Actors Revisited for Predictable Systems

Edward A. Lee

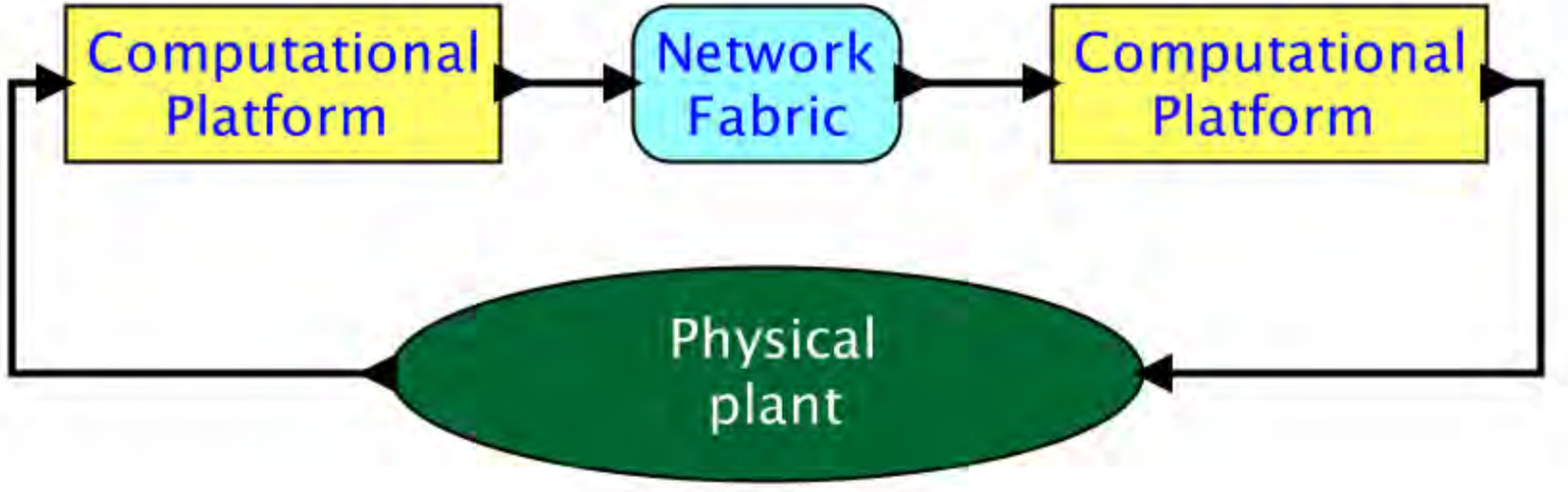
With thanks to:
Marten Lohstroh
Martin Schoeberl
Andrés Goens
Mehrdad Niknami
Christopher Gill
Marjan Sirjani
Matt Weber

Workshop on Programming Languages for Distributed Systems and Distributed Data Management

Dagstuhl, Germany, Oct. 27-31

University of California at Berkeley

Cyber-Physical Systems



Predictability requires determinacy and depends on timing, including execution times and network delays.

A Simple Challenge Problem

An actor that can receive either of two messages:

1. open_door
2. disarm_door

Assume state is closed and armed.

What should it do when it receives a message open_door?

A Simple Challenge Problem

An actor that can receive either of two messages:

1. open_door
2. disarm_door

Assume state is closed and armed.

What should it do when it receives a message open_door?

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

Some Solutions (?)

1. Just open the door.

   How much to test? How much formal verification? How to constrain the design of other actors?

2. Send a message “ok_to_open?” Wait for responses.

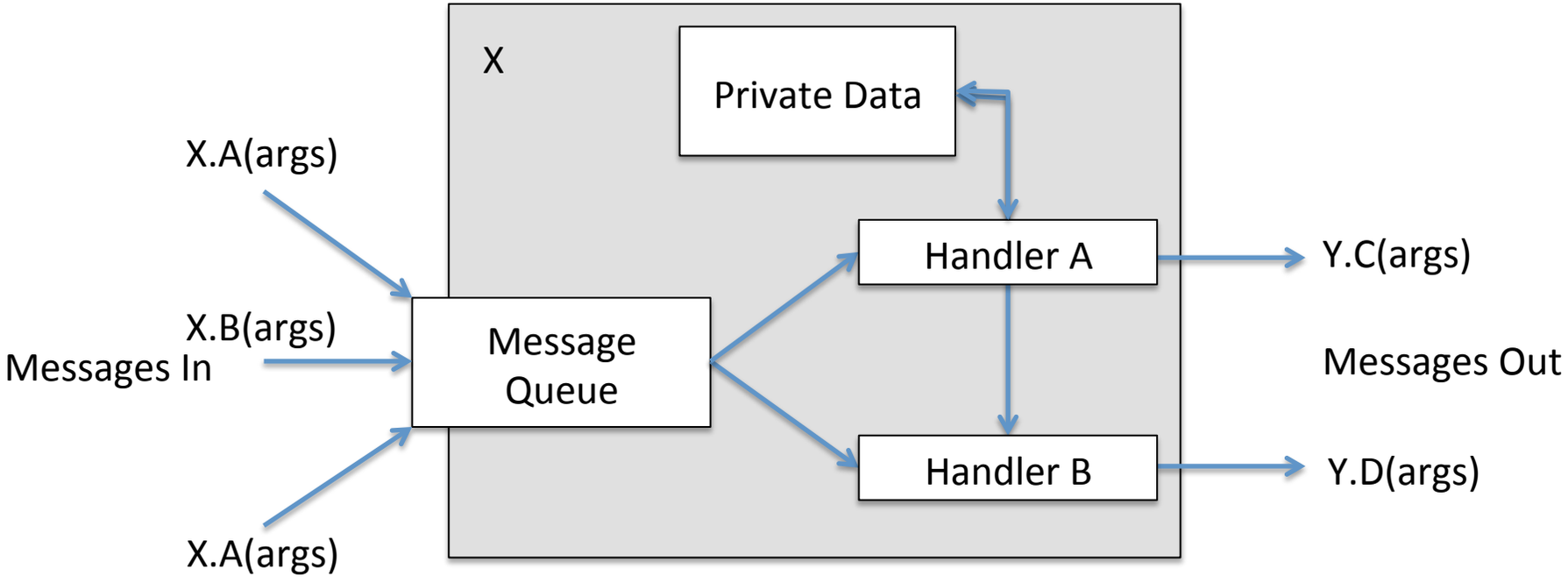
   How many responses? How long to wait? What if a component has failed and never responds?

3. Wait a while and then open.

   How long to wait?

Hewitt/Agha Actors

Data + Message Handlers



[Hewitt, 1977] [Agha, 1986, 1990, 1997]

Pseudo Code

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

What assumptions are needed for it to be safe for the handler to open the door?

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

Pseudo Code

```
Actor Source {
    handler main(){
        x = new Door();
        p = new PassDisarm();
        p.pass();
        x.open_door();
    }
}
```

```
Actor PassDisarm {
    handler pass(Door x){
        x.disarm_door();
    }
}
```

Now what assumptions are needed for it to be safe for the handler to open the door?

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

Fix with formal verification?

One possibility is to formally analyze the system (e.g. using Rebeca). Properties to verify:

1. If Door receives open_door, it will eventually open the door, even if all other components fail.
2. If any component sends disarm_door before any other component sends open_door, then the door will be disarmed before it is opened.

Can these be satisfied?

[Sirjani & Jaghoor, “Ten Years of Analyzing Actors: Rebeca Experience,” 2011]

Fix with formal verification?

One possibility is to formally analyze the system (e.g. using Rebeca). Properties to verify:

1. If Door receives open_door, it will event
   door, even if all other components fail.
2. If any component sends disarm_door before any other component sends open_door, then the door will be disarmed before it is opened

Can th

Makes a distributed-consensus solution challenging.

Requires comparing times of events on distributed platforms in a MoC that lacks time. Nondeterministic interleaving.

[Sirjani & Jagl nce," 2011]

A Challenge with Hewitt Actors

Properties to verify:

1. If Door receives open_door, it will eventually open the door, even if all other components fail.
2. If any component sends disarm_door before any other component sends open_door, then the door will be disarmed before it is opened.

Conjecture: These two cannot be satisfied (for a sufficiently complex program) without additional assumptions (e.g. bounds on network latency and/or clock synchronization error).

Safety in Numbers?

Publish-and-subscribe frameworks have the same flaw:

- ROS
- MQTT
- Microsoft Azure
- Google Cloud Pub/Sub
- XMPP
- DDS
- Amazon SNS
- …

Possible Solutions

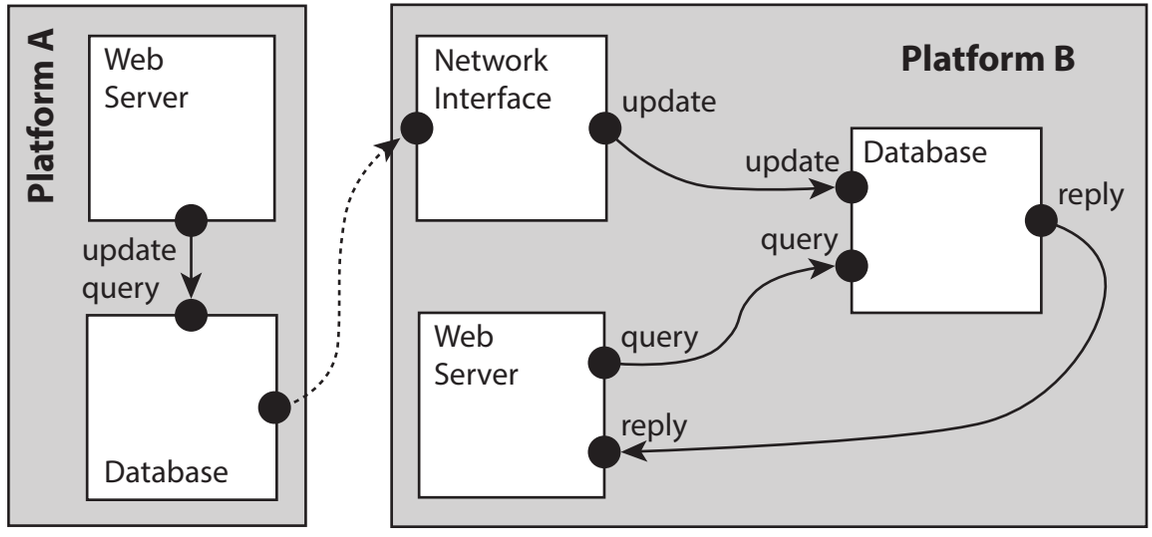
1. Ignore the problem
2. Model timing (Timed Rebeca is useful here)
3. Change the model of computation. E.g.:
   - Dataflow (DF)
   - Kahn Process Networks (KPN)
   - Synchronous/Reactive (SR)
   - Discrete Events (DE)

[Lohstroh and Lee, “Deterministic Actors,” Forum on Design Languages (FDL), 2019]

Example: Google Spanner
A Globally Distributed Database

Spanner's assumptions:

- Bounded clock synchronization error.
- Bounded network latency.
- Negligible execution times.



[Corbet, et al., "Spanner: Google's Globally-Distributed Database," OSDI 2011]

The Essential Change in the MoC (compared to Hewitt Actors)

- Messages are timestamped.
- Messages are processed in timestamp order.
- Simultaneity is well defined.

PTIDES: Deterministic Distributed Real Time Same MoC, five years before Spanner.

This model was introduced in 2007 with applications to cyber-physical systems:

http://ptolemy.org/projects/chess/ptides

in Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 07) , Bellevue, WA, United States.

A Programming Model for Time-Synchronized Distributed Real-Time Systems

Yang Zhao
EECS Department
UC Berkeley

Jie Liu
Microsoft Research
One Microsoft Way

Edward A. Lee
EECS Department
UC Berkeley

At What Cost Determinism?

- Synchronized clocks
  - Becoming ubiquitous
- Bounded network latency
  - Violations are *faults*. They are detectable.
- Bounded execution times
  - Only needed in particular places.
  - Solvable with PRET machines (more later).

What can be verified with the PTIDES/Spanner approach?

1. If Door receives open_door, it will ~~eventually~~ open the door in bounded time, even if all other components fail.
2. If any component sends disarm_door before any other component sends open_door, and the message is received in bounded time, then the door will be disarmed before it is opened.

The first is stronger, the second weaker.

And these properties are satisfied for any program complexity.

[Zhao et al., "A Programming Model for Time-Synchronized Distributed Real-Time Systems," RTAS 2007]

Principle

Use a MoC where:

1. Designing software that satisfies the properties of interest is easy.
2. The implementation of the MoC (the framework) is verifiably correct under reasonable assumptions.

The hard part is 2, where it should be, since that is done once for many applications.

"Keep the hard stuff out of the application logic"

[Sebastian Burckhardt, Oct. 28, 2019]

Today: Lingua Franca

A polyglot meta-language for deterministic, concurrent, time-sensitive systems.

Lingua Franca Wiki

Pages 15

Topics

- Overview
- Language Specification
- Writing Reactors in C
- Accessors Target
- Downloading and Building

Papers

- FDL 2019 paper on Deterministic Actors.
- EMSOFT 2019 work-in-progress paper.
- DAC 2019 paper on Reactors.

https://github.com/icyphy/lingua-franca/wiki

Contents

- Overview
  - Reactors
  - Time
  - Real-Time Systems
  - References
- Language Specification
  - Reactor Block
    - Parameter Declaration
    - State Declaration
    - Input Declaration

Hello World

```
target C;
main reactor HelloWorld {
    timer t;
    reaction(t) {=
        printf("Hello World.\n");
    =}
}
```

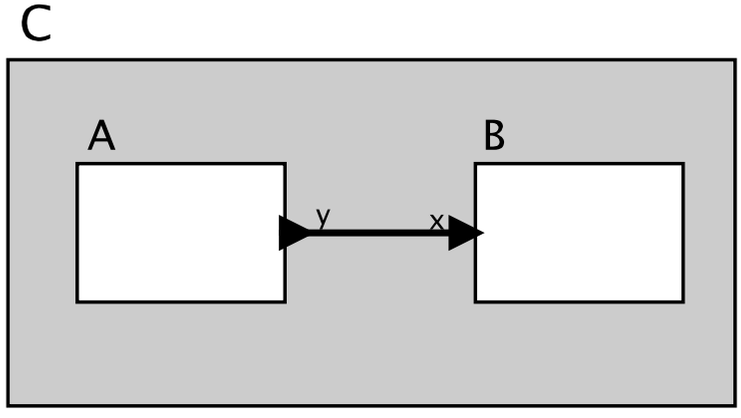
Target language (currently C or JavaScript. Plans for Python, C++, Rust, Java)

Arbitrary code in the target language.

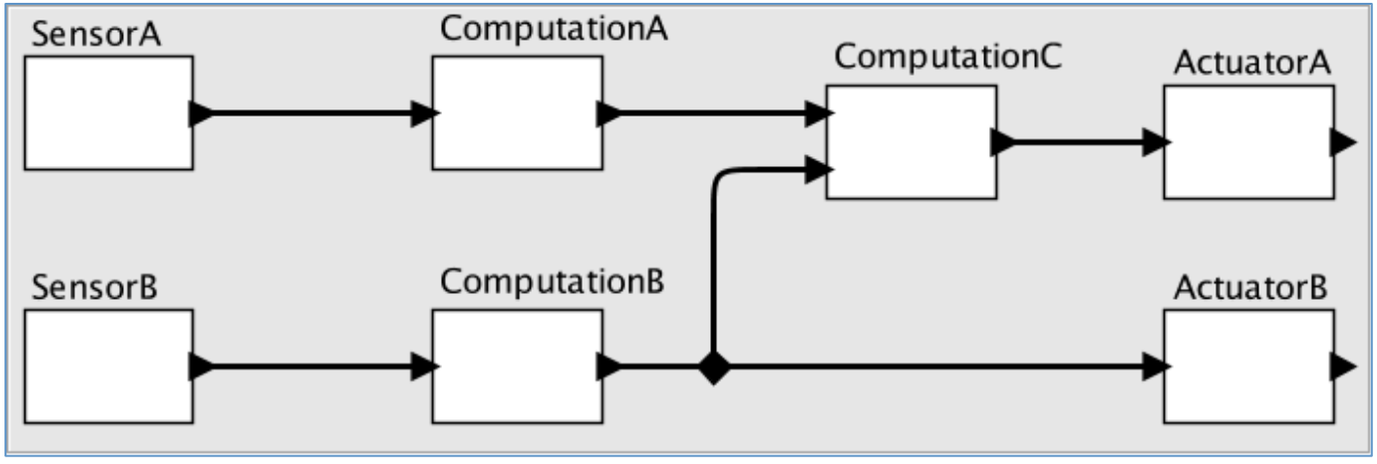
Events of various kinds trigger reactions

Hierarchical Composition and Ports

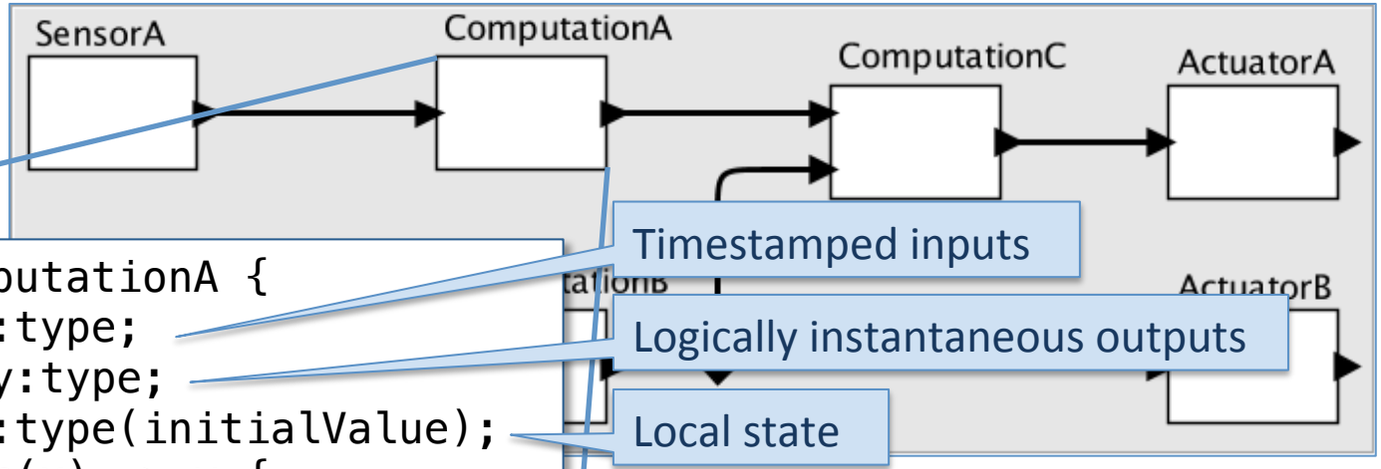
```
reactor A {
  output y;
  ...
}
reactor B {
  input x;
  ...
}
main reactor C {
  a = new A();
  b = new B();
  a.y -> b.x;
}
```



Application Sketch



Reactors




```
reactor ComputationA {
    input x:type;
    output y:type;
    state s:type(initialValue);
    reaction(x) -> y {=
        Target-language code
        referencing x, y, and s.
    =}
}
```

Timestamped inputs

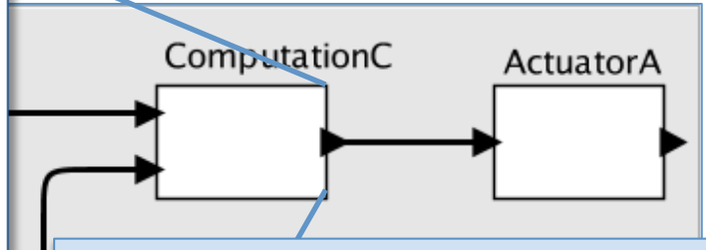
Logically instantaneous outputs

Local state

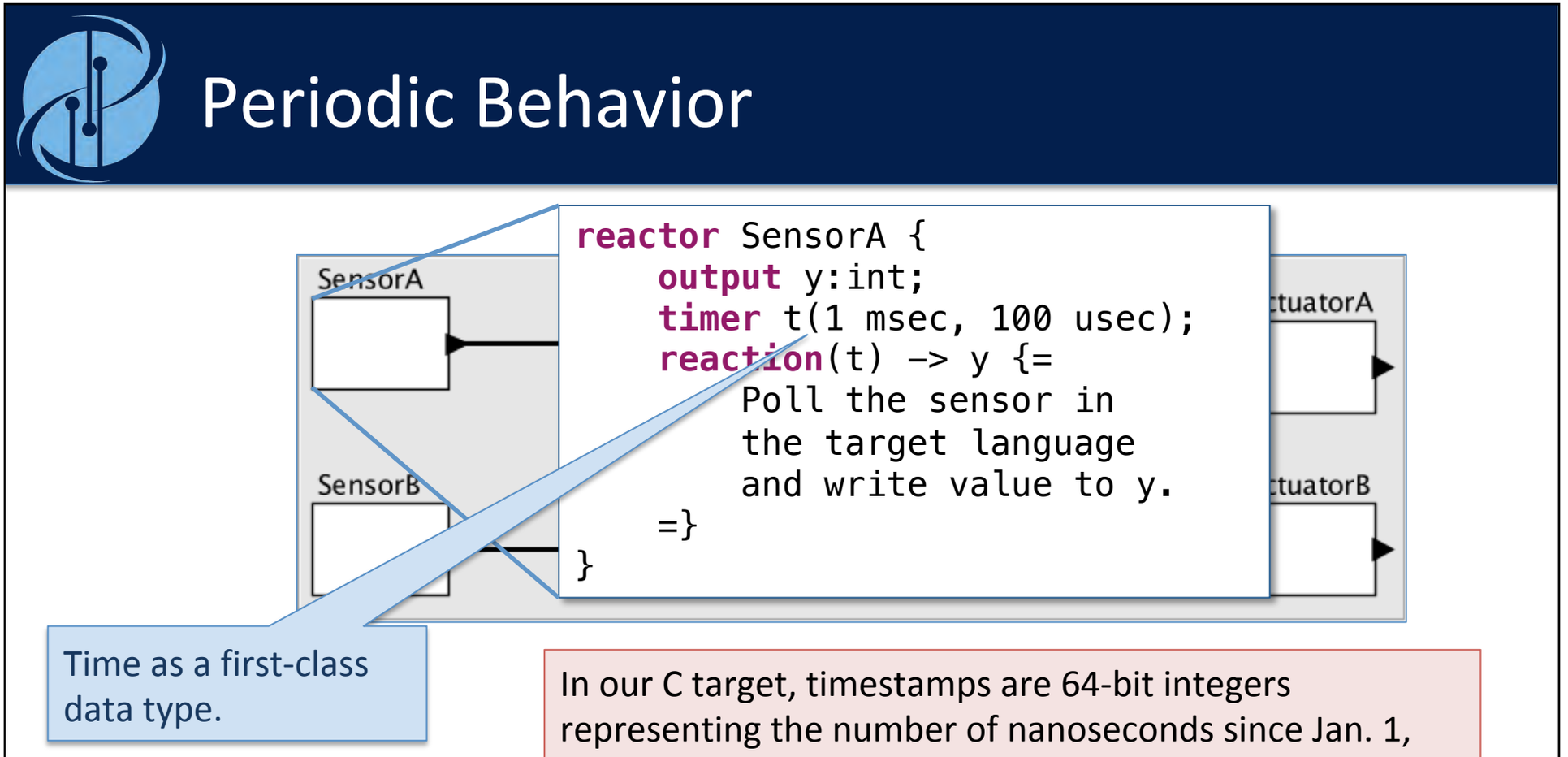
Reaction signature gives trigger(s) and production

Determinism

```
reactor Add {
    input in1:int;
    input in2:int;
    output out:int;
    reaction(in1, in2) -> out {=
        int result = 0;
        if (in1_is_present) {
            result += in1;
        }
        if (in2_is_present) {
            result += in2;
        }
        set(out, result);
    =}
}
```



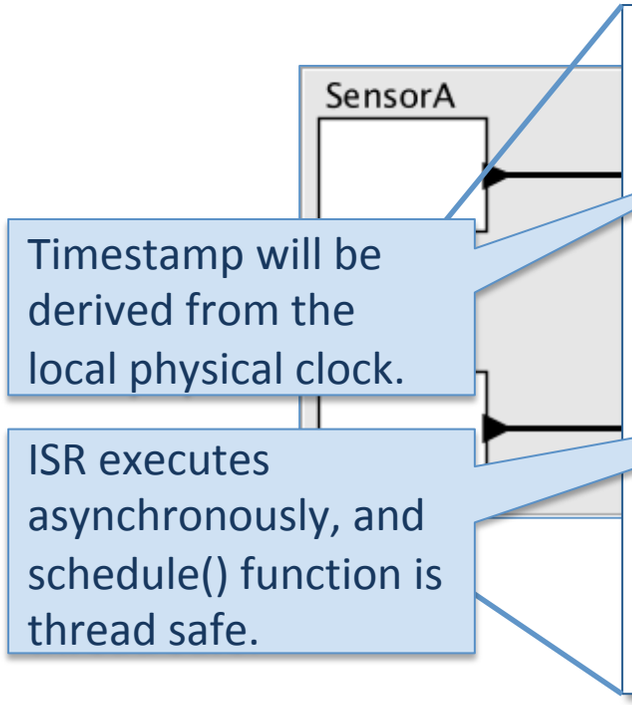
Whether the two triggers are present simultaneously depends only on their timestamps, not on on when they are received nor on where in the network they are sent from.



Periodic Behavior

In our C target, timestamps are 64-bit integers representing the number of nanoseconds since Jan. 1, 1970 (if the platform has a clock) or the number of nanoseconds since starting (if not).

Event-Triggered Behavior



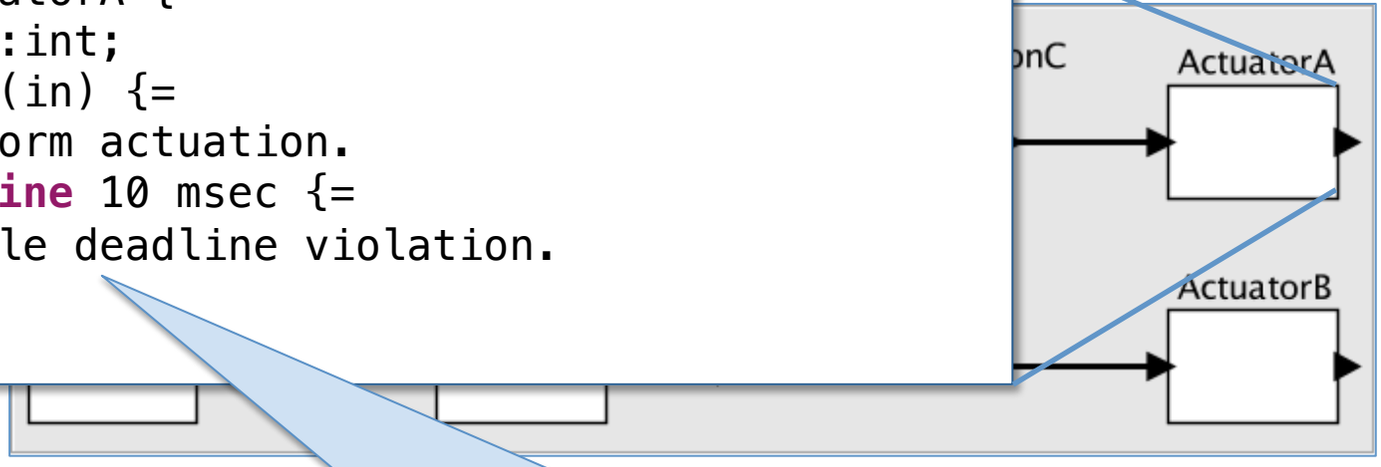
Timestamp will be derived from the local physical clock.

ISR executes asynchronously, and schedule() function is thread safe.

```
reactor SensorB {
    output y:int;
    physical action a:int;
    timer start;
    reaction(start) -> a {=
        Set up an interrupt service
        routine that will call:
        schedule(a, 0, value);
    =}
    reaction(a) -> y {=
        set(y, *(*int)(a->payload));
    =}
}
```

Deadlines

```
reactor ActuatorA {
    input in:int;
    reaction(in) {=
        perform actuation.
    =} deadline 10 msec {=
        handle deadline violation.
    =}
}
```



Deadline is violated if the input d.x triggers more than 10 msec (in physical time) after the timestamp of the input.

Status

https://github.com/icyphy/lingua-franca

Still very early. Only small test cases work.

- Eclipse/Xtext-based IDE
- C and JavaScript targets
- C code runs on Mac, Linux, Windows, and bare iron
- Command-line compiler
- Regression test suite
- Wiki documentation

Performance

Behaviors of the C target in the regression tests running on a 2.6 GHz Intel Core i7 running MacOS:

- Up to 23 million reactions per second (43 ns per).
- Linear speedup on four cores.
- Code size is tens of kilobytes.

Clock Synchronization

- NTP is widely available but not precise enough.
- IEEE 1588 PTP is widely supported in networking hardware but not yet by the OSs.
- Lingua Franca works without clock synchronization by reassigning timestamps to network messages.
- Determinism is preserved within each multicore platform, but not across platforms.

Work in Progress

- EDF scheduling on multicore.
- Distributed execution based on Ptides.
- Targeting PRET machines for real time.
- Formal verification of Lingua Franca apps [with Marjan Sirjani]
- Leverage Google's Protobufs and gRPC.
  - Complex datatypes
  - Polyglot systems

Background Project: PRET Machines

- **PRE**cision-**T**imed processors = **PRET**
- **P**redictable, **RE**peatable **T**iming = **PRET**
- **P**erformance *with* **RE**peatable **T**iming = **PRET**

http://ptolemy.org/projects/chess/pret

```
// Perform the convolution.
for (int i=0; i<10; i++) {
  x[i] = a[i]*b[j-i];
  // Notify listeners.
  notify(x[i]);
}
```

Computing

+

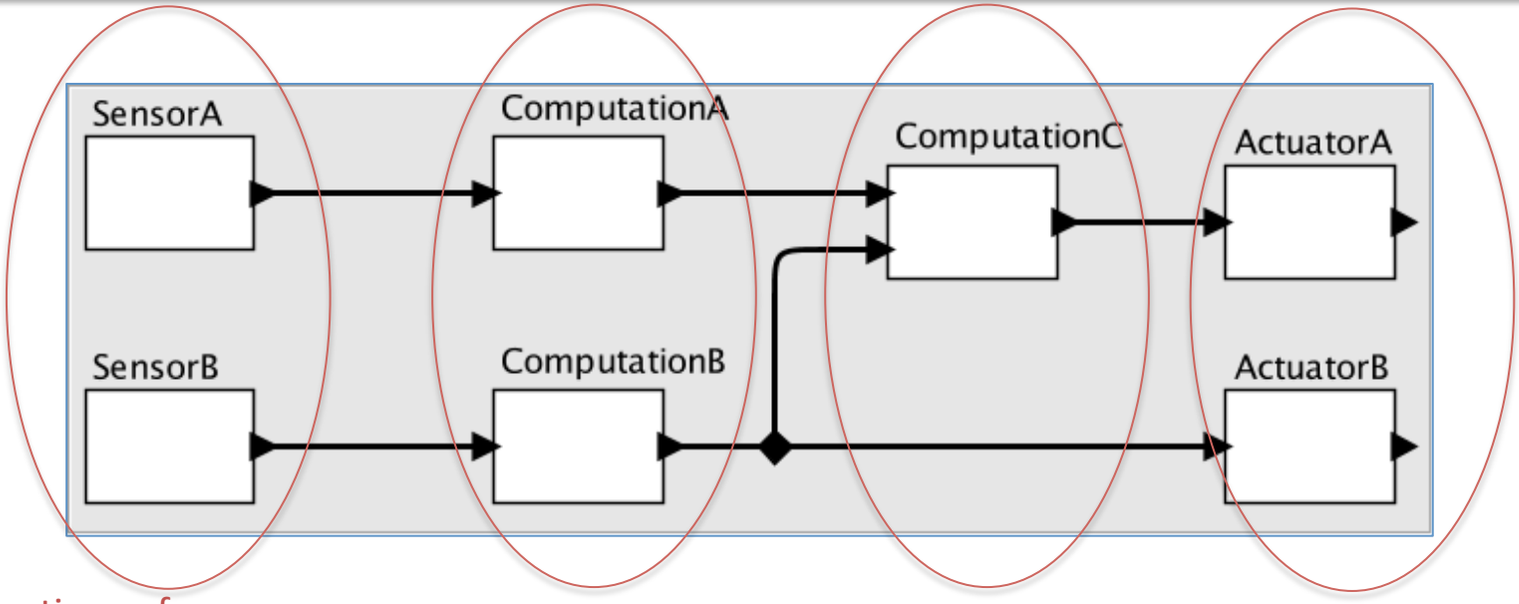
With time

= PRET

PRET targets

With PRET machines, we can deploy systems where deadlines are provably never violated (with explicitly stated assumptions).

Questions That can be Addressed by Lingua-Franca



What combinations of periodic, sporadic, behaviors are feasible?

How do execution times affect feasibility? How can we know execution times?

How do we get repeatable and testable behavior even when communication is across networks?

How do we specify, ensure, and enforce deadlines?

Conclusions

- Hewitt actors are more nondeterministic than they need to be.
- With better infrastructure, a more deterministic model is possible.

See: Lohstroh and Lee, “Deterministic Actors,”
Forum on Design Languages (FDL), 2019