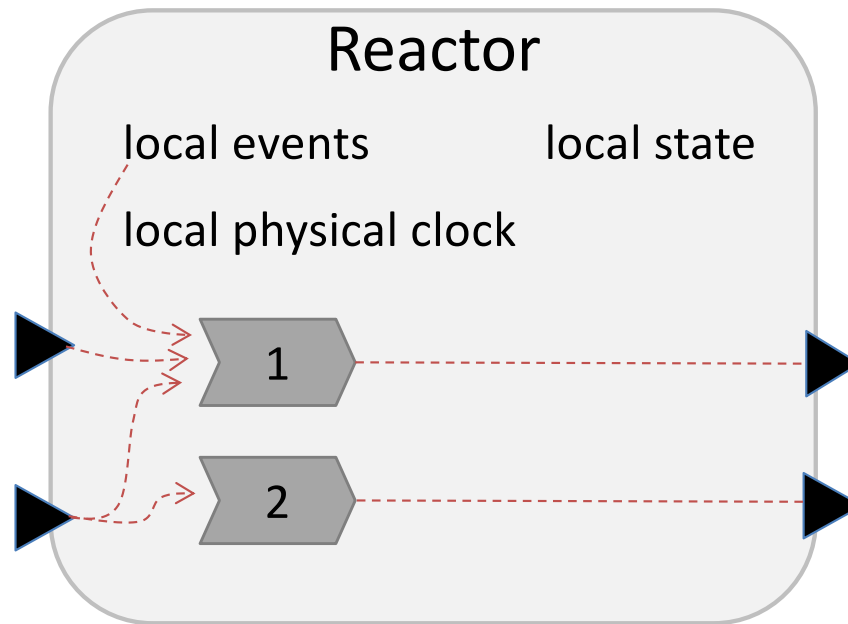
Timestamped events are received in timestamp order at each port.

Reactions:

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

Output ports:

Written to by reactions.
Logically instantaneous.



Local events:

- Periodic
- Scheduled
- Asynchronous

Local physical clock:

Asynchronous events are timestamped using this clock.

Goal is to process events in timestamp order.

When events are simultaneous, use reaction order.



Conservative Decentralized Coordination for Distributed Systems

Input ports:

Timestamped events are received in timestamp order at each port.

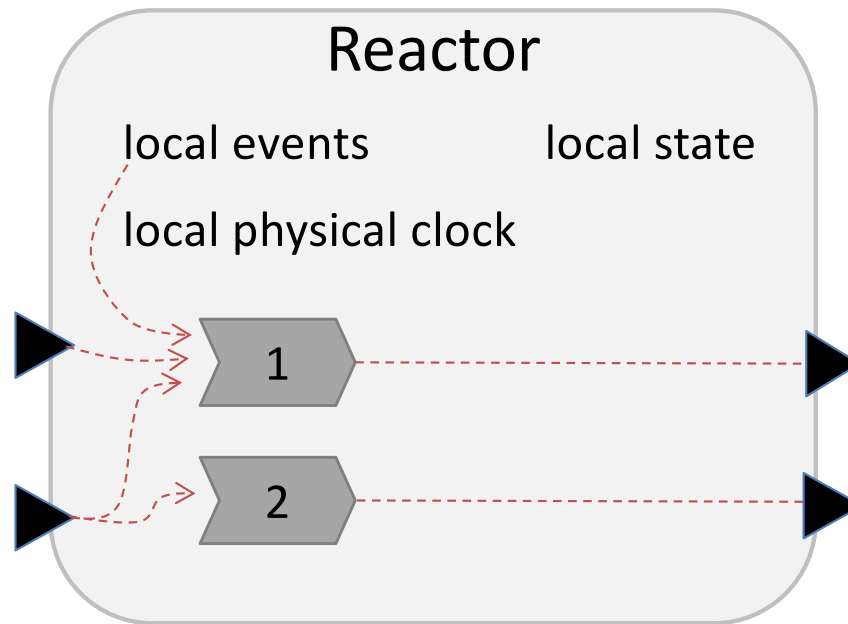
Reactions:

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

Output ports:

Written to by reactions.
Timestamp matches the trigger.

Edward Lee, UC Berkeley



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.

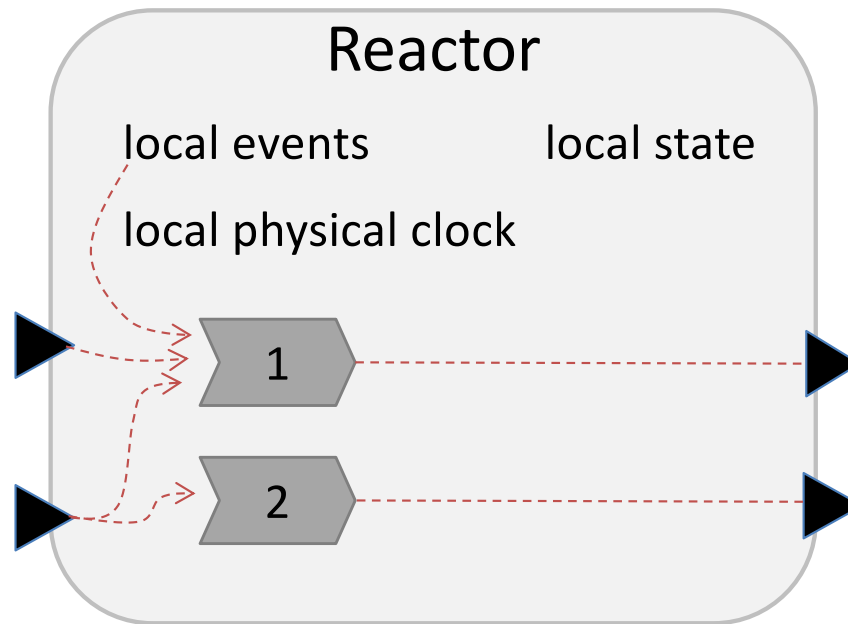


Conservative Decentralized Coordination has Unbounded Unavailability

If any network connection or remote component fails, this policy waits forever.

Unavailability is unbounded.

An unavoidable consequence of the CAP & CAL theorems.



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.

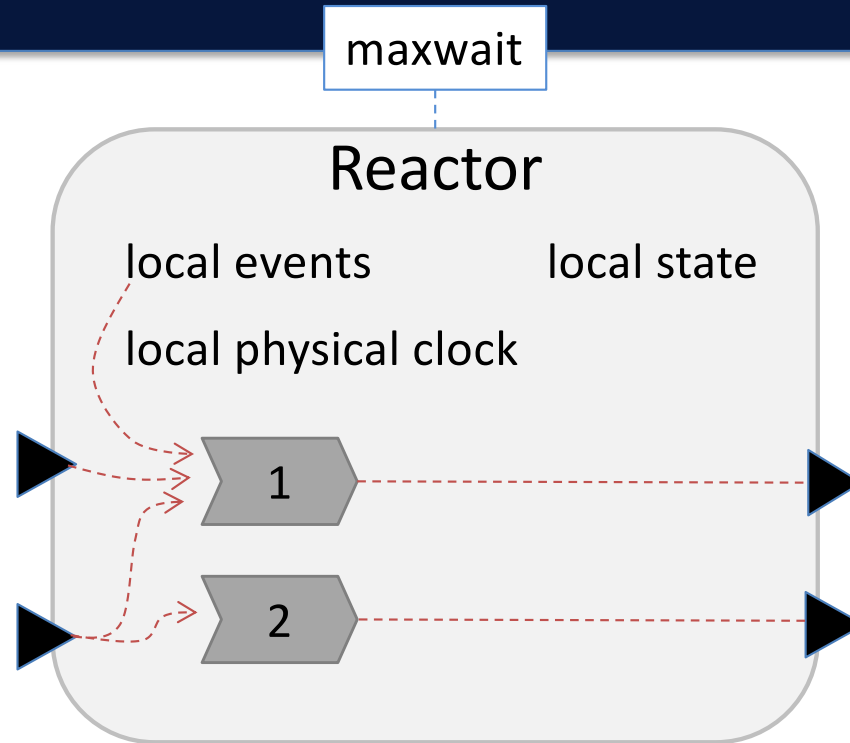


Bounded Unavailability Maxwait

Unavailability is now bounded by maxwait.

If input is excessively delayed (or lost), then inconsistency can occur (unavoidable per the CAL theorem).

But assumptions are clear, and violations are detectable!



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. $T > t + \text{maxwait}$.

(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>

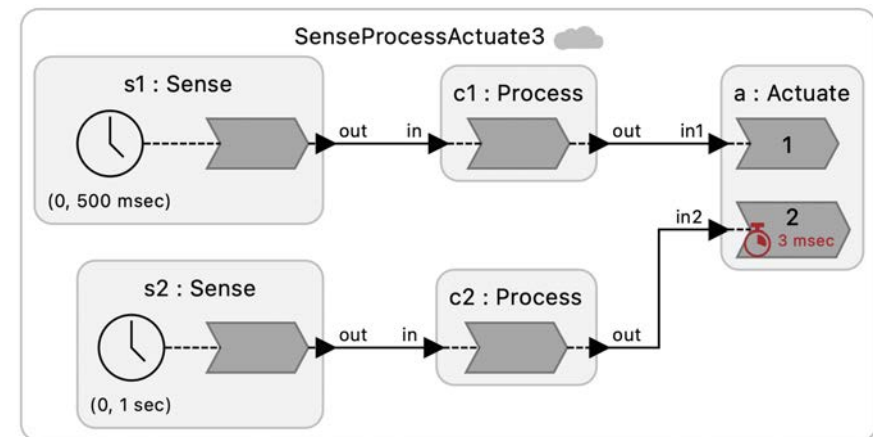


Build **predictable**, **concurrent**, **time-sensitive**, and **distributed** systems.

Lingua Franca is a coordination language for efficient, deterministic, multi-threaded, time-sensitive, embedded, and distributed programs. The result of decades of research, it offers more repeatable behavior than other concurrent programming frameworks, including threads, pub-sub, actors, and service-oriented architectures.

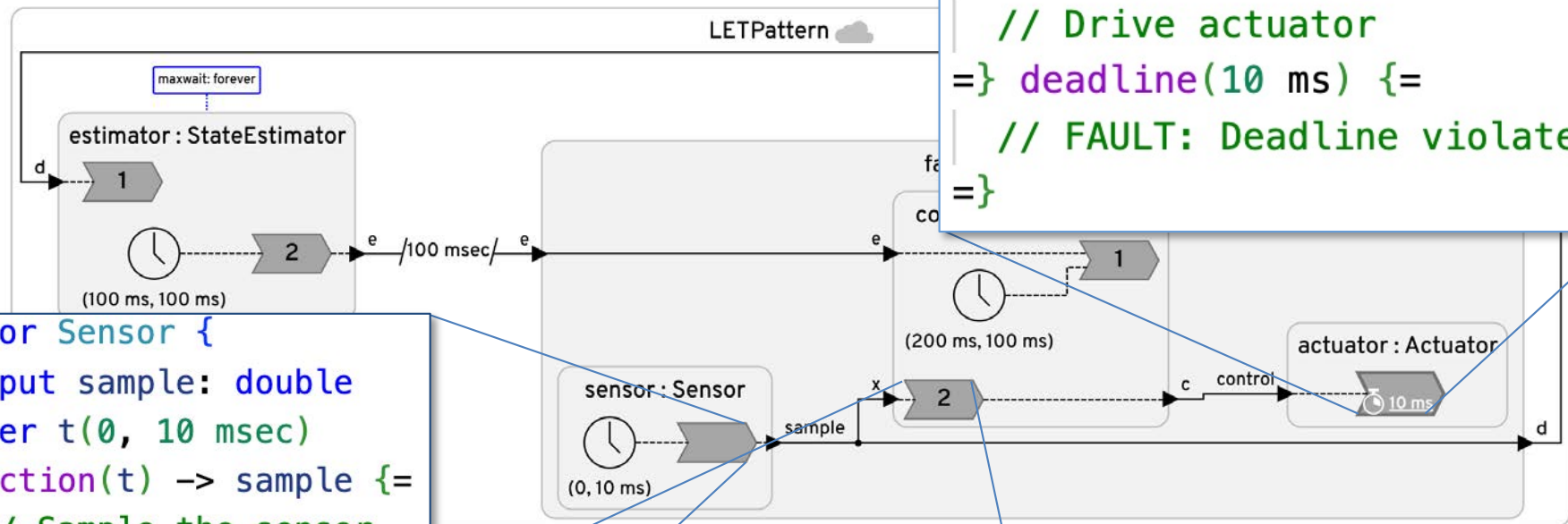
Get Started Read the Docs 296

Deterministic Concurrency Built-in Timing Semantics Simplified Distribution





Classic Sense-Control-Actuate Chain



```
reactor Sensor {
    output sample: double
    timer t(0, 10 msec)
    reaction(t) -> sample {=
        // Sample the sensor.
        ... your code here ...
        lf_set(sample, value);
    =}
}
```

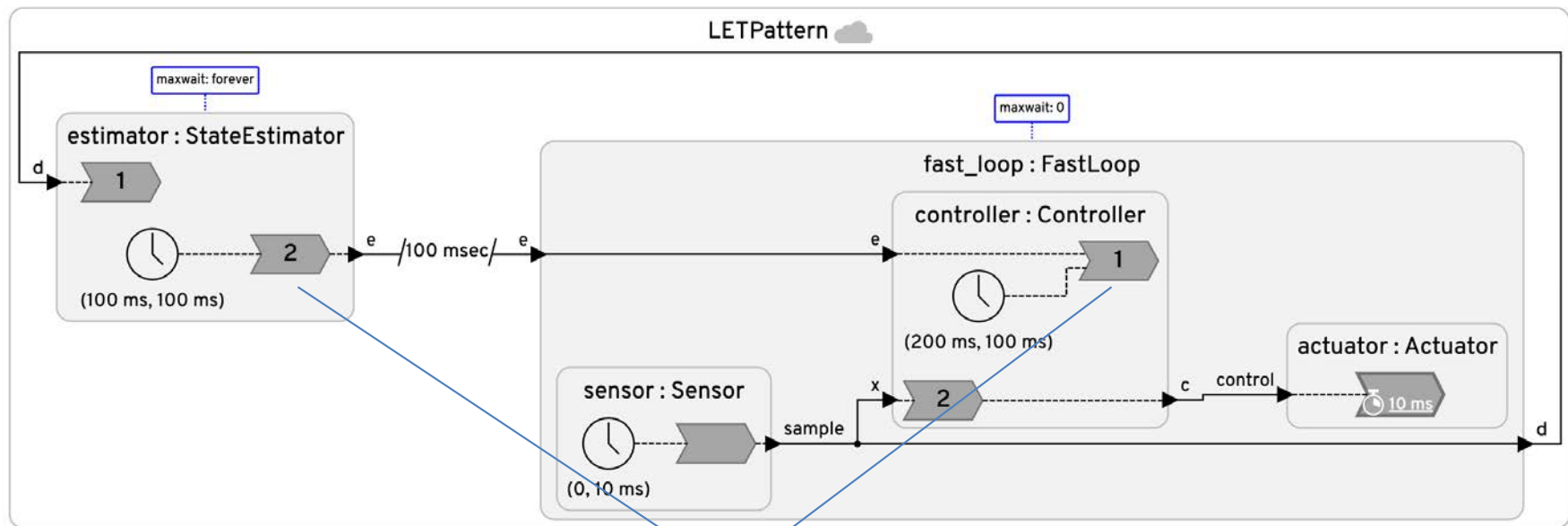
```
reaction(x) -> c {=  
  // Control (e.g. PID)  
=}
```

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

Deadline guides scheduling and provides a **fault handler** for late inputs.



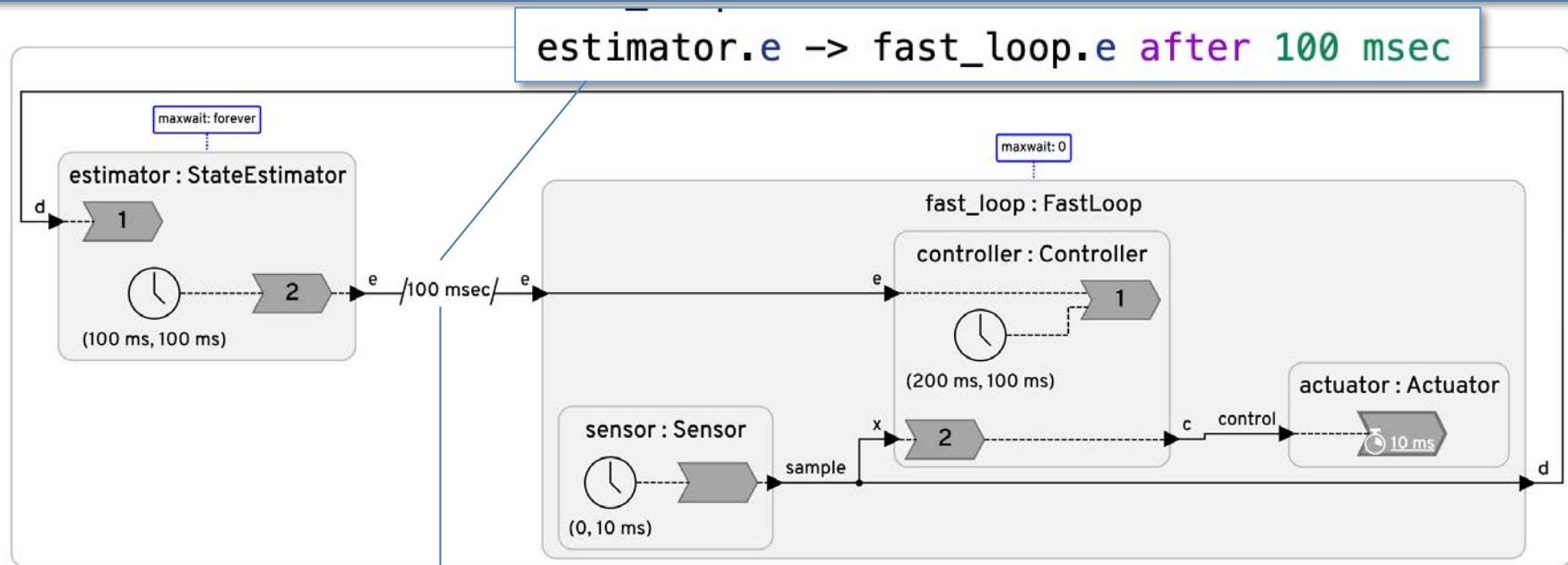
Concurrent Tasks



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



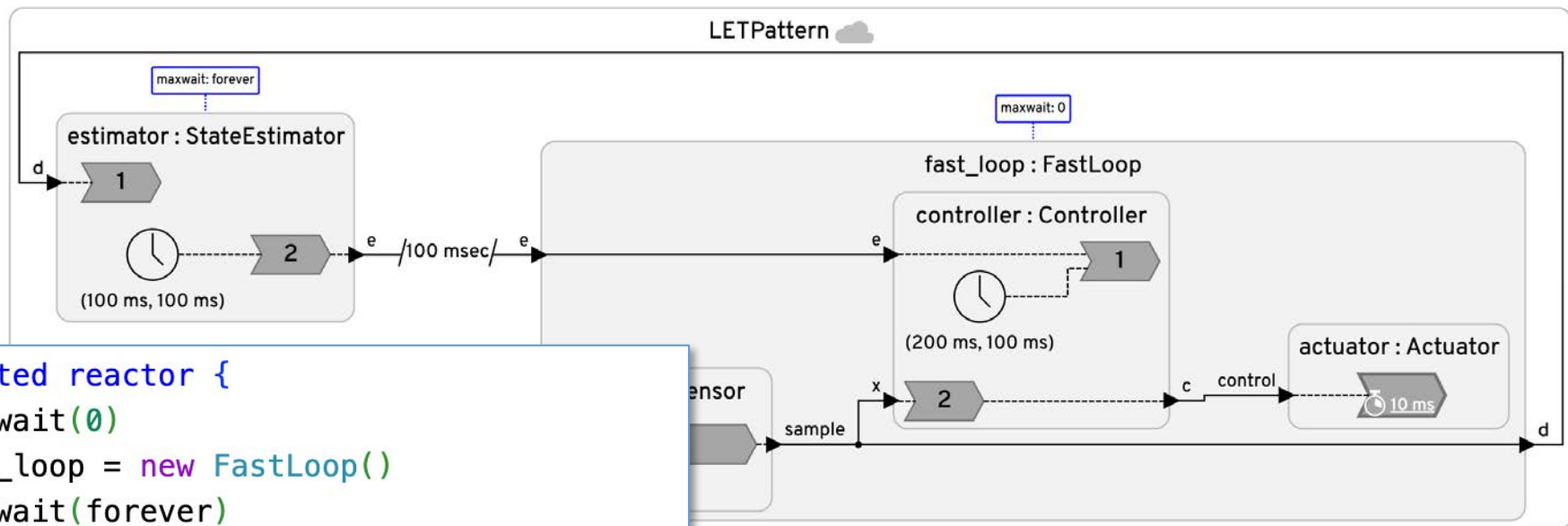
Logical Execution Time (LET)



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



Federated Lingua Franca

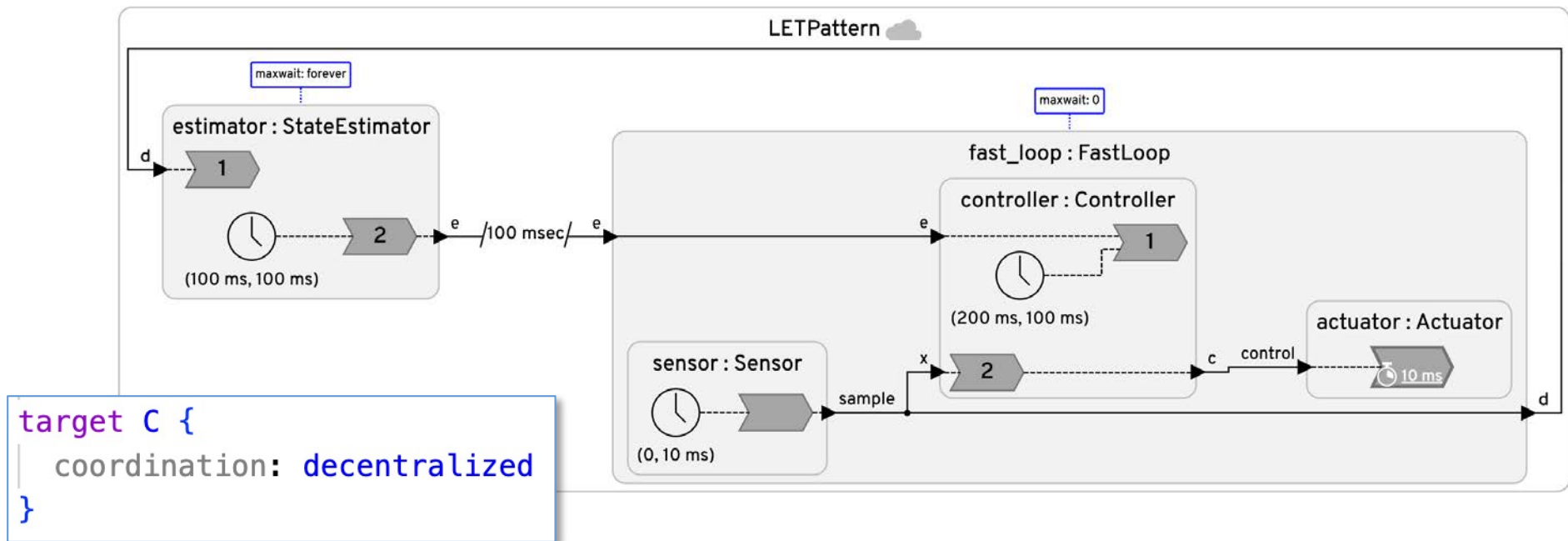


```
federated reactor {  
  @maxwait(0)  
  fast_loop = new FastLoop()  
  @maxwait(forever)  
  estimator = new StateEstimator()  
  
  fast_loop.d -> estimator.d  
  estimator.e -> fast_loop.e after 100 msec  
}
```

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



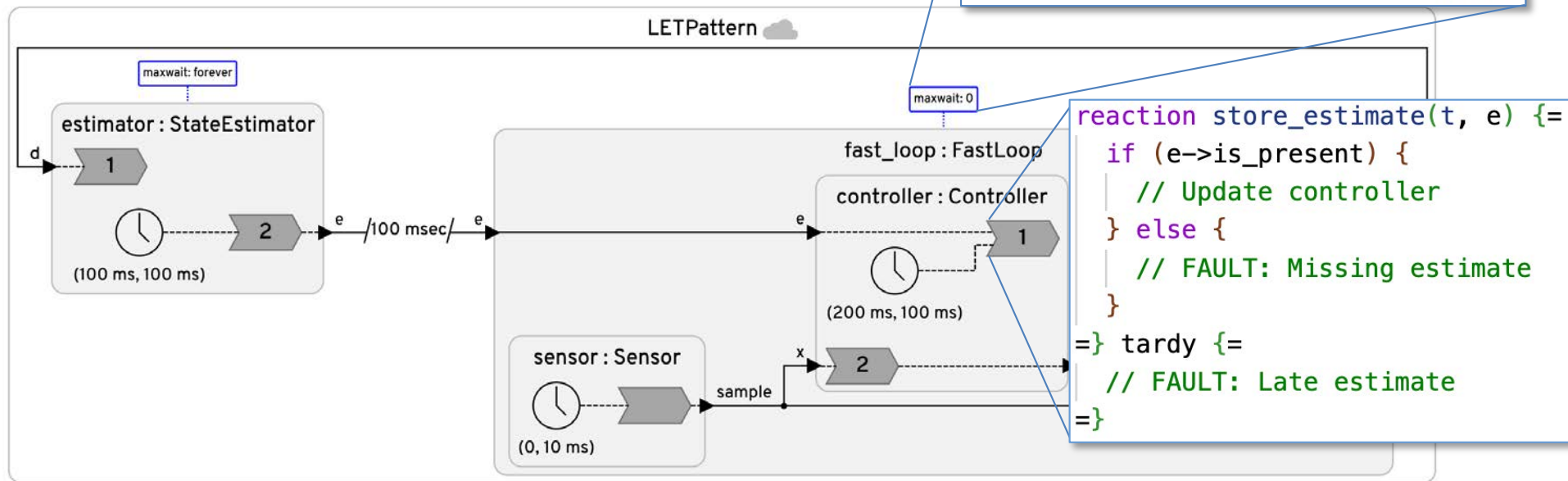
Decentralized Coordination



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



Fast Responses + Fault Handling



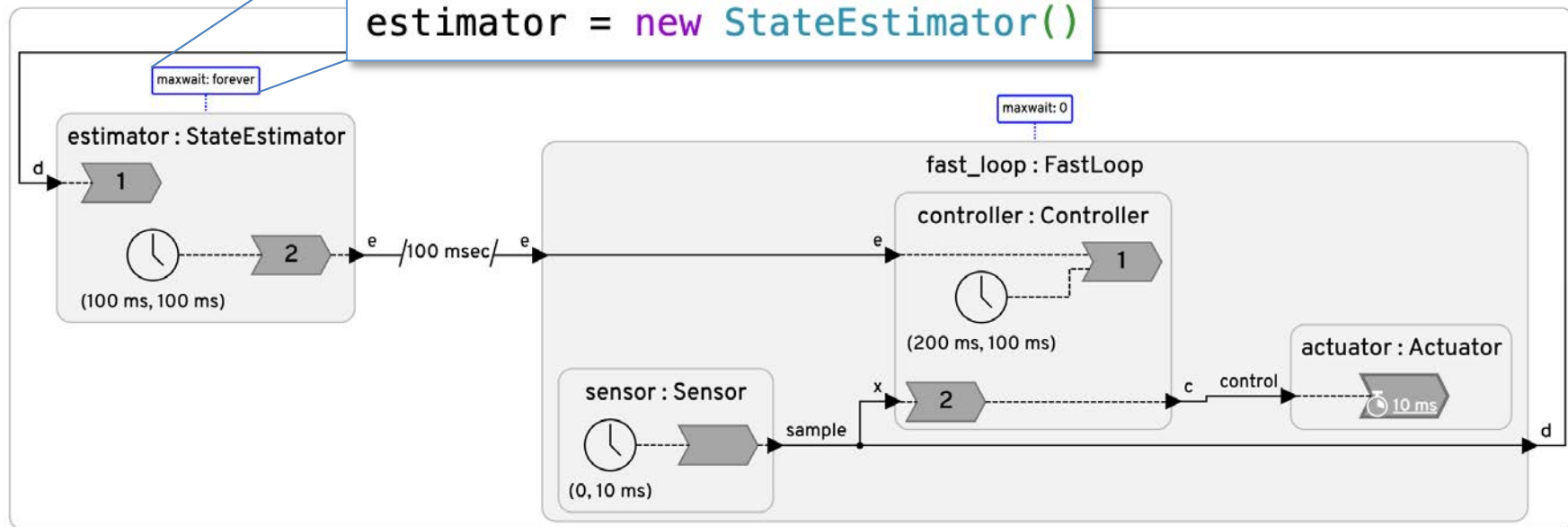
A maxwait of zero means the federate can react to events without waiting.

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



Enforcing Required Data

```
@maxwait(forever)  
estimator = new StateEstimator()
```



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

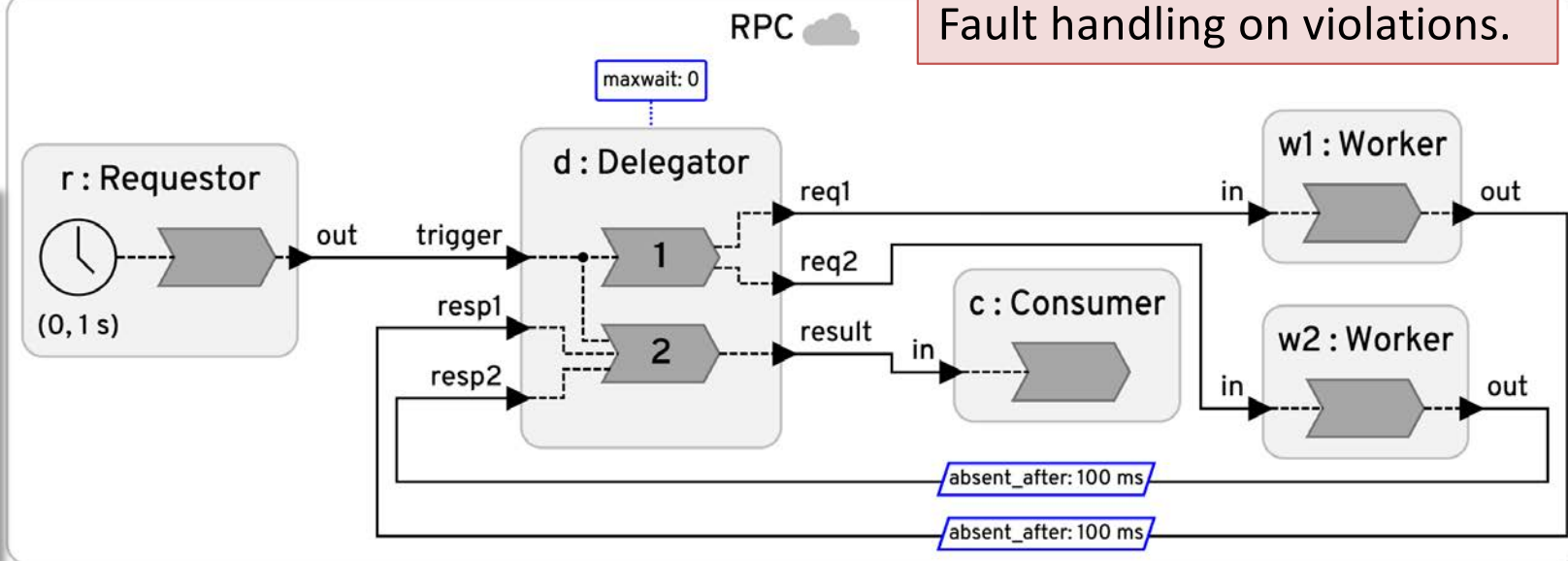
- Input data is not optional.
- Reaction order is assured.



Ex. 2: Remote Procedure Calls with Fault Tolerance

Deterministic under clearly stated assumptions.
Fault handling on violations.

```
federated reactor {  
  r = new Requestor()  
  @maxwait(0)  
  d = new Delegator()  
  w1 = new Worker()  
  w2 = new Worker()  
  c = new Consumer()  
  r.out -> d.trigger  
  d.req1 -> w1.in  
  d.req2 -> w2.in  
  d.result -> c.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.



Determinism? Really?

The world is unpredictable.
Things will go wrong.

Are deterministic models
really a good fit?



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”



Asking the Right Questions

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).



A Program is an *Engineering Model*

```
// 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: a number
Behavior: printed outputs
Observer: only sees printed outputs

The *model* is deterministic.

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

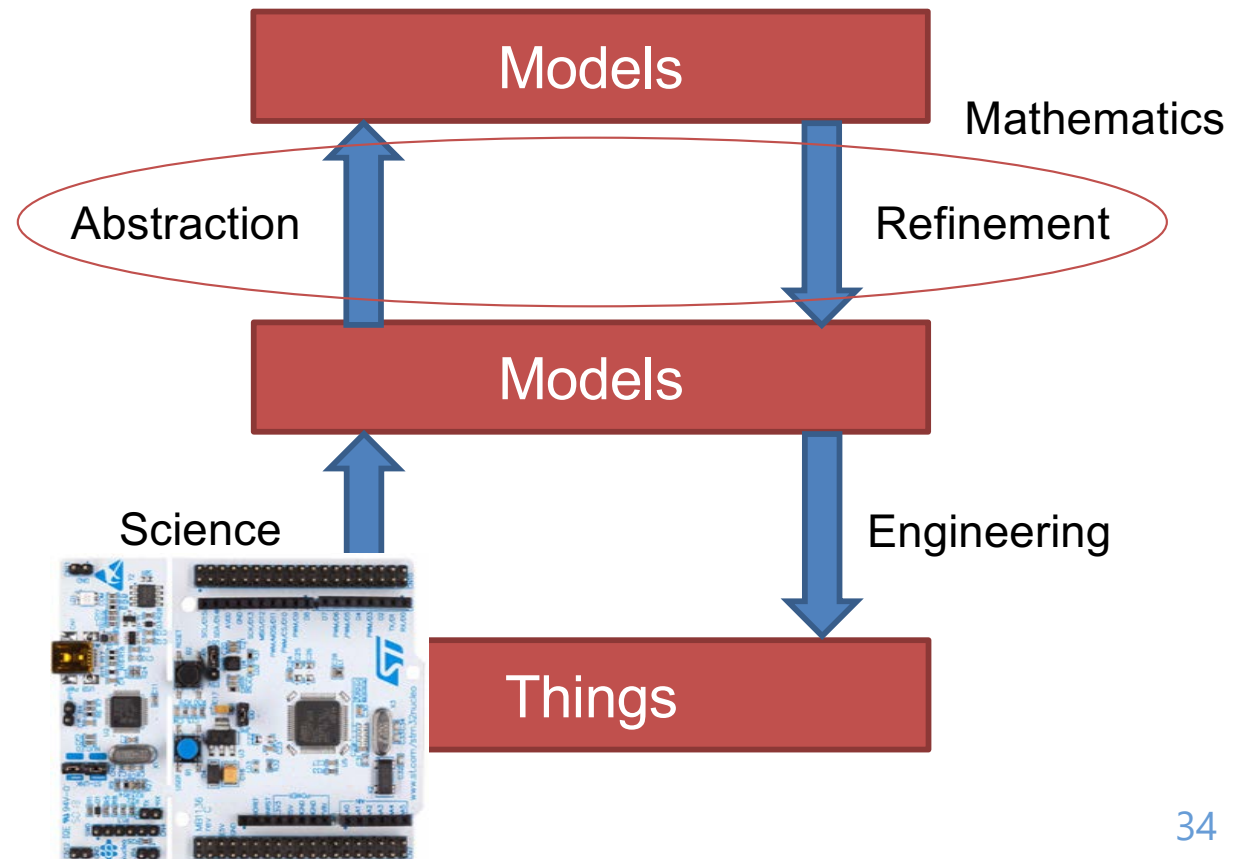
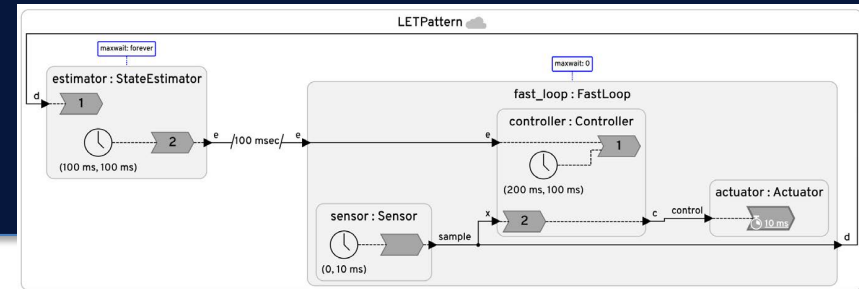


Specifying Timing vs. Discovering It

Most real-time systems development today is based on scientific models, using, e.g., WCET analysis.

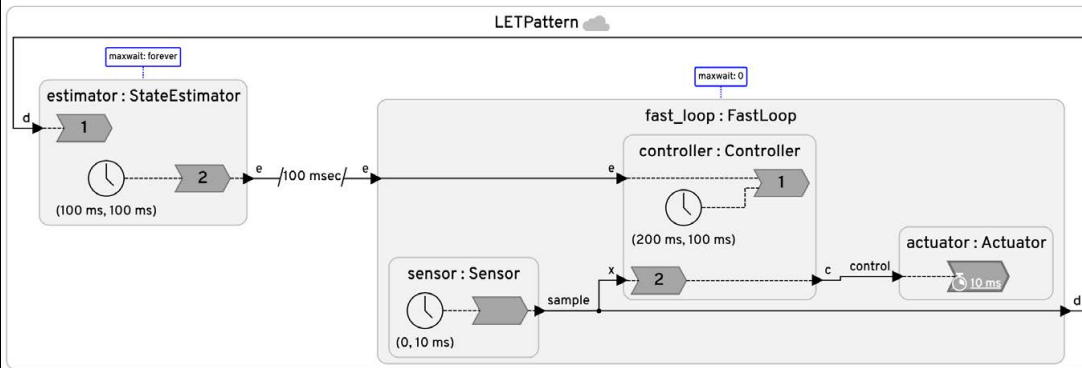
The behavior is revealed by the implementation.

Our approach, instead, specifies timing in engineering models.





The *Model* is Deterministic Under Clearly-Stated Assumptions



Behavior is *defined* in terms of **logical timestamps**. Components (reactors) see events in timestamp order under **clearly-stated timing assumptions**. Faults are detected and handled.

Lingua Franca

Models

C, Python, Rust, ...

Models

ISA

Models

Silicon

Things

lfc

compile

deploy



Determinism? Timing? What about resilience? Adaptability? Faults?

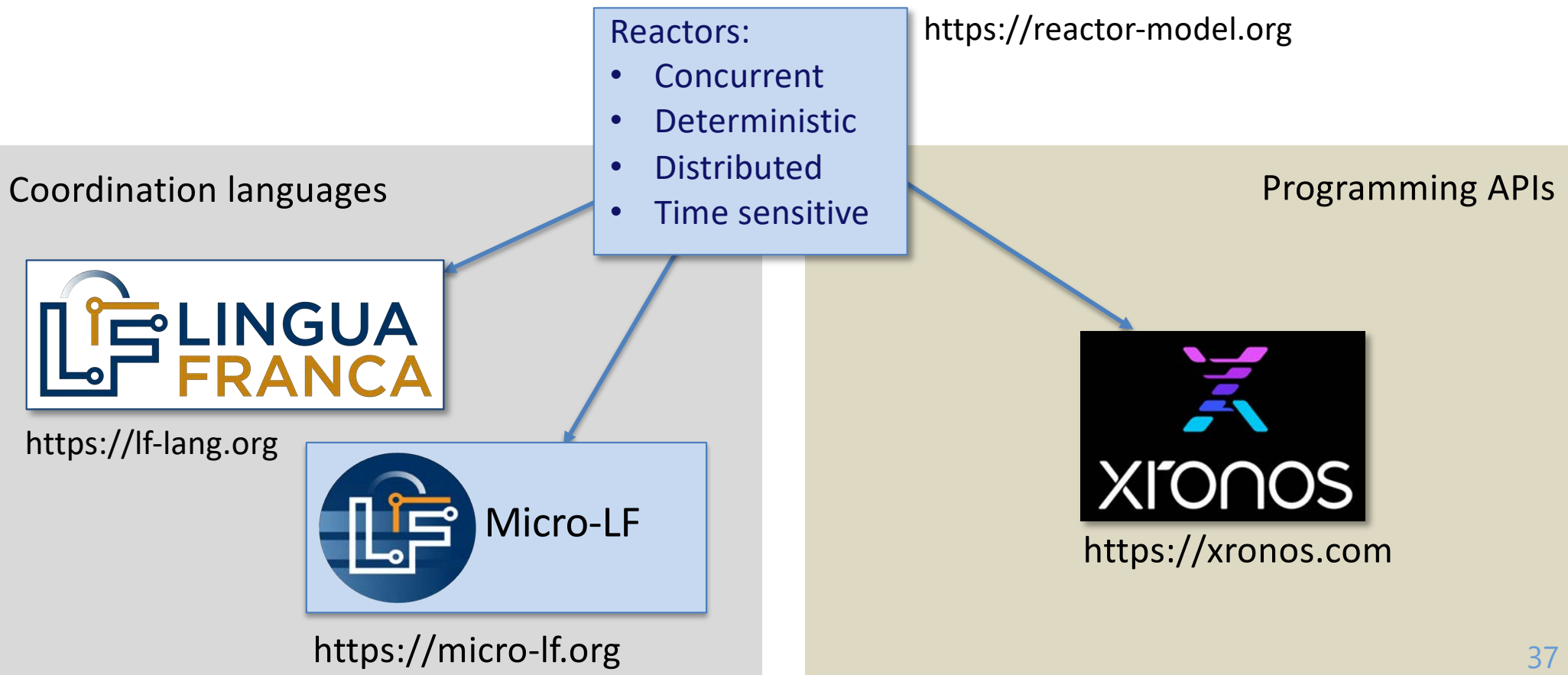
Deterministic and timely models do not eliminate the need for robust, fault-tolerant designs.

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.



Implementations of the Reactor Model



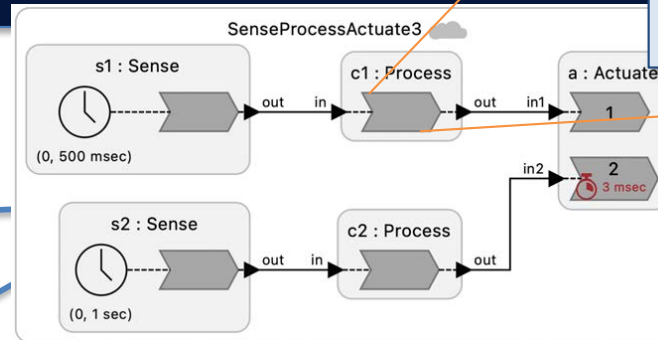


One Program, One Behavior

Behavior is defined and identical across targets, absent faults.

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

Diagram generated & updated as you write code.



Bare
Metal

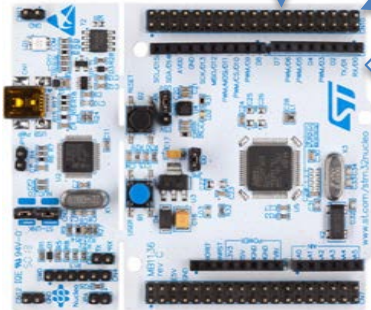
R
IOT

Zephyr®

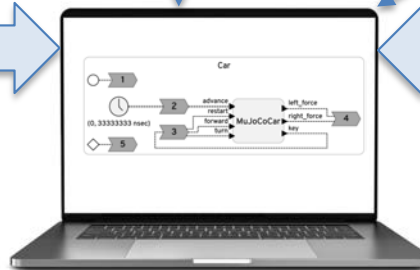
Microsoft
Windows®

Mac OS

Linux



Embedded processors



C, C++, Python, Rust, TypeScript



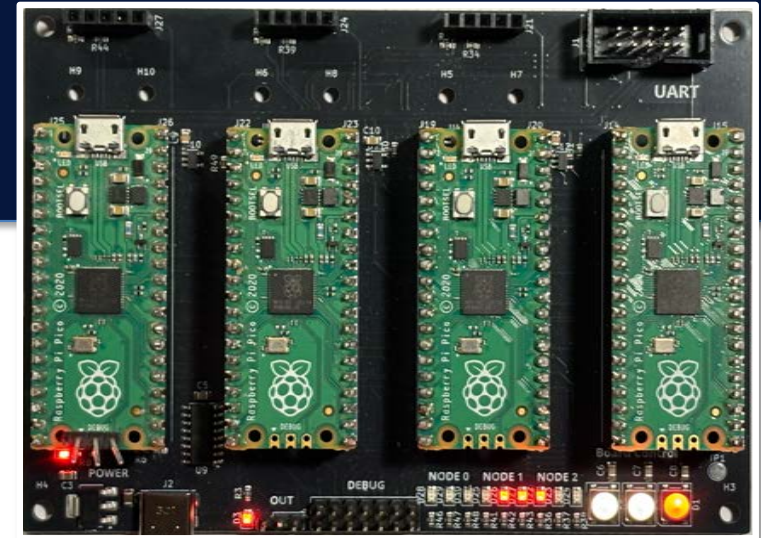
VMs on cloud servers



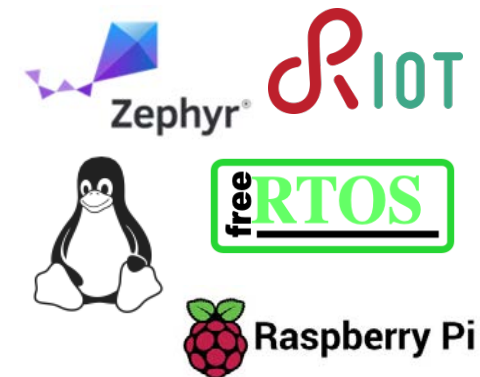
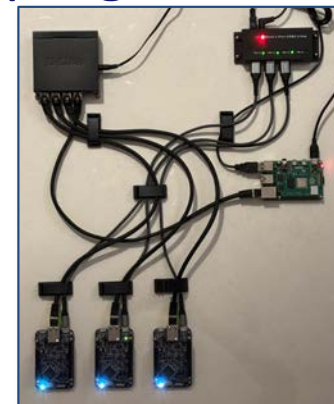
Micro-LF

Reliable real-time embedded software:

- C code generation
- No dynamic memory allocation
- Linux, bare metal, and microkernel
 - RIOT, Zephyr, FreeRTOS
- Deterministic multicore and distributed programming
- Support for transient federates
- Flexible networking
 - Ethernet, Zigbee, Bluetooth, WiFi, LoRa, UART, CoAP



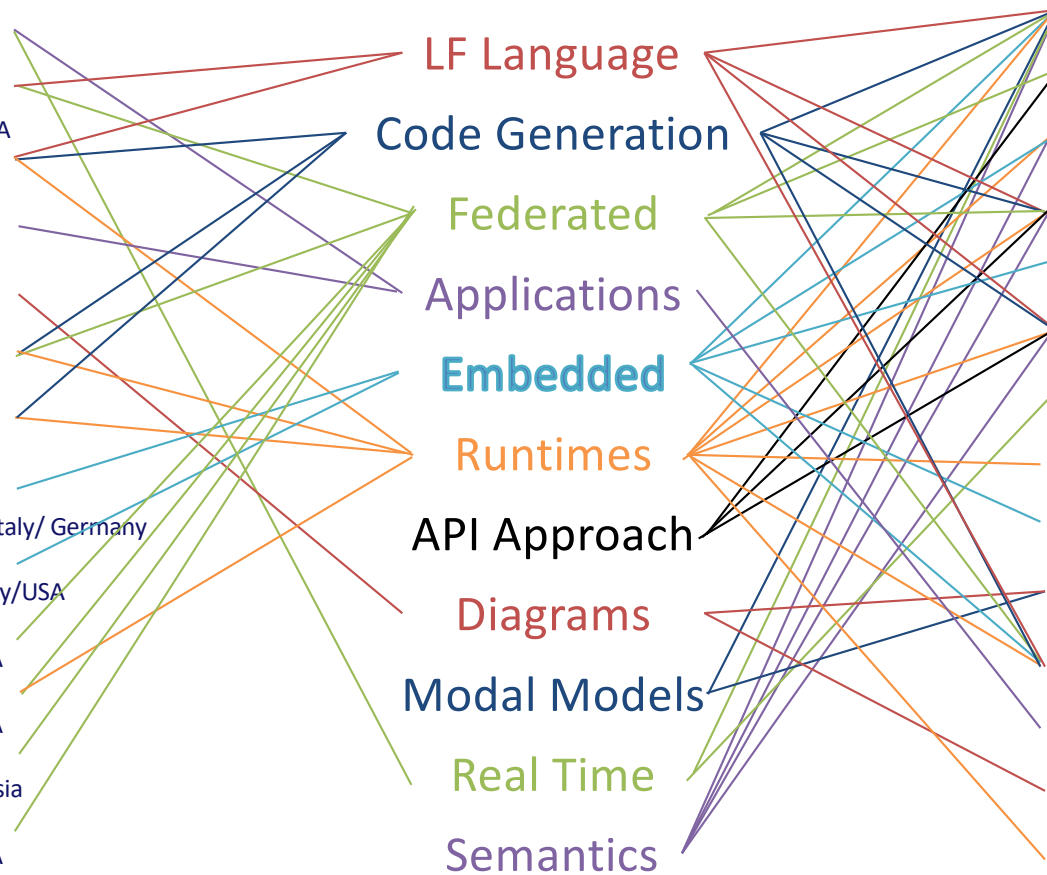
Redundant Picos application
by Tassilo Tanneberger, TU Dresden





Major Contributors to Lingua Franca

- Ravi Akella
 - Denso, USA
- Soroush Bateni
 - UT Dallas, Berkeley, USA
- Jeronimo Castrillon
 - TU Dresden
- Jian-Jia Chen
 - Aachen, Germany
- Soren Domrös
 - Kiel Univ, Germany
- Peter Donovan
 - UC Berkeley, USA
- Clément Fournier
 - TU Dresden, Germany
- Sebastiano Gaiardelli
 - U Verona/TU Munich, Italy/ Germany
- Erling Jellum
 - NTNU/Berkeley, Norway/USA
- Dongha Kim
 - Arizona State Univ, USA
- Hokeun Kim
 - Arizona State Univ, USA
- Chadlia Jerad
 - Univ of Manouba, Tunisia
- Byeonggil Jun
 - Arizona State Univ, USA



- Edward Lee
 - UC Berkeley, USA
- Shulu Li
 - UC Berkeley, USA
- Shaokai Lin
 - UC Berkeley, USA
- Marten Lohstroh
 - UC Berkeley/Xronos Inc., USA
- Magnus Maehlum
 - NTNU, Norway
- Christian Menard
 - TU Dresden/Xronos Inc., Germany
- Francesco Paladino
 - UC Berkeley, USA
- Anirudh Rengarajan
 - UC Berkeley, USA
- Lasse Rosenow
 - HAW Hamberg, Germany
- Alexander Schulz-Rosengarten
 - Kiel Univ, Germany
- Tassilo Tanneberger
 - TU Dresden, Germany
- Harun Teper
 - TU Dortmund, Germany
- Reinhard von Hanxleden
 - Kiel Univ, Germany
- Steven Wong
 - UC Berkeley, USA



Conclusion

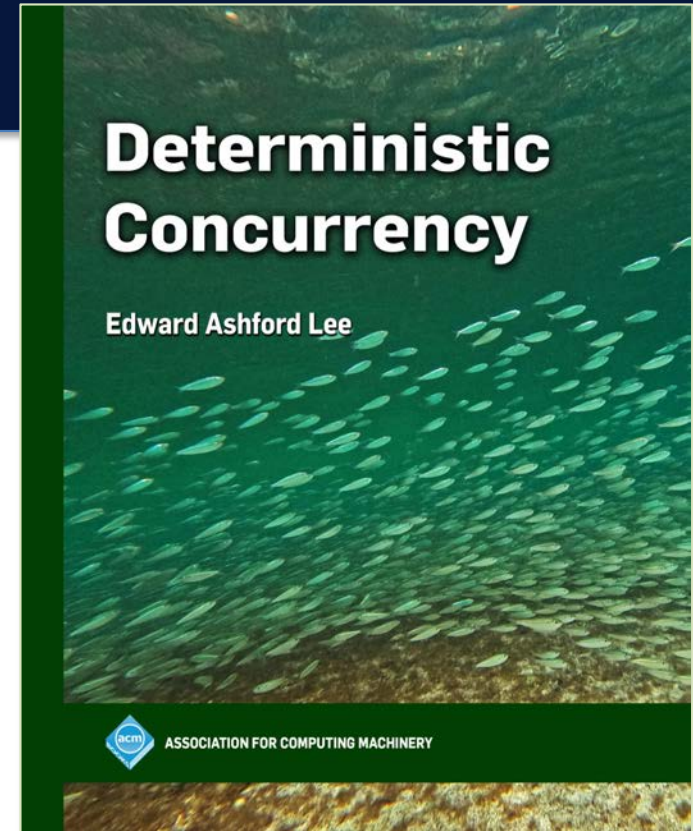
Distributed real-time computing practice today admits more nondeterminism than necessary (e.g. using ROS) and can benefit enormously from more deterministic mechanisms.

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!

Edward Lee, UC Berkeley



Book in progress
Expected in December 2026
<https://ptolemy.org/~eal>