iCyPhy

Introduction to Lingua Franca, a Meta Language for Real-Time Systems

Edward A. Lee

Professor of the Graduate School

Systèmes embarqués et traitement de l'information (SETI)

Université Paris-Saclay

Saclay, France, January 31, 2020

University of California at Berkeley

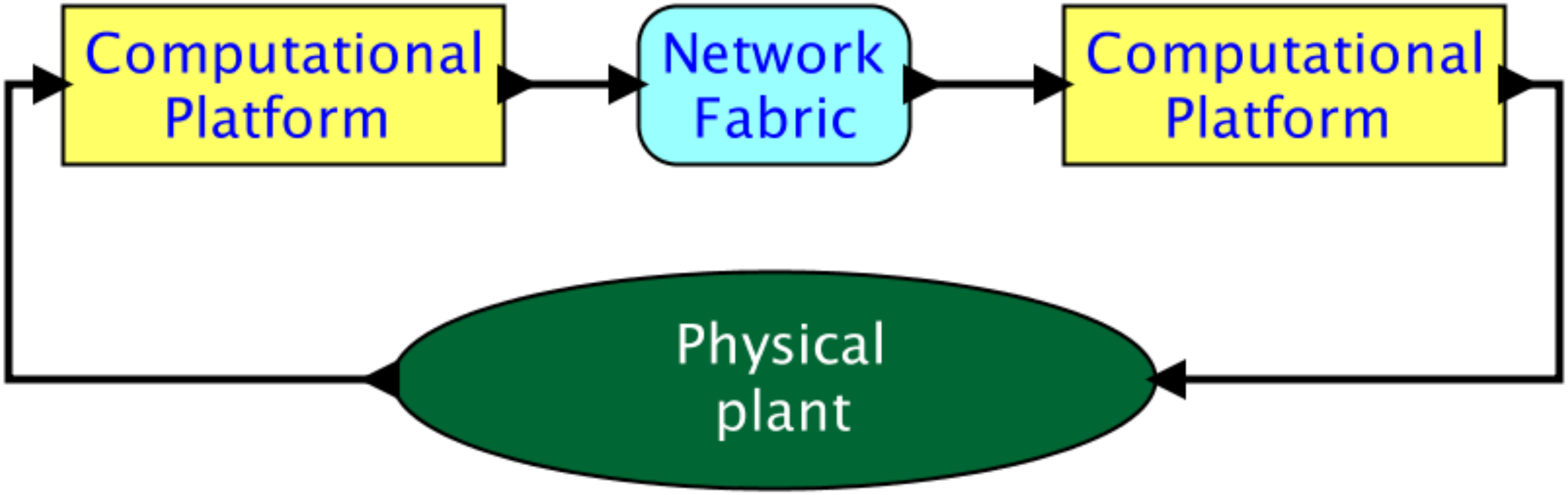
The Slides

Last Week (Updated):

This Week:

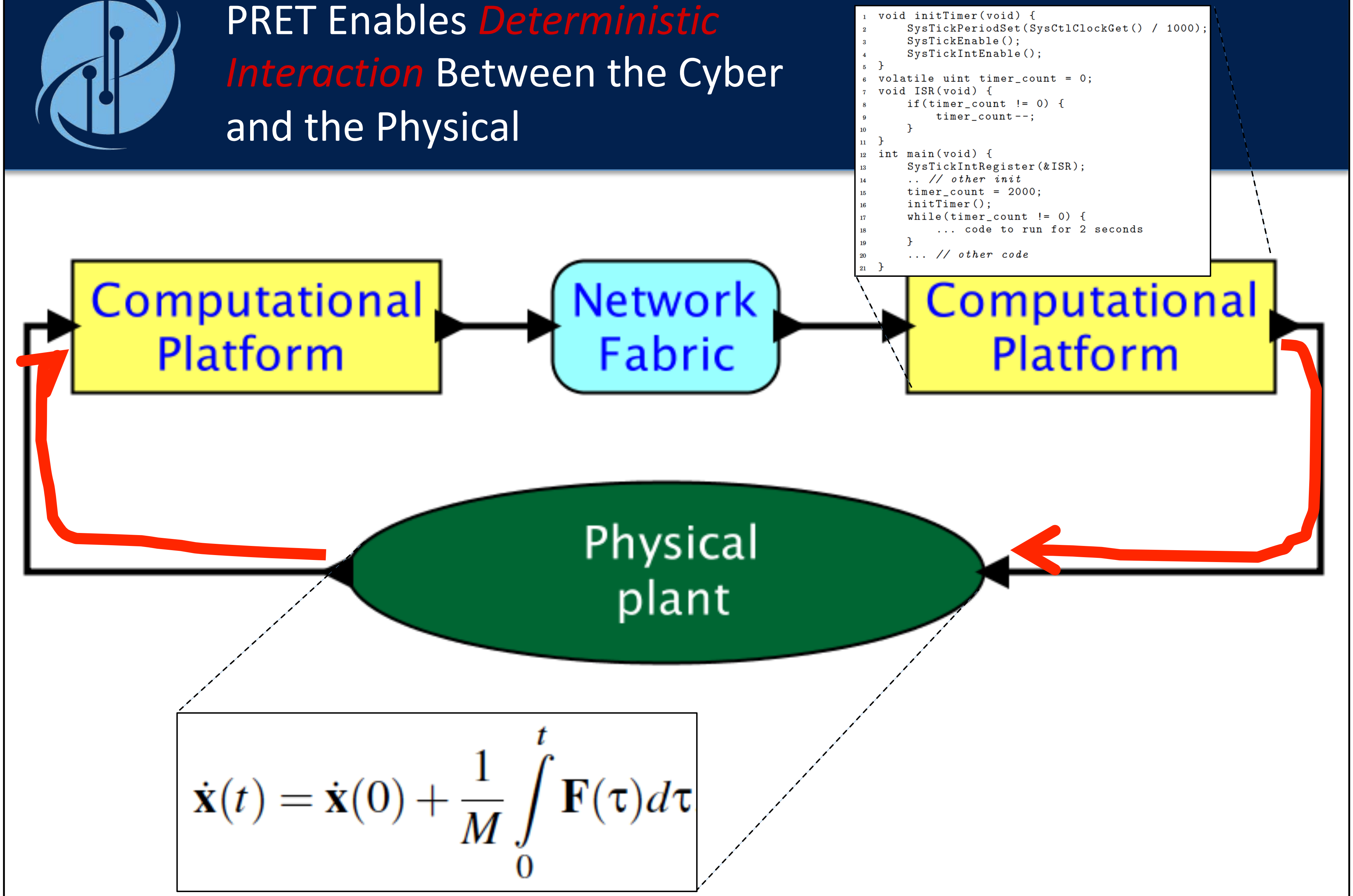
http://ptolemy.org/~eal/presentations/Lee_PrecisionTimedMicroprocessors_Saclay.pdf
http://ptolemy.org/~eal/presentations/Lee_LinguaFranca_Saclay.pdf

Cyber Physical Systems



The major challenge: **Integrating complex subsystems** with adequate **reliability**, **repeatability**, and **testability**.

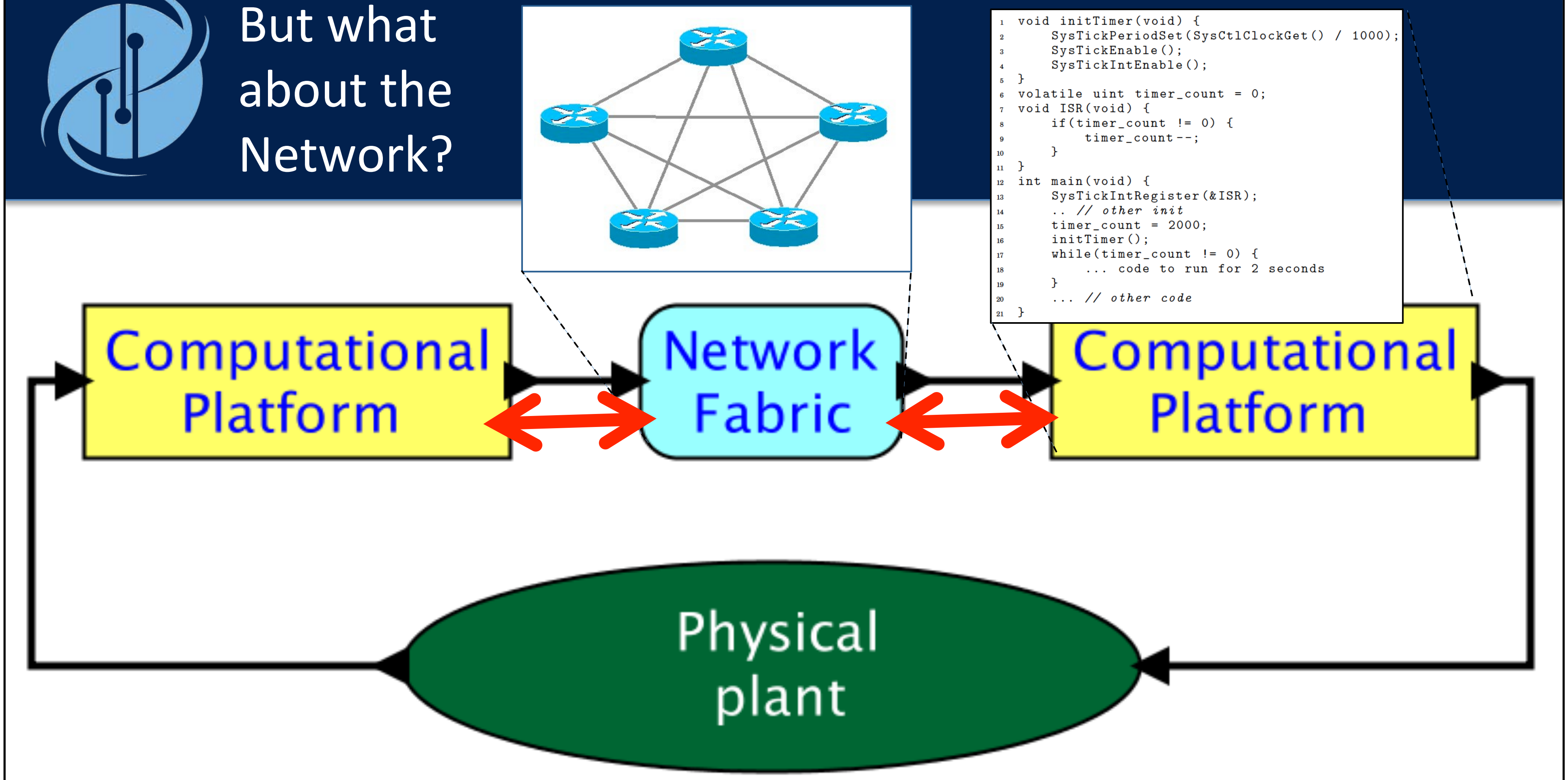
PRET Enables *Deterministic Interaction* Between the Cyber and the Physical



```
void initTimer(void) {
    SysTickPeriodSet(SysCtlClockGet() / 1000);
    SysTickEnable();
    SysTickIntEnable();
}
volatile uint timer_count = 0;
void ISR(void) {
    if(timer_count != 0) {
        timer_count--;
    }
}
int main(void) {
    SysTickIntRegister(&ISR);
    .. // other init
    timer_count = 2000;
    initTimer();
    while(timer_count != 0) {
        ... code to run for 2 seconds
    }
    ... // other code
}
```

$$\dot{\mathbf{x}}(t) = \dot{\mathbf{x}}(0) + \frac{1}{M}\int_{0}^{t} \mathbf{F}(\tau)d\tau$$

But what about the Network?



```
void initTimer(void) {
    SysTickPeriodSet(SysCtlClockGet() / 1000);
    SysTickEnable();
    SysTickIntEnable();
}
volatile uint timer_count = 0;
void ISR(void) {
    if(timer_count != 0) {
        timer_count--;
    }
}
int main(void) {
    SysTickIntRegister(&ISR);
    .. // other init
    timer_count = 2000;
    initTimer();
    while(timer_count != 0) {
        ... code to run for 2 seconds
    }
    ... // other code
}
```

We have also developed deterministic models for distributed real-time software, using a technique called PTIDES and a language called Lingua Franca.

Outline

- Motivating Problems
- Why Existing Methods Fall Short
- Ports, Hierarchy, Models of Computation
- The Lingua Franca Language
- Distributed Execution
- Clock Synchronization
- Conclusion

A Simple Challenge Problem

An actor or service that can receive either of two messages:

1. “open”
2. “disarm”

Assume state is closed and armed.

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

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

A Simple Challenge Problem

An actor or service that can receive either of two messages:

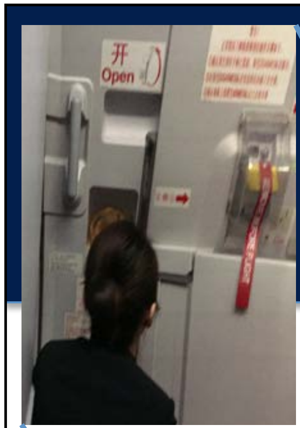
1. “open”
2. “disarm”

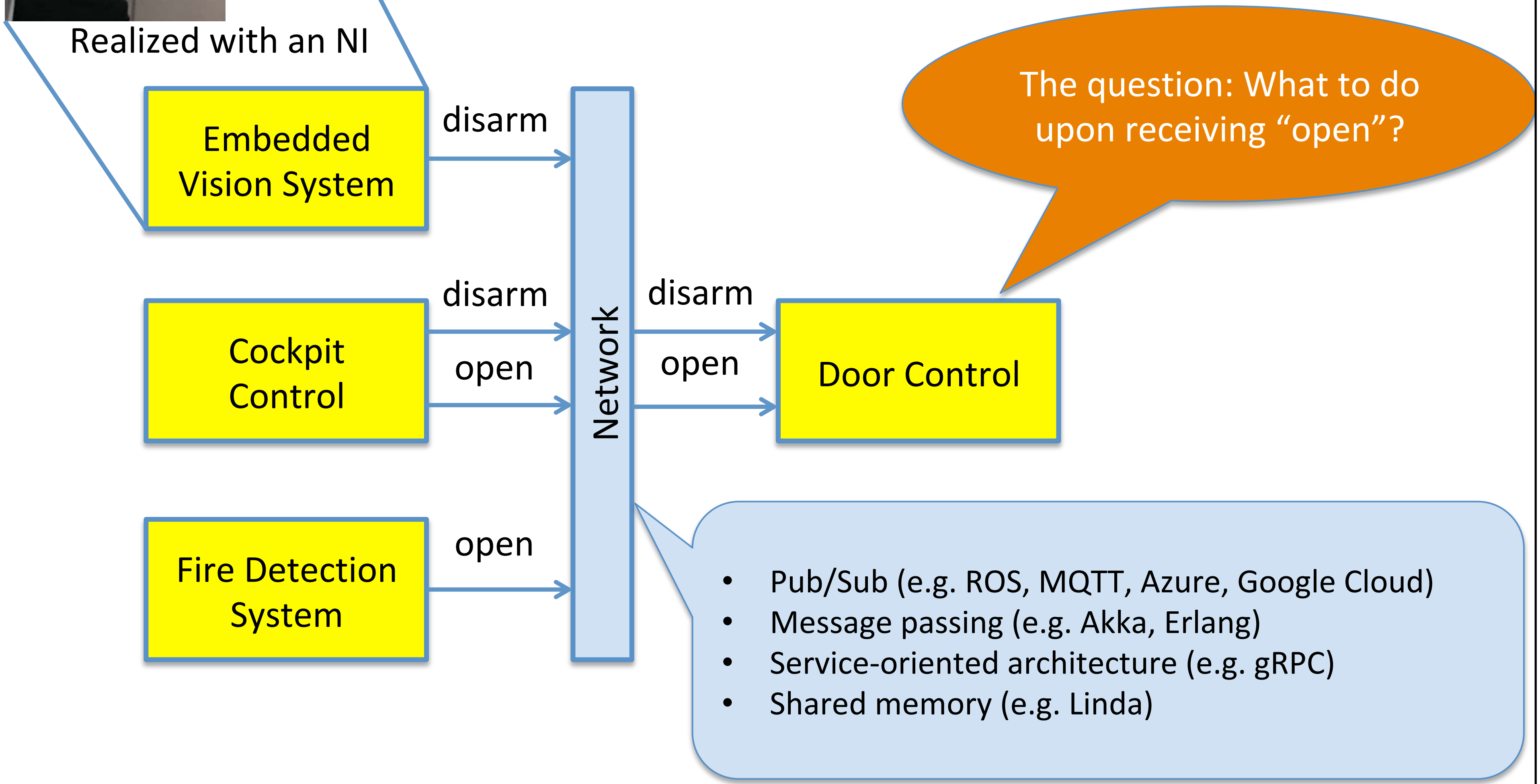
Assume state is closed and armed.

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

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

Possible Architectures




Realized with an NI

The question: What to do upon receiving “open”?

- Pub/Sub (e.g. ROS, MQTT, Azure, Google Cloud)
- Message passing (e.g. Akka, Erlang)
- Service-oriented architecture (e.g. gRPC)
- Shared memory (e.g. Linda)

Some Solutions (?)

1. Just open the door.

   How much to test? How much formal verification? How to constrain the design of other components? The network?

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?

Better go read all of Lamport’s papers.

Fix with formal verification?

One possibility is to formally analyze the system.

Properties to verify:

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

Can these be satisfied?

Fix with formal verification?

One possibility is to formally analyze the s

Properties to verify:

1. If Door receives “open,” it will eventu even if all other components fail.
2. If any component sends “disarm” before any other component sends “open,” then the door will be disarmed before it is opened.

Can these be satisfied?

Makes a distributed-consensus solution challenging.

Requires comparing times of events on distributed platforms in a model of computation that lacks time.

Can these properties be satisfied?

Properties to verify:

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

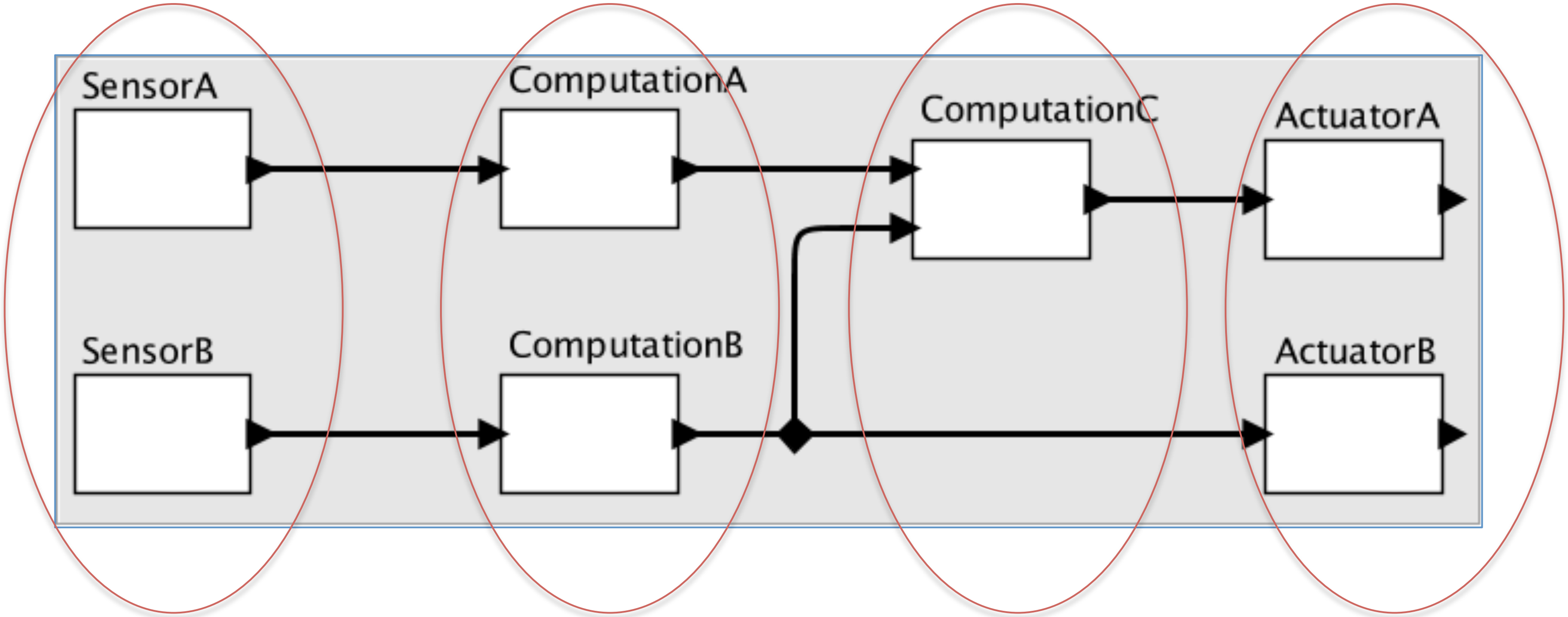
Conjecture: These two cannot be satisfied (for a sufficiently complex program) without additional assumptions (e.g. bounds on network latency and/or clock synchronization).

Possible Solutions

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

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

A Broader Set of Questions



What combinations of periodic, sporadic, arrival curve behaviors are manageable?

How do execution and communication times affect feasibility? How can we know these times?

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

How do we specify, ensure, and enforce deadlines?

Outline

- Motivating Problems
- Why Existing Methods Fall Short
- Ports, Hierarchy, Models of Computation
- The Lingua Franca Language
- Distributed Execution
- Clock Synchronization
- Conclusion

Popular Techniques

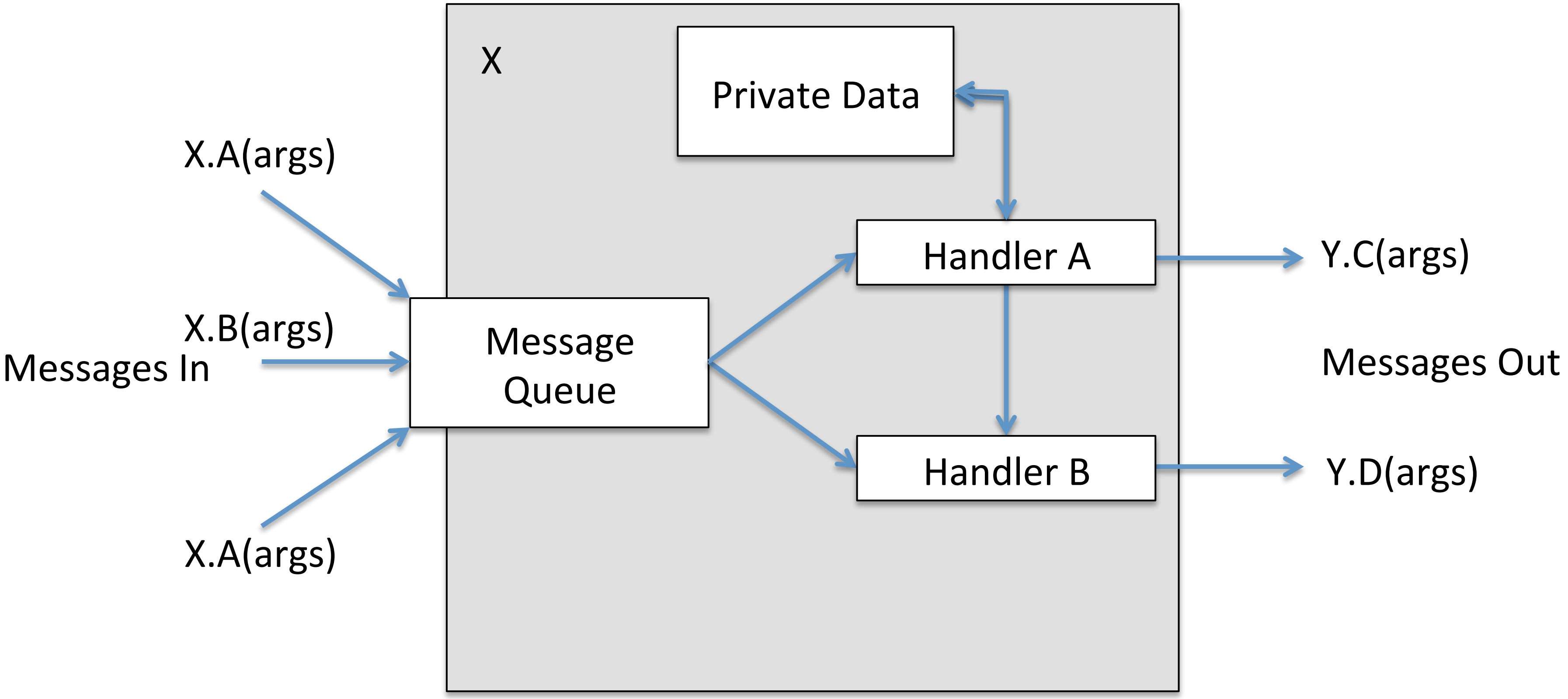
- Publish and Subscribe
  - ROS, MQTT, DDS, Azure, Google Cloud
- Actors
  - Akka, Erlang, Orleans,Rebeca, Scala …
- Service-oriented architecture
  - gRPC, Bond, Thrift, …
- Shared memory
  - Linda, pSpaces, …

Actors, Loosely

Actors are concurrent objects that communicate by sending each other messages.

Hewitt/Agha Actors

Data + Message Handlers



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

Example

An actor with simple operations on its state:

```
Actor Foo {
   int state = 1;
   handler double(){
      state *= 2;
   }
   handler increment(arg){
      state += arg;
      print state;
   }
}
```

Example

An actor that uses actor Foo:

```
Actor Bar {
   handler main(){
      Foo x = new Foo();
      x.double();
      x.increment(1);
   }
}
```

Semantics is “send and forget.”

Composition

```
Actor Bar {
    handler main(){
        Foo x = new Foo();
        x.double();
        x.increment(1);
    }
}
```

```
Actor Foo {
    int state = 1;
    handler double(){
        state *= 2;
    }
    handler increment(arg){
        state += arg;
        print state;
    }
}
```

What is printed?

Pass-Through Actor

Baz: Given an actor of type Foo, send it “double”:

```
Actor Baz {
   handler pass(Foo x){
      x.double();
   }
}
```

New Composition

```
Actor Bar {
    handler main(){
        Foo x = new Foo();
        Baz z = new Baz();
        z.pass(x);
        x.increment(1);
    }
}
```

```
Actor Baz {
    handler pass(Foo x){
        x.double();
    }
}
```

What is printed?

```
Actor Foo {
    int state = 1;
    handler double(){
        state *= 2;
    }
    handler increment(arg){
        state += arg;
        print state;
    }
}
```

Aircraft Door Using Actors

```
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(){
        …
    }
}
```

Aircraft Door Using Actors

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

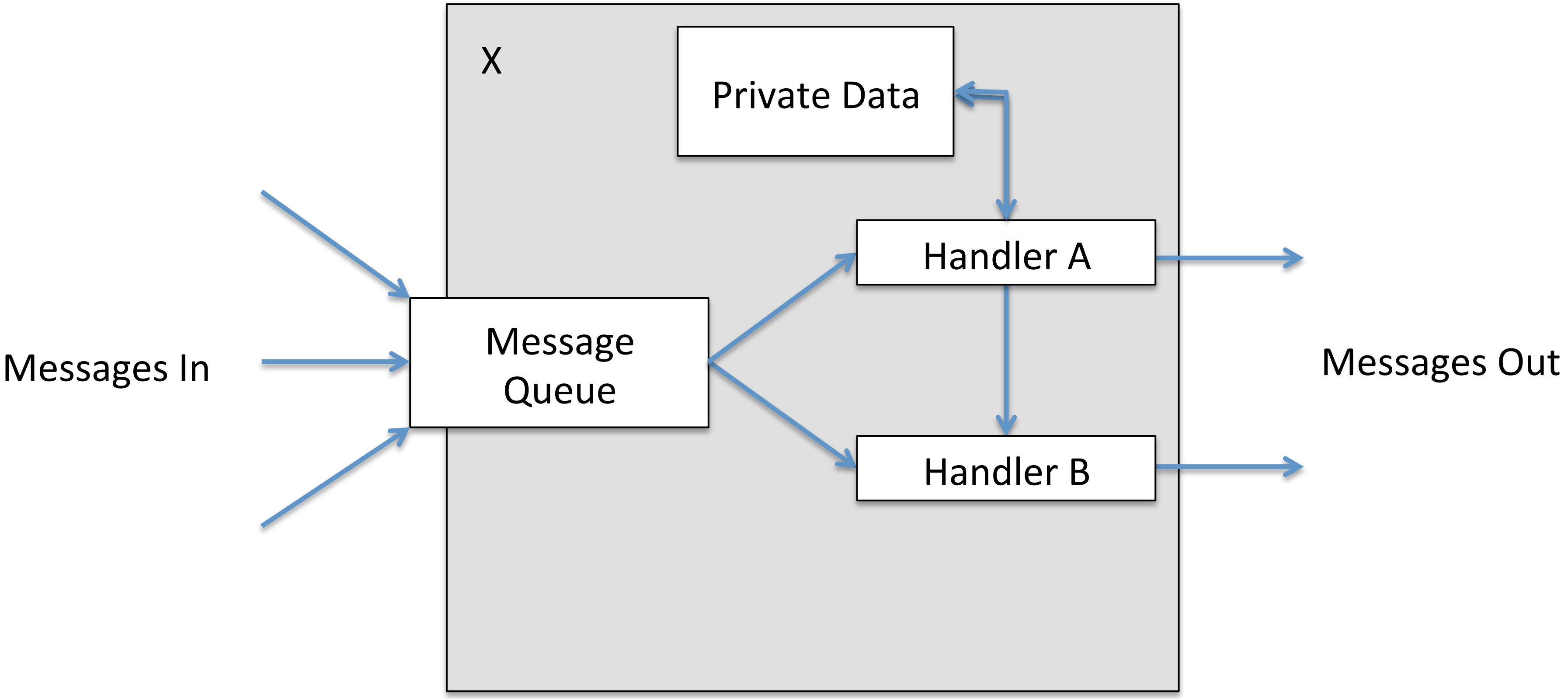
```
Actor PassDisarm {
    handler pass(Door x){
        x.disarm_door();
    }
}
```

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(){
        …
    }
}
```

Hewitt/Agha Actors are Not Predictable

Messages are handled in nondeterministic order.



Aside: Innovation in Ray

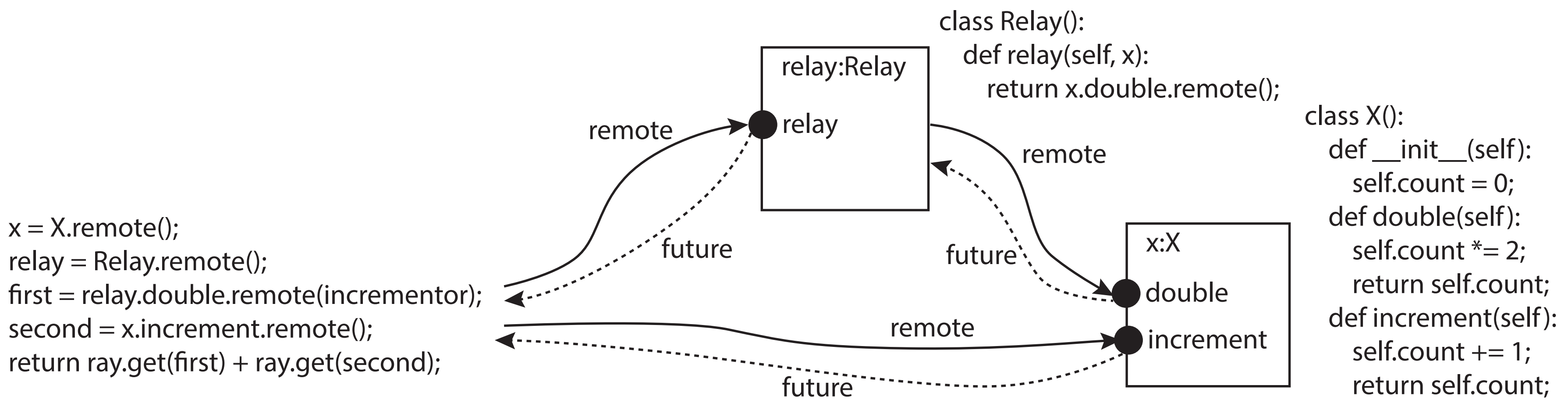
[Moritz, et al. 2017]

Messages can return “futures”:

```
Actor Bar {
    handler main(){
        Foo x = new Foo();
        Future a = x.double();
        Future b = x.increment(1);
        print a.get() + b.get();
    }
}
```

Semantics is still “send and forget,” but later remember.

Unexpected Nondeterminism Example from Ray



```
x = X.remote();
relay = Relay.remote();
first = relay.double.remote(incrementor);
second = x.increment.remote();
return ray.get(first) + ray.get(second);
```

```
class Relay():
  def relay(self, x):
    return x.double.remote();
```

```
class X():
  def __init__(self):
    self.count = 0;
  def double(self):
    self.count *= 2;
    return self.count;
  def increment(self):
    self.count += 1;
    return self.count;
```

The Relay actor is the actor version of a "no op," but it makes the program nondeterministic.

[Moritz, et al., "Ray: A Distributed Framework for Emerging AI Applications" arXiv, 2018]

Safety in Numbers?

Publish-and-subscribe (Pub/Sub) frameworks and Service-Oriented Architectures (SOA) have the same flaw:

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

One Solution:
Analyze and Use Dependencies

```
Actor Bar {
    handler main(){
        Foo x = new Foo();
        Baz z = new Baz();
        z.pass(x);
        x.increment(1);
    }
}
```

```
Actor Baz {
    handler pass(Foo x){
        x.double();
    }
}
```

But how? Where is the dependence graph?

```
Actor Foo {
    int state = 1;
    handler double(){
        state *= 2;
    }
    handler increment(arg){
        state += arg;
        print state;
    }
}
```

One Solution:
Analyze and Use Dependencies

```
Actor Bar {
    handler main(){
        Foo x = new Foo();
        Baz z = new Baz();
        z.pass(x);
        x.increment(1);
    }
}
```

```
Actor Baz {
    handler pass(Foo x){
        if (something) {
            x.double();
        }
    }
}
```

And what if the dependence graph is data dependent?

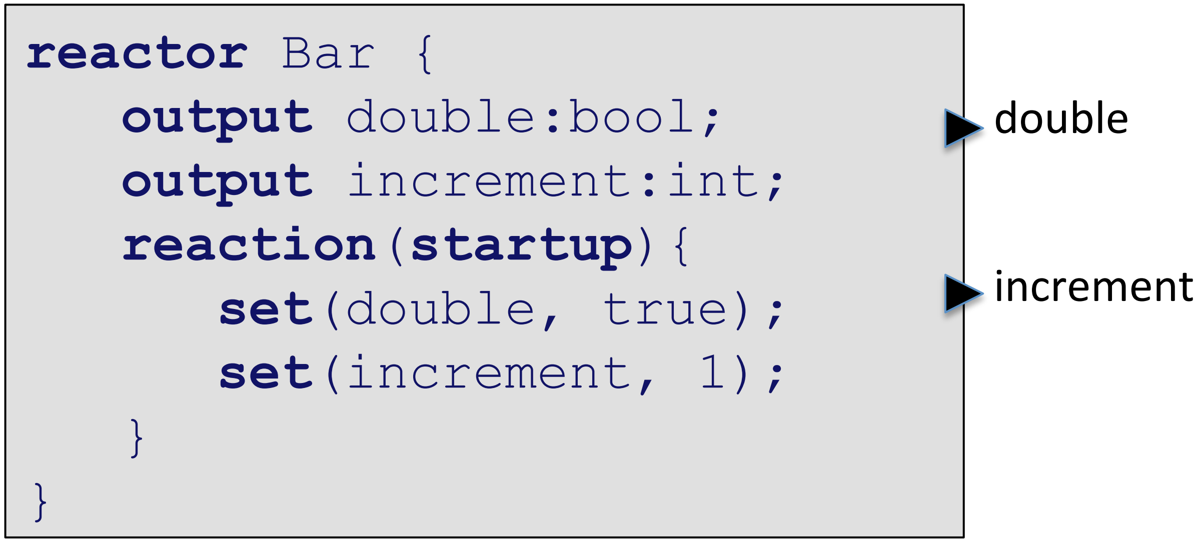
```
Actor Foo {
    int state = 1;
    handler double(){
        state *= 2;
    }
    handler increment(arg){
        state += arg;
        print state;
    }
}
```

Outline

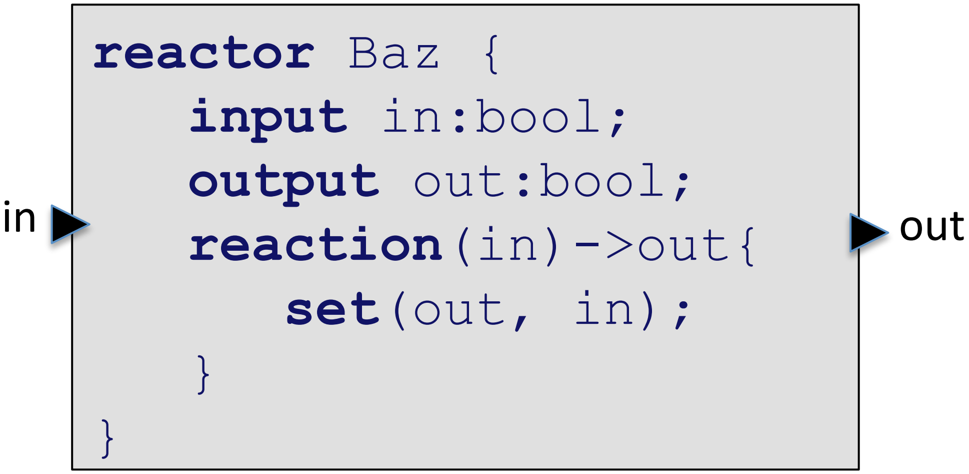
- Motivating Problems
- Why Existing Methods Fall Short
- Ports, Hierarchy, Models of Computation
- The Lingua Franca Language
- Distributed Execution
- Clock Synchronization
- Conclusion

Part 1 of our Solution: Ports

Instead of referring to other actors, an actor refers to its own ports.



```
reactor Bar {
    output double:bool;
    output increment:int;
    reaction(startup){
        set(double, true);
        set(increment, 1);
    }
}
```

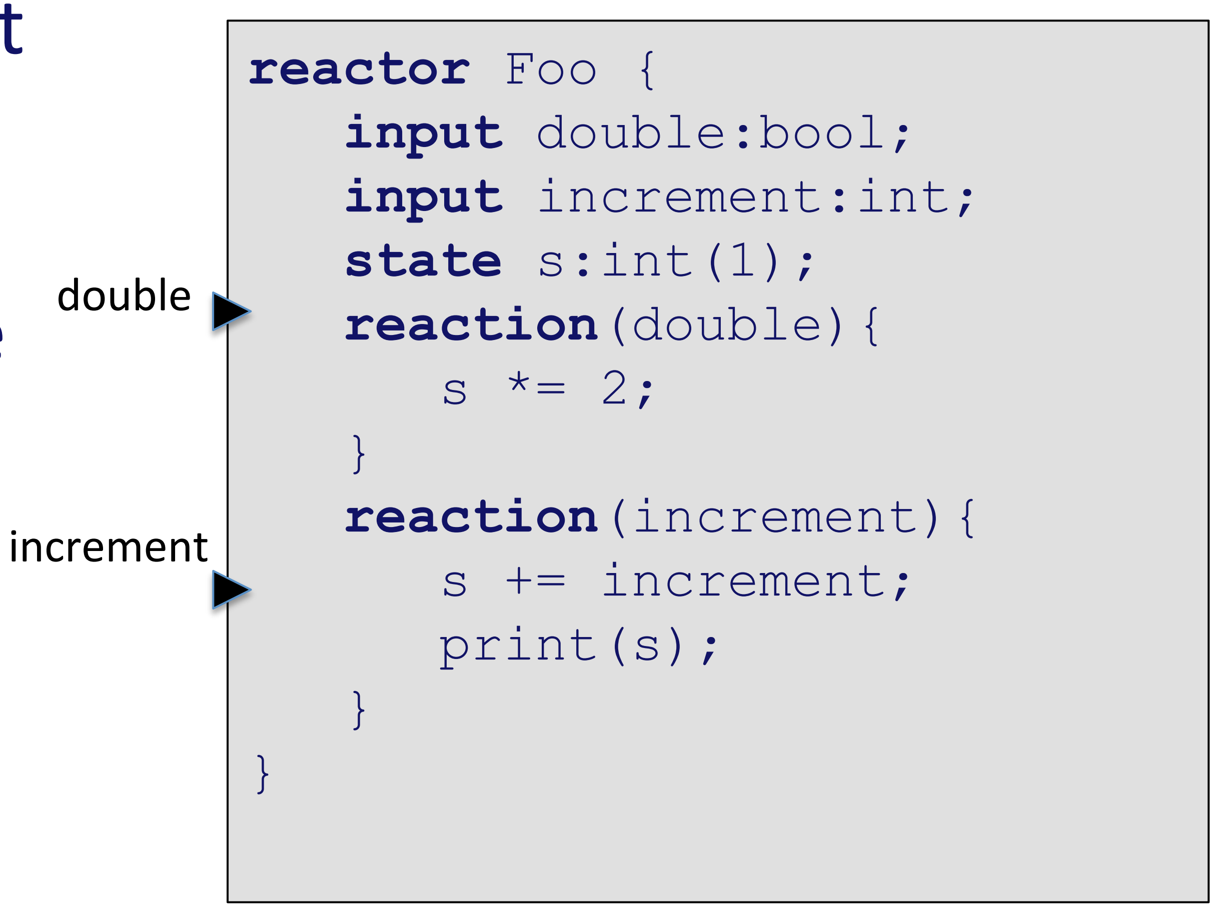


```
reactor Baz {
    input in:bool;
    output out:bool;
    reaction(in)->out{
        set(out, in);
    }
}
```

[Ptolemeus, 2014]

Part 1 of our Solution: Ports

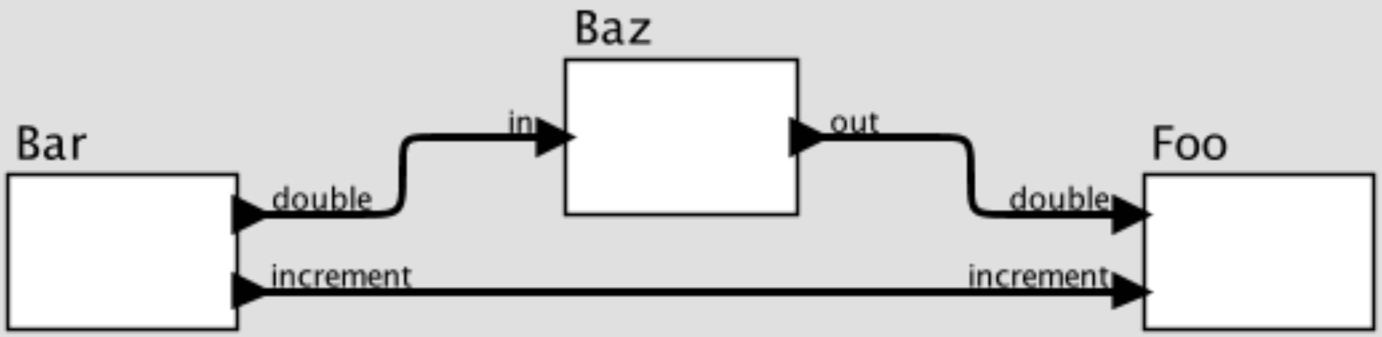
Input ports do not look much different from ordinary message handlers.



```
reactor Foo {
    input double:bool;
    input increment:int;
    state s:int(1);
    reaction(double){
        s *= 2;
    }
    reaction(increment){
        s += increment;
        print(s);
    }
}
```

Part 2 of our Solution: Hierarchy

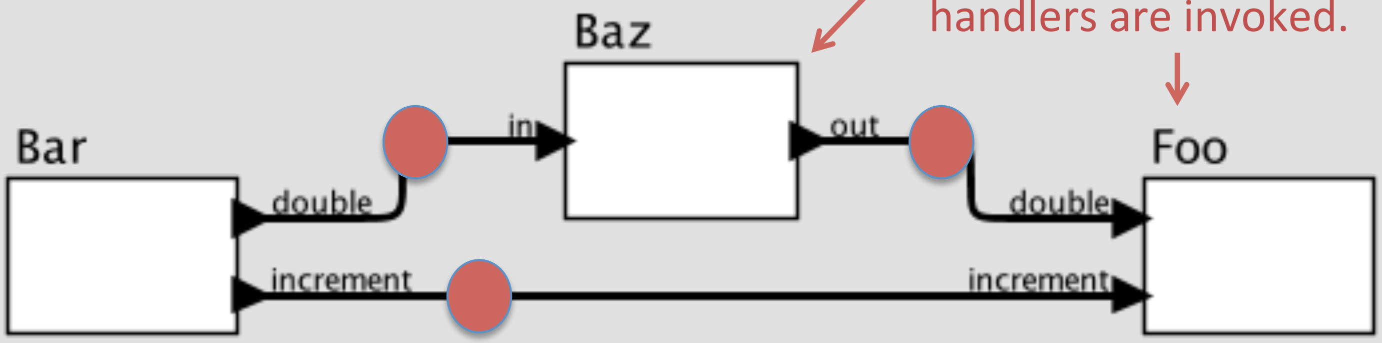
```
main reactor Top {
    x = new Foo();
    y = new Bar();
    z = new Baz();
    y.double -> z.in;
    y.increment -> x.increment;
    z.out -> x.double;
}
```



Part 3 of our Solution: Scheduling

```
main reactor Top {
    x = new Foo();
    y = new Bar();
    z = new Baz();
    y.double -> z.in;
    y.increment -> x.increment;
    z.out -> x.double;
}
```

Scheduling becomes especially interesting when production or consumption of messages is data dependent.



Ensure that Baz completes before Foo's handlers are invoked.

Deterministic Concurrent Models of Computation

- Dataflow (DF)
- Process Networks (PN)
- Synchronous/Reactive (SR)
- Discrete Events (DE)

Discrete Events (DE), Traditionally a Simulation Technology

Time-stamped events that are processed in time-stamp order.

This MoC is widely used in simulation and HDLs.

Given time-stamped inputs, it is a deterministic concurrent MoC.

A few texts that use the DE MoC

Synchronization and Linearity
An Algebra for Discrete Event Systems
Wiley
2001 Standard
Thomas & Moorby's
The Verilog Hardware Description Language
Fifth Edition
GNS3 Network Simulation Guide
"RedNectar" Chris Welsh
PACKT open source
Networking
Second Edition
Network Simulator Mapping Guide
Jeffrey S. Beasley
Pearson
Advances in CAD for VLSI, Volume 7
Hardware Description Languages
Edited by R.W. Hartenstein
North-Holland
Introduction to Discrete Event Systems
Christos G. Cassandras
Stéphane Lafortune
Kluwer Academic Publishers
Introduction to Discrete Event Systems
Second Edition
Christos G. Cassandras
Stéphane Lafortune
Discrete-Event System Simulation
Fourth Edition
Jerry Banks · John S. Carson II
Barry L. Nelson · David M. Nicol
Network Simulation Experiments
Computer Networks
a systems approach
MK
Larry L. Peterson and Bruce S. Davie

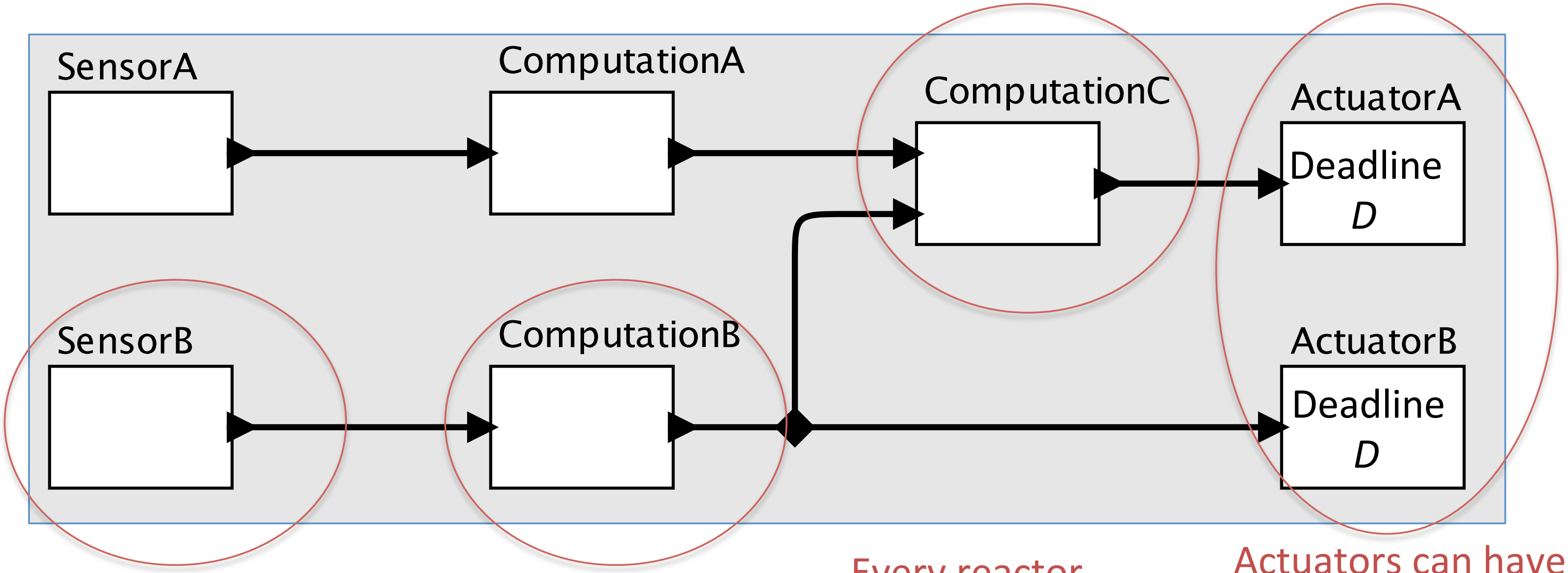
Warning

Any discussion of Discrete-Event systems involves (at least) two time lines: logical and physical.

Natural languages have no constructs for talking about two or more time lines at the same time.

[Lee & Zheng, 2007]

Motivating Example



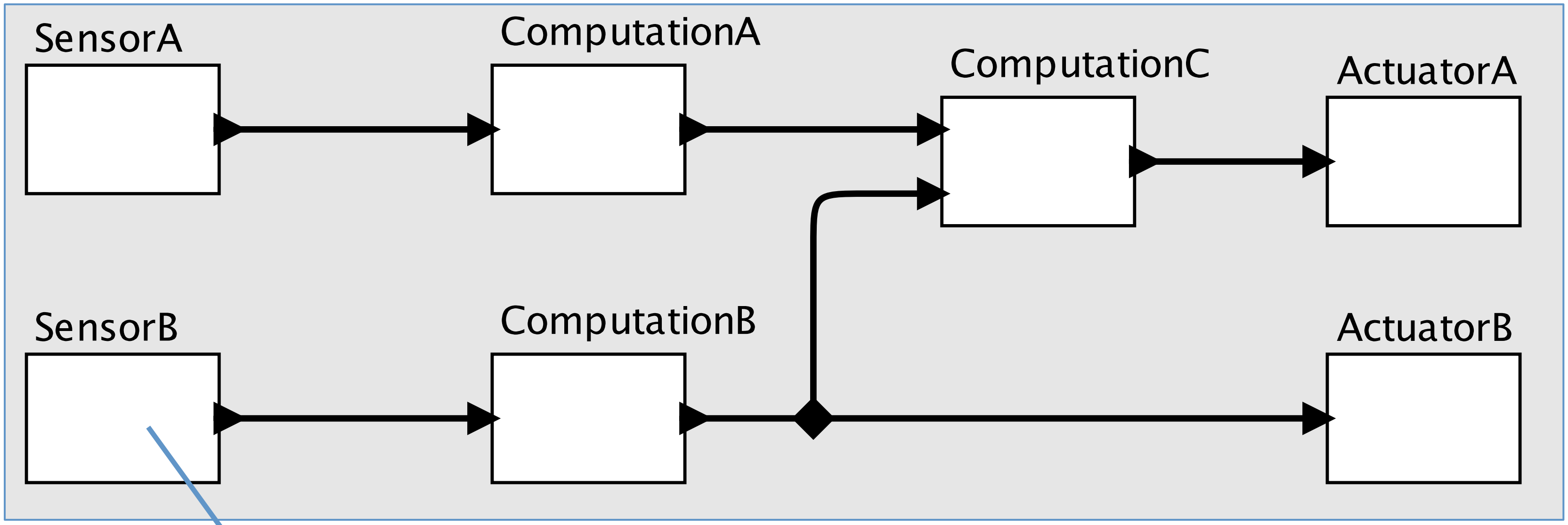
Sporadic events are assigned a time stamp based on the local physical-time clock

Computations have logically zero delay.

Every reactor handles events in time-stamp order. If time-stamps are equal, events are "simultaneous"

Actuators can have a deadline *D*. An input with time stamp *t* is required to be delivered to the actuator before the local clock hits *t* + *D*.

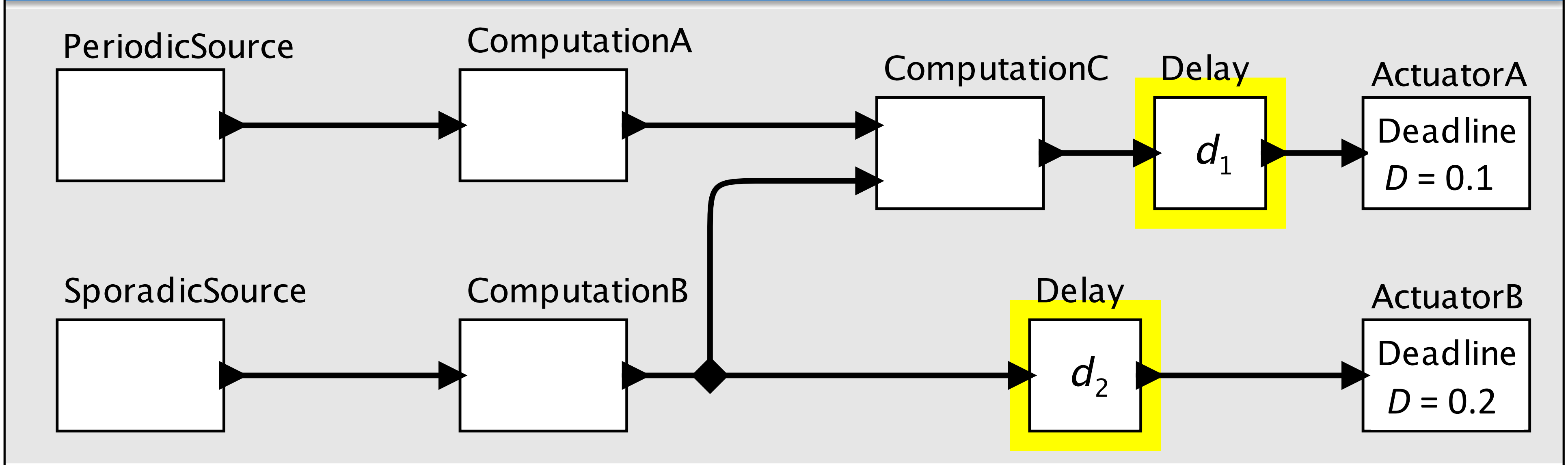
Simple, Single-Machine Realization



When a sporadic sensor triggers (or an asynchronous event like a network message arrives), assign a time stamp based on the local physical-time clock.

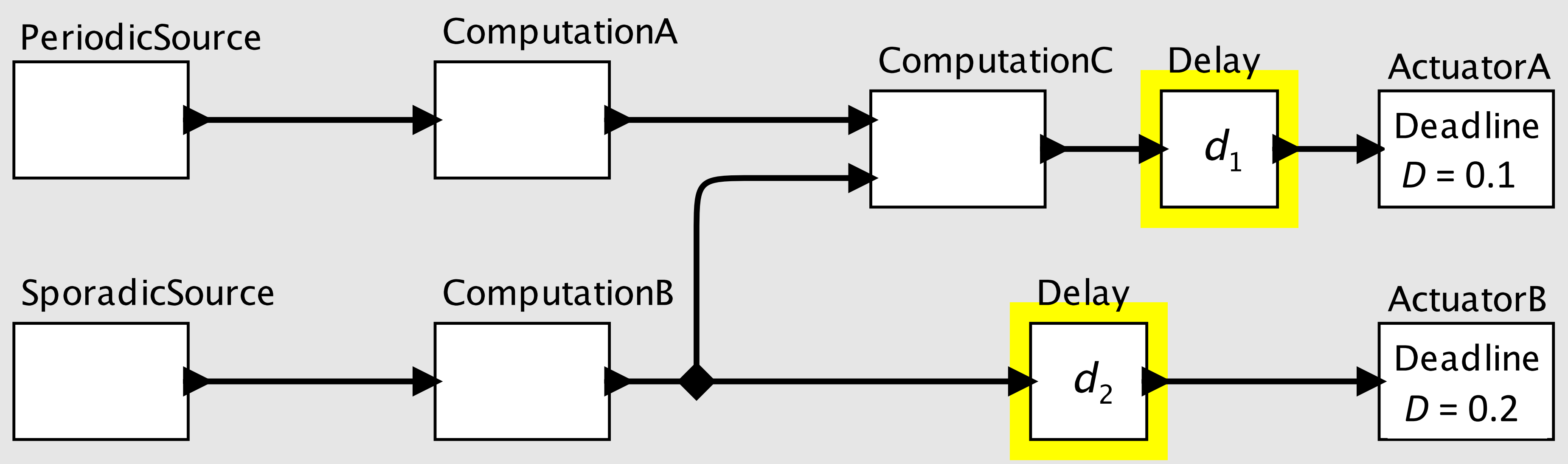
- Sort reactions topologically based on precedences.
- Global notion of "current time" *t.*
- Event queue containing future events.
- Choose earliest time stamp *t'* on the queue.
- Wait for the real-time clock to match *t'.*
- Execute reactions in topological sort order.

Temporal Operators (Logical Time)



This example has a pre-defined latency from physical sensing to physical actuation, thereby delivering a closed-loop deterministic cyber-physical model.

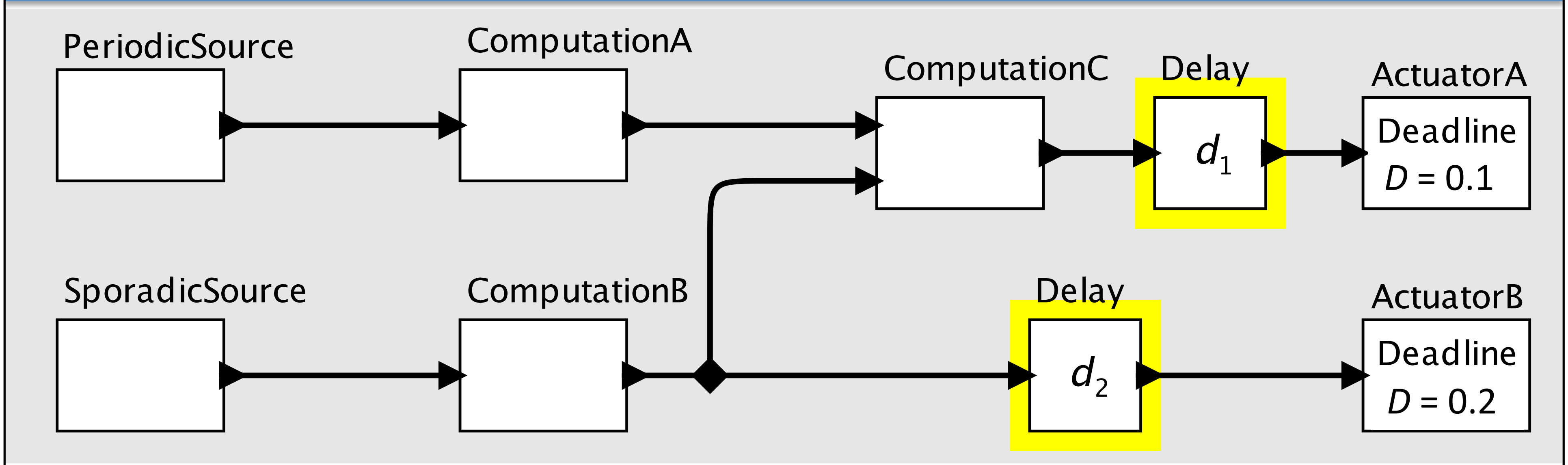
Real-Time Systems



Classical real-time systems scheduling and execution-time analysis determines whether the specification can be met.

[Buttazzo, 2005] [Wilhelm et al., 2008]

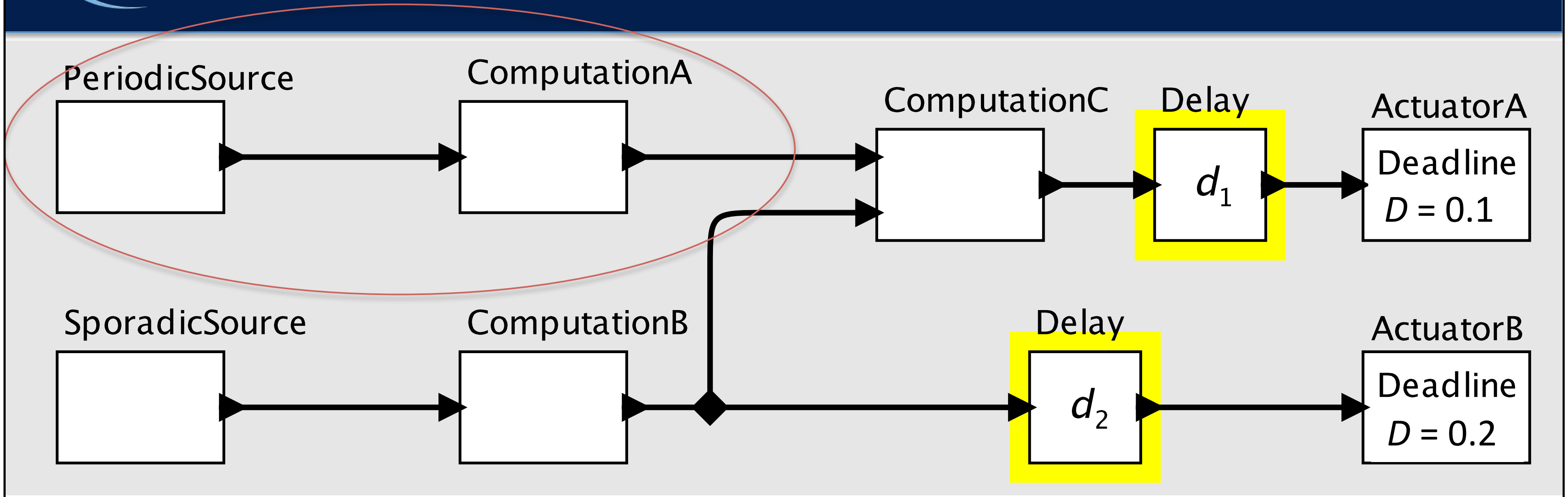
Iron-Clad Guarantees with PRET Machines



Precision-timed (PRET) machines deliver deterministic clock-cycle-level repeatable timing with no loss of performance on sporadic workloads.

[Edwards & Lee, 2007] [Lee et al., 2017]

Opportunity for Optimization



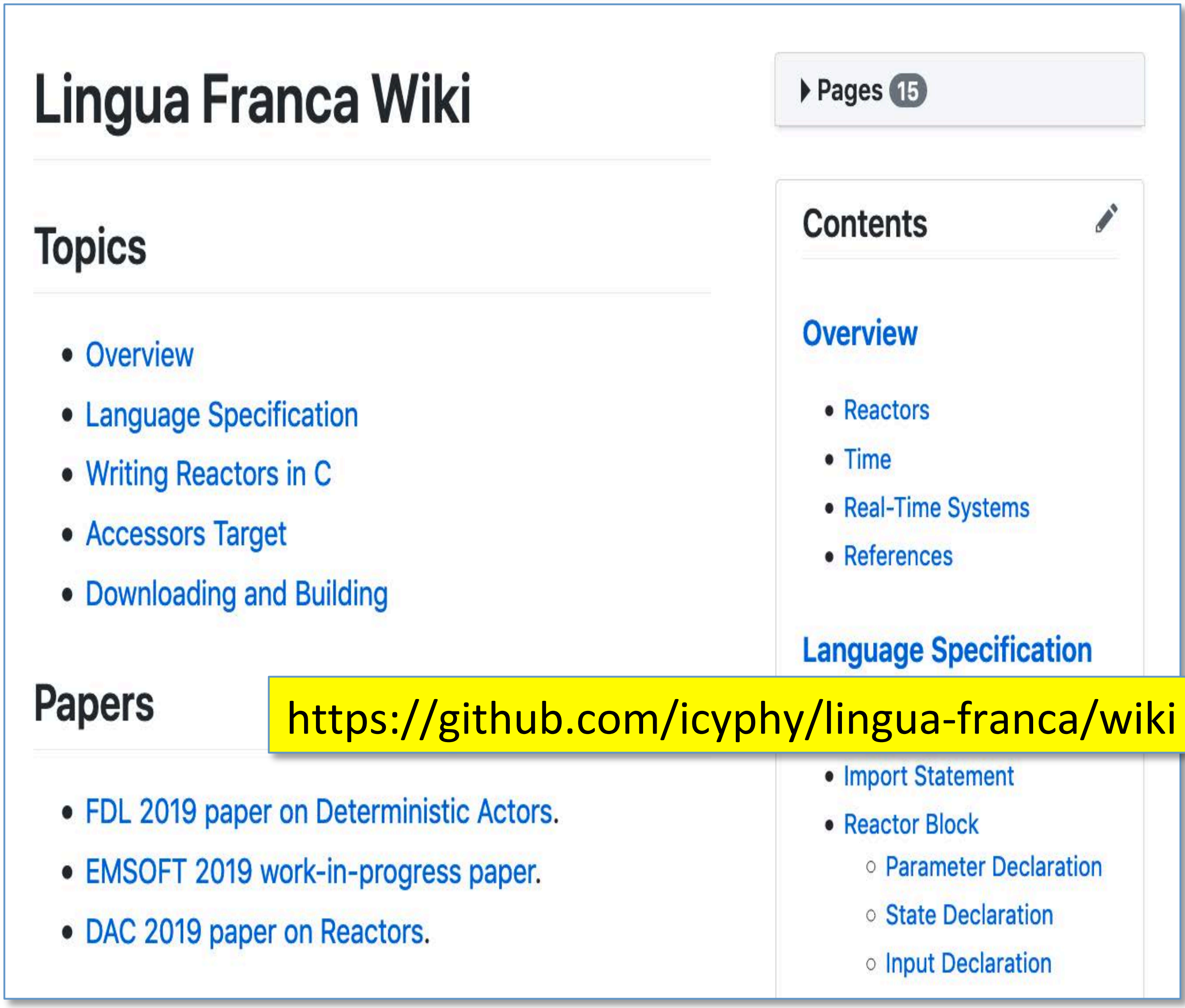
If the PeriodicSource does not depend on physical inputs, then pre-computing (logical time ahead of physical time) becomes possible, based on dependence analysis.

Outline

- Motivating Problems
- Why Existing Methods Fall Short
- Ports, Hierarchy, Models of Computation
- The Lingua Franca Language
- Distributed Execution
- Clock Synchronization
- Conclusion

A Solution: Lingua Franca

A polyglot meta-language with DE semantics for implementation (not simulation) of deterministic, concurrent, time-sensitive systems.



Lingua Franca Wiki

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

Hello World in Lingua Franca

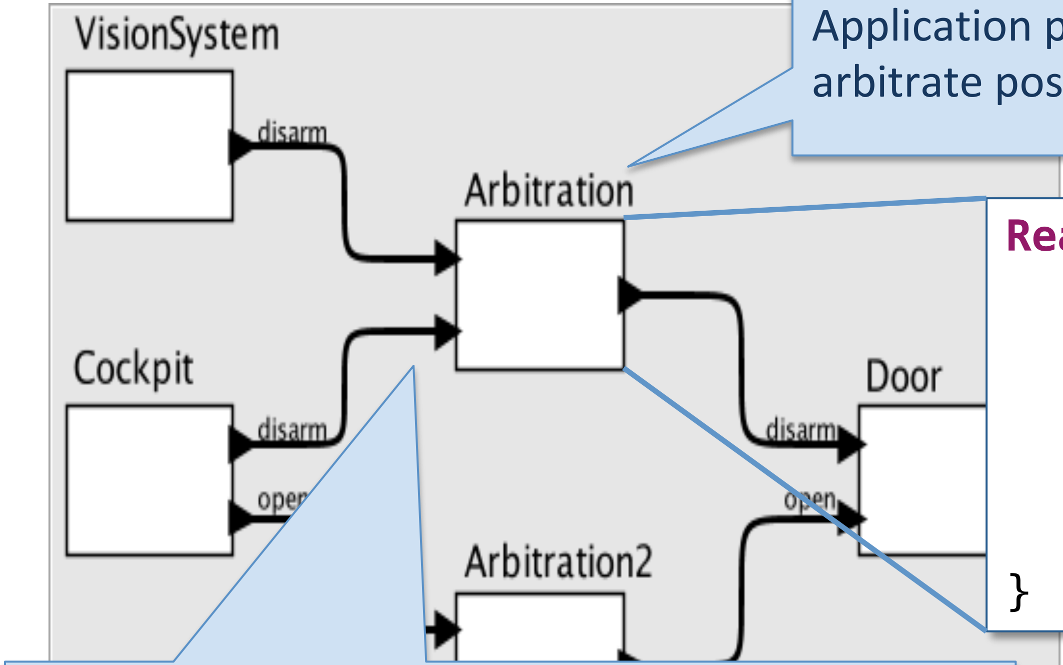
Target language (currently C, C++, and JavaScript. Plans for Python, Rust, Java)

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

Arbitrary code in the target language.

Events of various kinds trigger reactions

Determinism

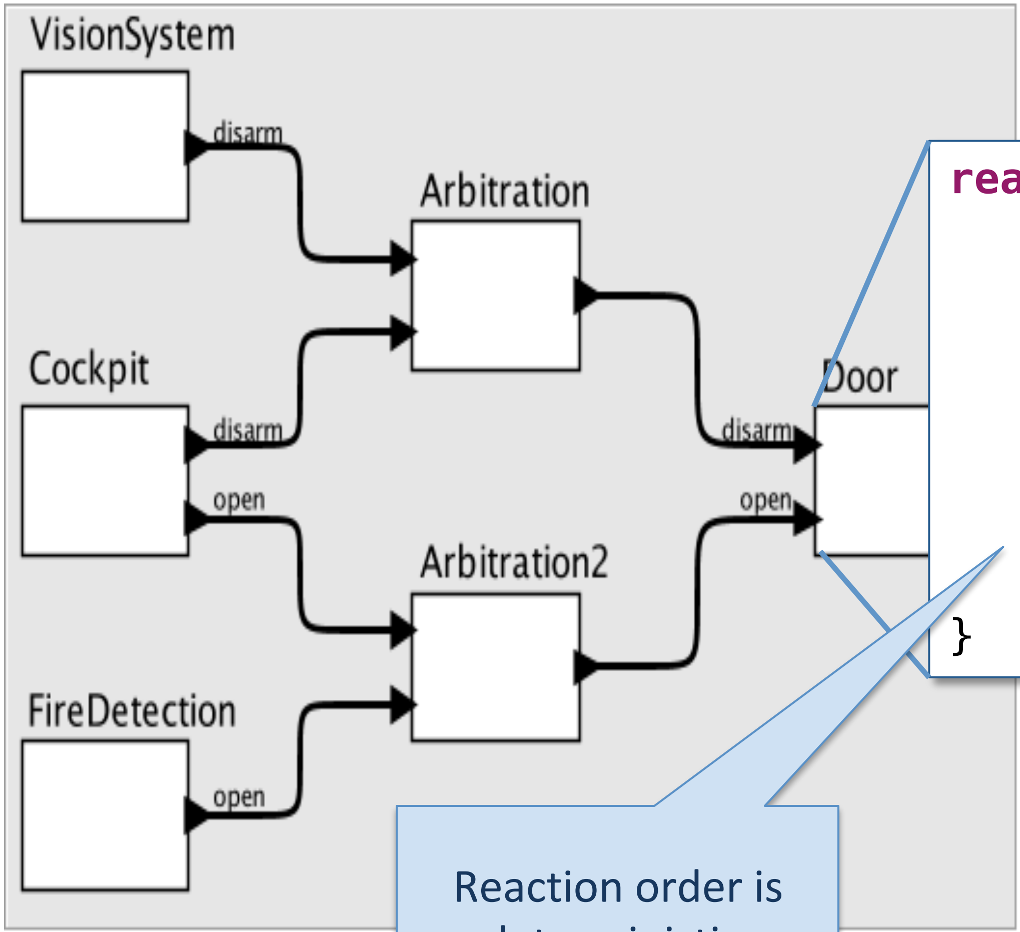


Application programmer is forced to explicitly arbitrate possibly conflicting commands.

```
Reactor Arbitration {
    input in1;
    input in2;
    output out:bool;
    reaction(in1, in2) -> out {=
        set(out, true);
    =}
}
```

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.

Determinism

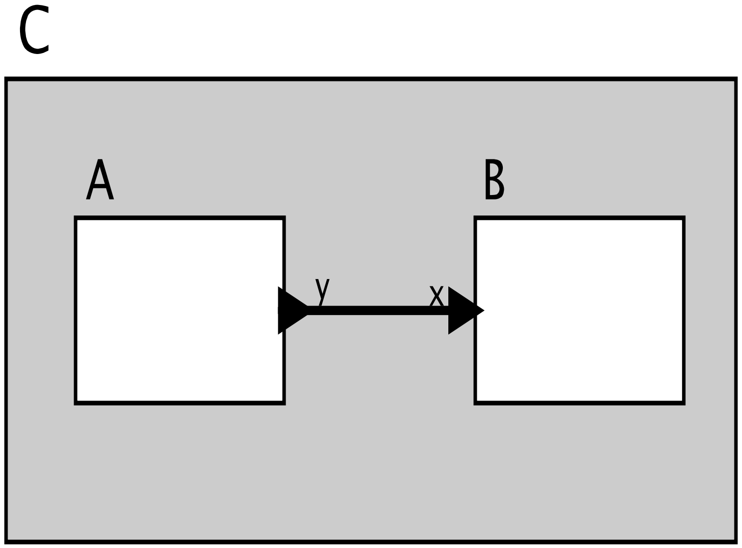


```
reactor Door {
    input disarm:bool;
    input open:bool;
    reaction(disarm) -> out {=
        … actuate disarm …
    =}
    reaction(open) -> out {=
        … actuate open…
    =}
}
```

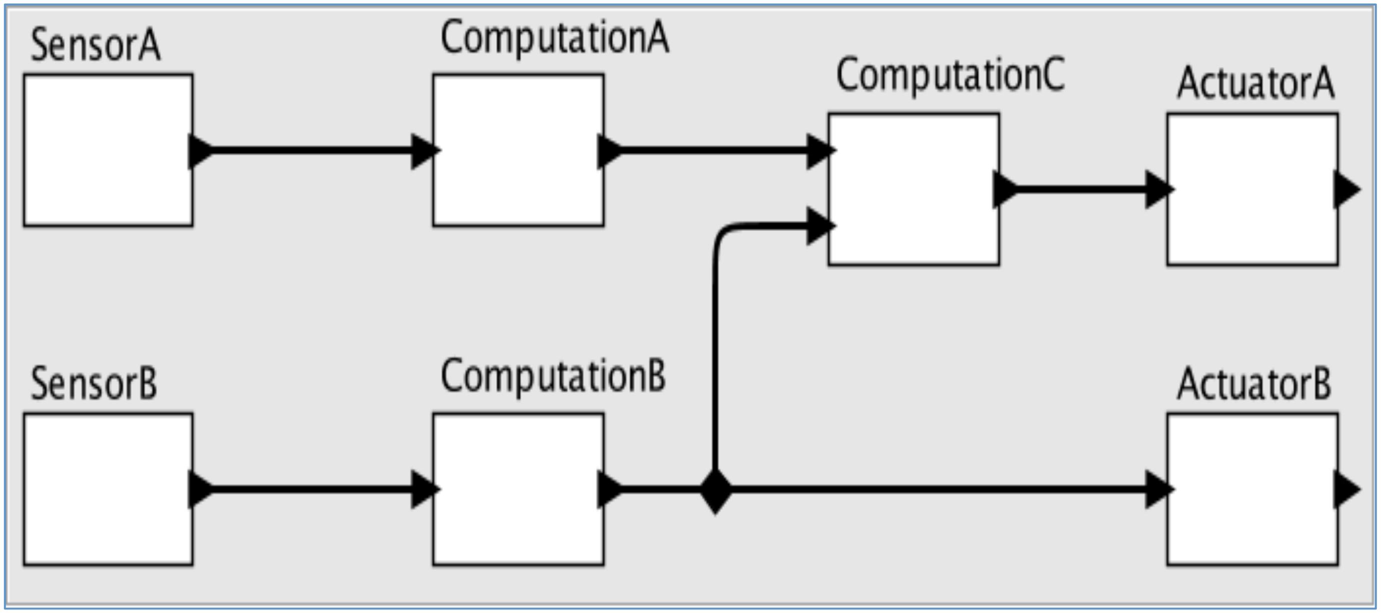
Reaction order is deterministic.

Hierarchical Composition

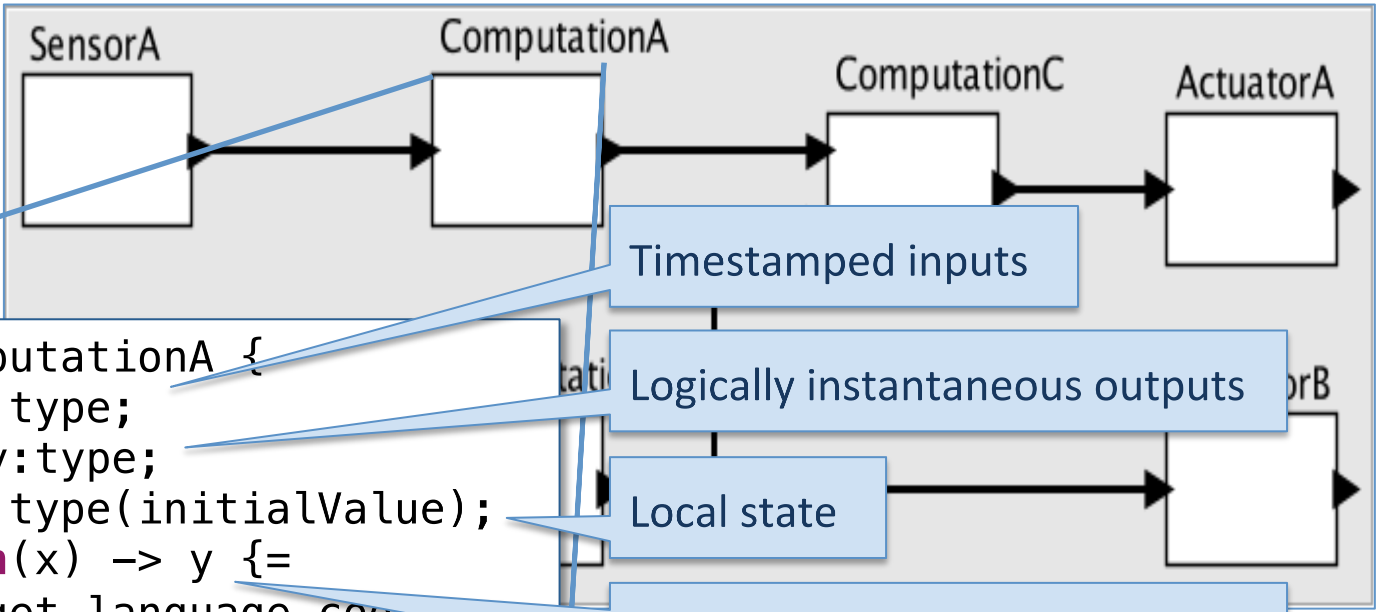
```
reactor A {
  output y;
  ...
}
reactor B {
  input x;
  ...
}
main reactor C {
  a = new A();
  b = new B();
  a.y -> b.x;
}
```



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

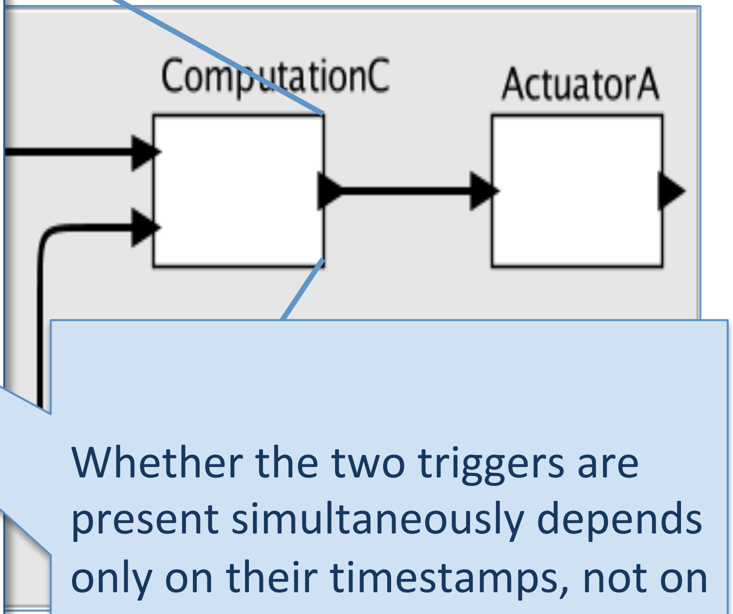
Logically instantaneous outputs

Local state

Reaction signature gives trigger(s) and production

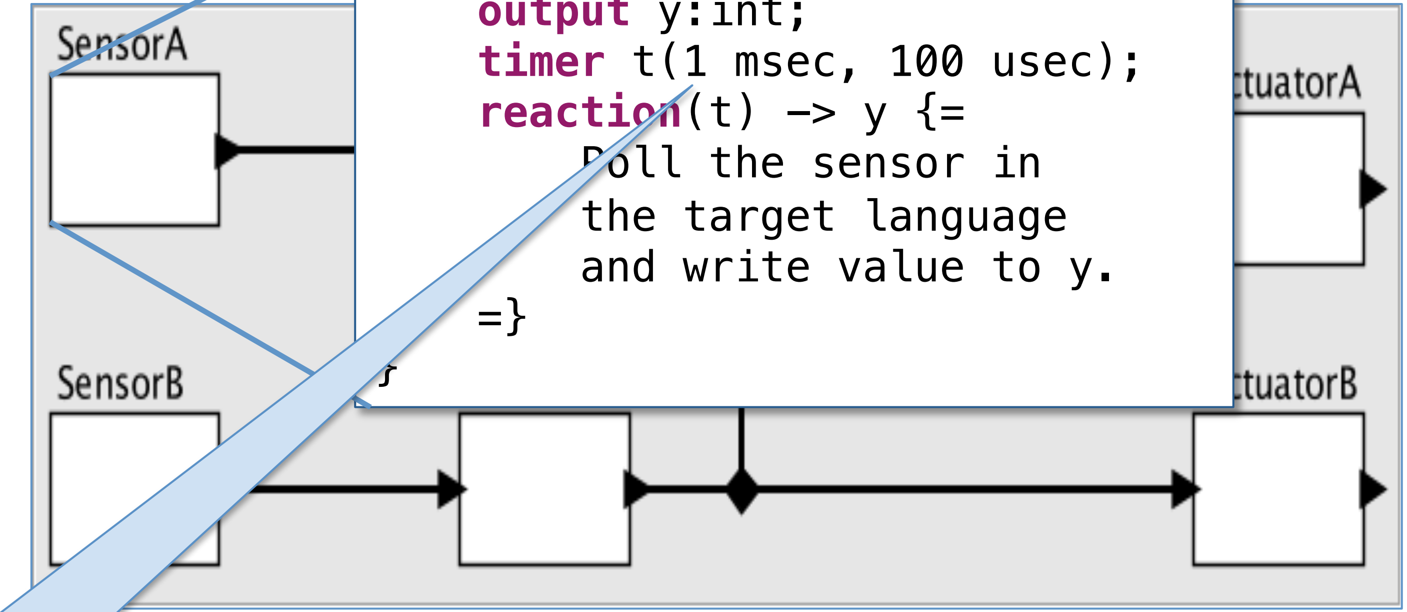
Determinism

```
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

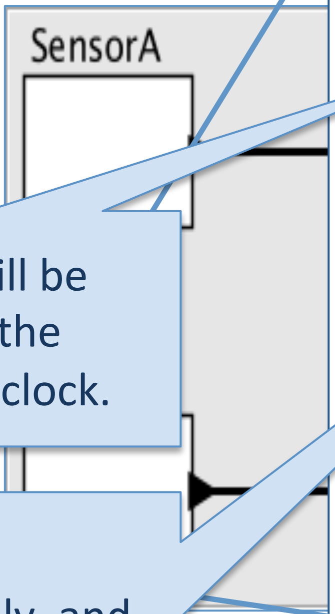


```
reactor SensorA {
    output y:int;
    timer t(1 msec, 100 usec);
    reaction(t) -> y {=
        Poll the sensor in
        the target language
        and write value to y.
    =}
}
```

Time as a first-class data type.

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



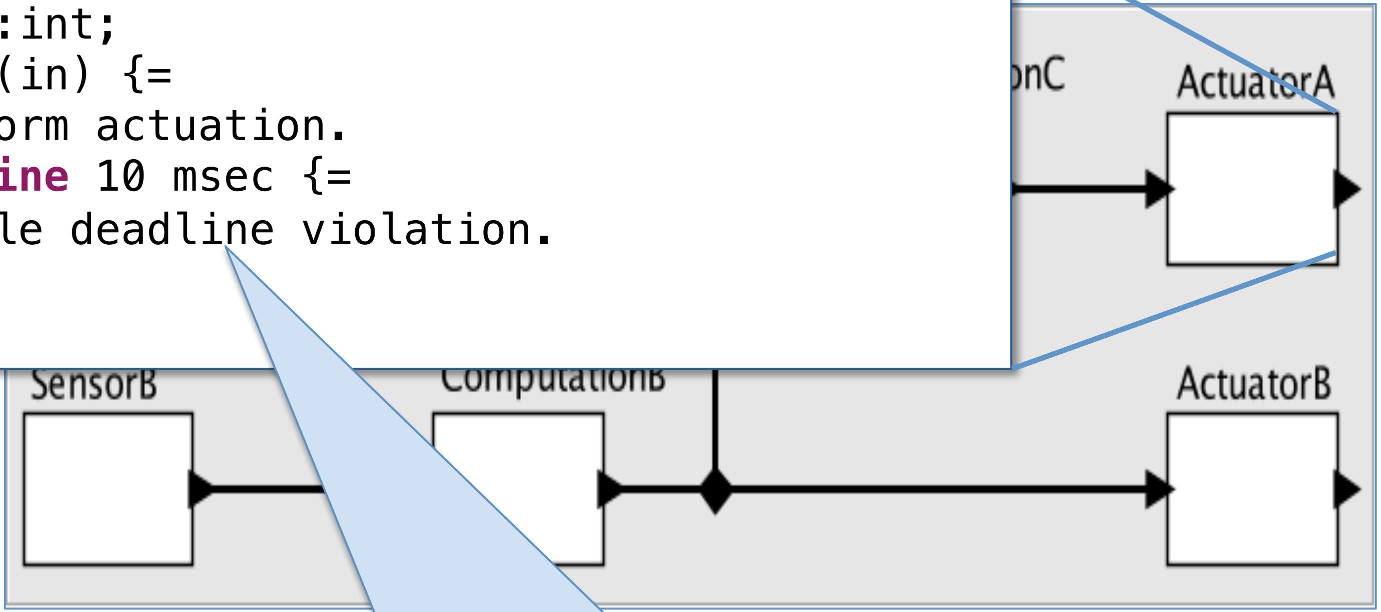
```
reactor SensorB {
    output y:int;
    physical action a:int;
    timer start;
    reaction(start) -> a {=
        Set up an interrupt service
        routine that will call:
        schedule(a, 0, value);
    =}
    reaction(a) -> y {=
        set(y, *(*int)(a->payload));
    =}
}
```

Timestamp will be derived from the local physical clock.

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

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 early.

- Eclipse/Xtext-based IDE
- C, C++, and TypeScript targets
- C code runs on Mac, Linux, Windows, and bare iron (Patmos)
- 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 multiple cores.
- Code size is tens of kilobytes.

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

Work in Progress

- EDF scheduling on multicore.
- Distributed execution based on Ptides.
- Targeting PRET machines for real time.
- Leverage Google’s Protobufs and gRPC.
  - Complex datatypes
  - Polyglot systems

Outline

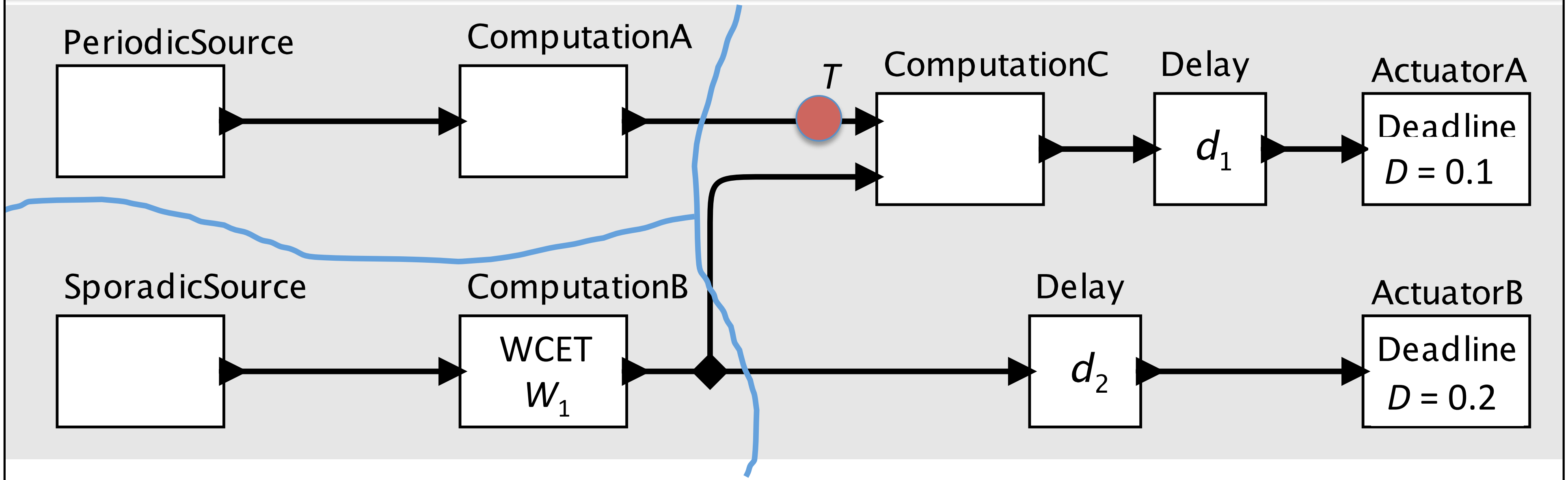
- Motivating Problems
- Why Existing Methods Fall Short
- Ports, Hierarchy, Models of Computation
- The Lingua Franca Language
- Distributed Execution
- Clock Synchronization
- Conclusion

Simple, but Nondeterministic Strategy

Lingua Franca can assign a timestamp to every incoming message using the local clock, but the overall resulting behavior will be nondeterministic.

This is OK for some applications, but not all.

Networked Scheduling: PTides



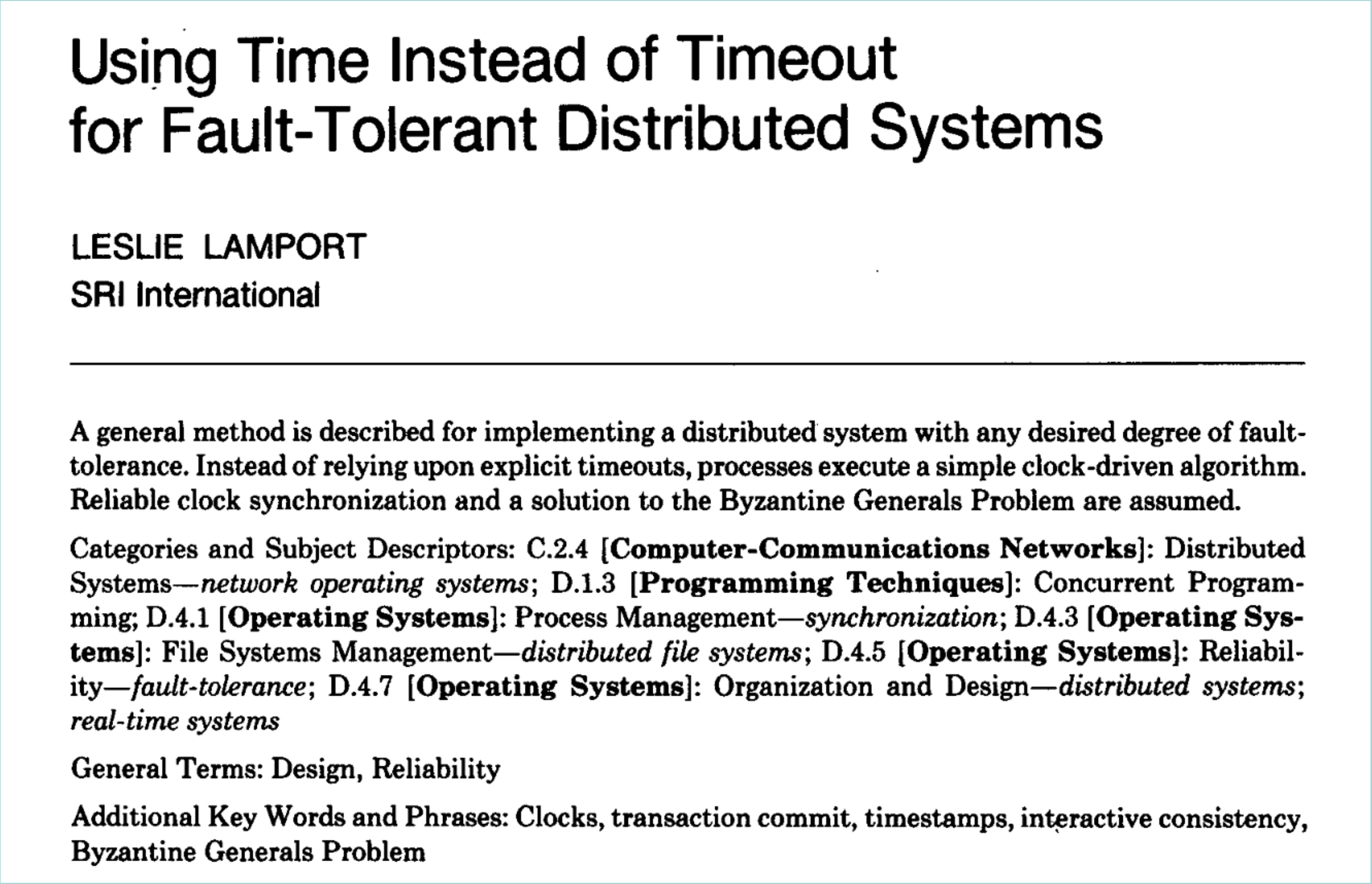
When is this "safe to process"?

When $\tau \geq T + W_1 + E + N$, where

- τ is the local physical clock time
- W_1 is worst-case execution time
- E is the bound on the clock synchronization error
- N the bound on the network delay

[Zhao et al., 2007]
[Edison et al., 2012]
[Corbett et al., 2012]

Roots of the Idea



Using Time Instead of Timeout for Fault-Tolerant Distributed Systems

LESLIE LAMPORT
SRI International

A general method is described for implementing a distributed system with any desired degree of fault-tolerance. Instead of relying upon explicit timeouts, processes execute a simple clock-driven algorithm. Reliable clock synchronization and a solution to the Byzantine Generals Problem are assumed.

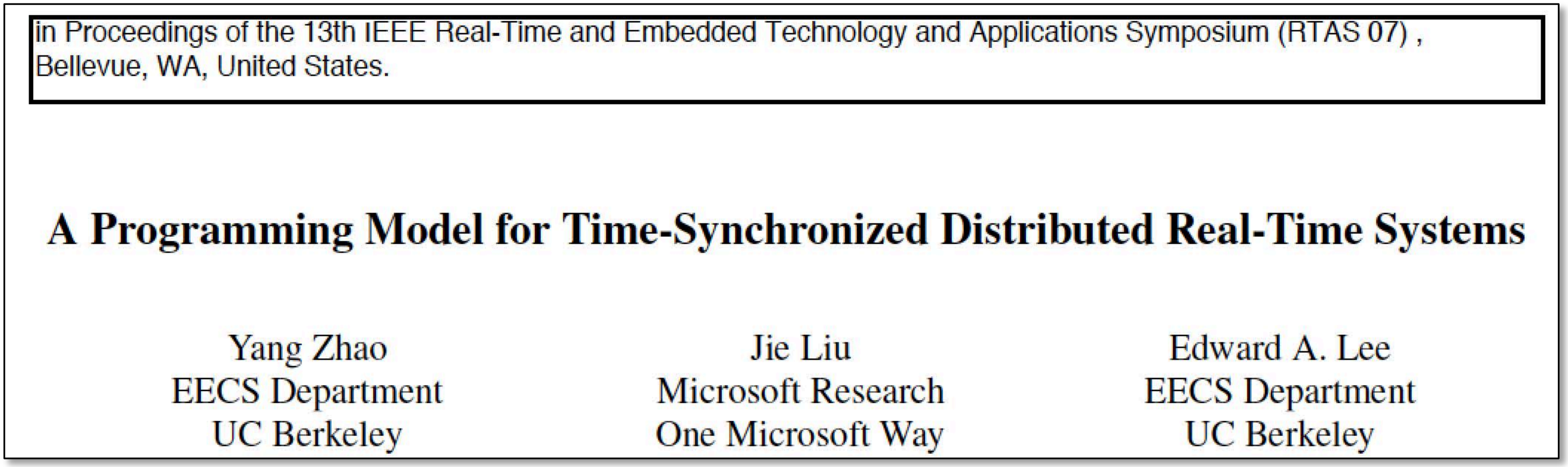
Categories and Subject Descriptors: C.2.4 [**Computer-Communications Networks**]: Distributed Systems—*network operating systems*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.4.1 [**Operating Systems**]: Process Management—*synchronization*; D.4.3 [**Operating Systems**]: File Systems Management—*distributed file systems*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*; *real-time systems*

General Terms: Design, Reliability

Additional Key Words and Phrases: Clocks, transaction commit, timestamps, interactive consistency, Byzantine Generals Problem

ACM Transactions on Programming Languages and Systems, 1984.

Ptides – A Robust Distributed DE MoC for IoIT Applications



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

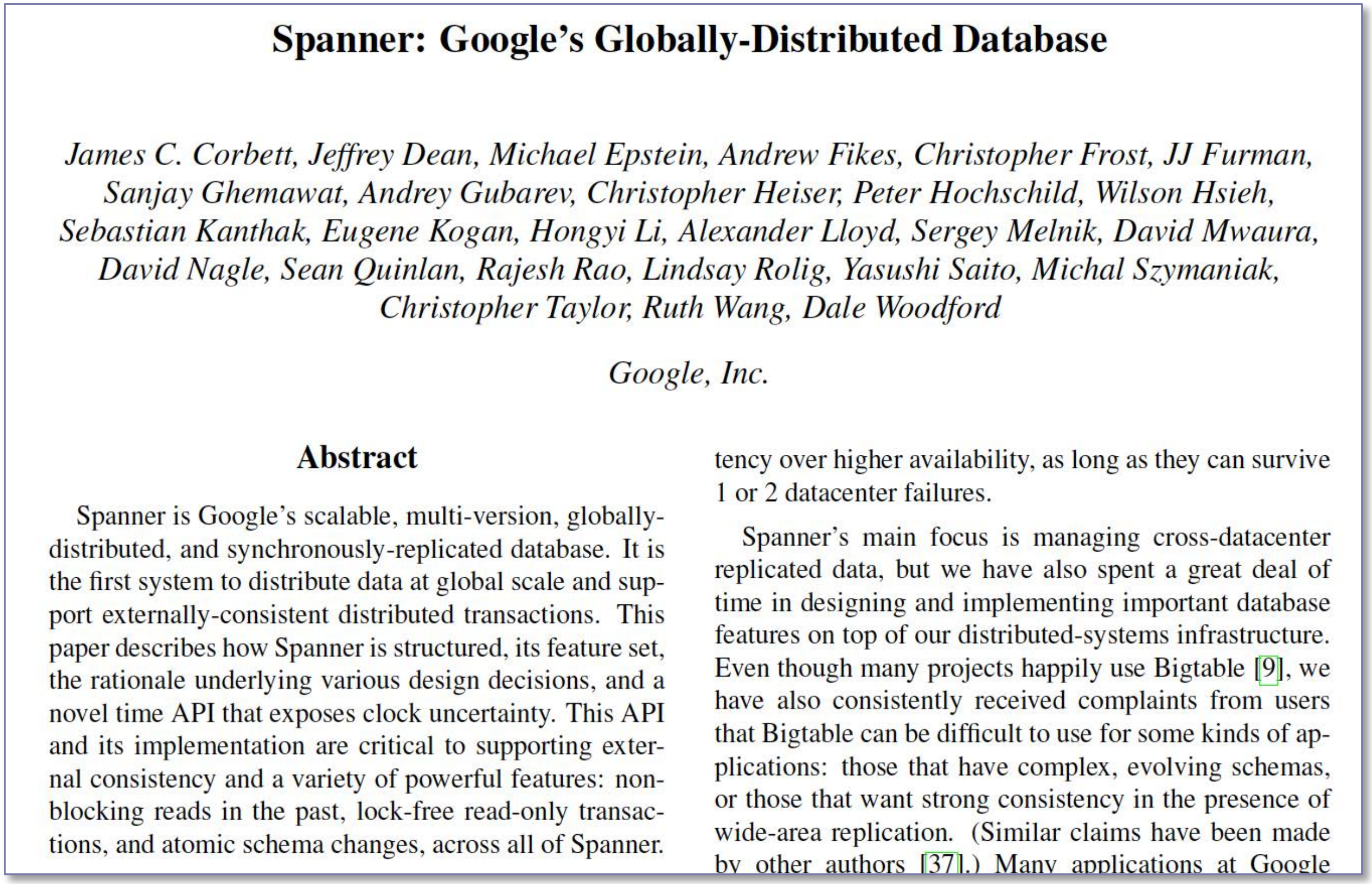
A Programming Model for Time-Synchronized Distributed Real-Time Systems

Yang Zhao	Jie Liu	Edward A. Lee
EECS Department	Microsoft Research	EECS Department
UC Berkeley	One Microsoft Way	UC Berkeley

Abstract: Discrete-event (DE) models are formal system specifications that have analyzable deterministic behaviors. Using a global, consistent notion of time, DE components communicate via time-stamped events. DE models have primarily been used in performance modeling and simulation, where time stamps are a modeling property bearing no relationship to real time during execution of the model. In this paper, we extend DE models with the capability of relating certain events to physical time…

Google Spanner – A Reinvention

Google independently developed a very similar technique and applied it to distributed databases.



Spanner: Google's Globally-Distributed Database

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford

Google, Inc.

Abstract

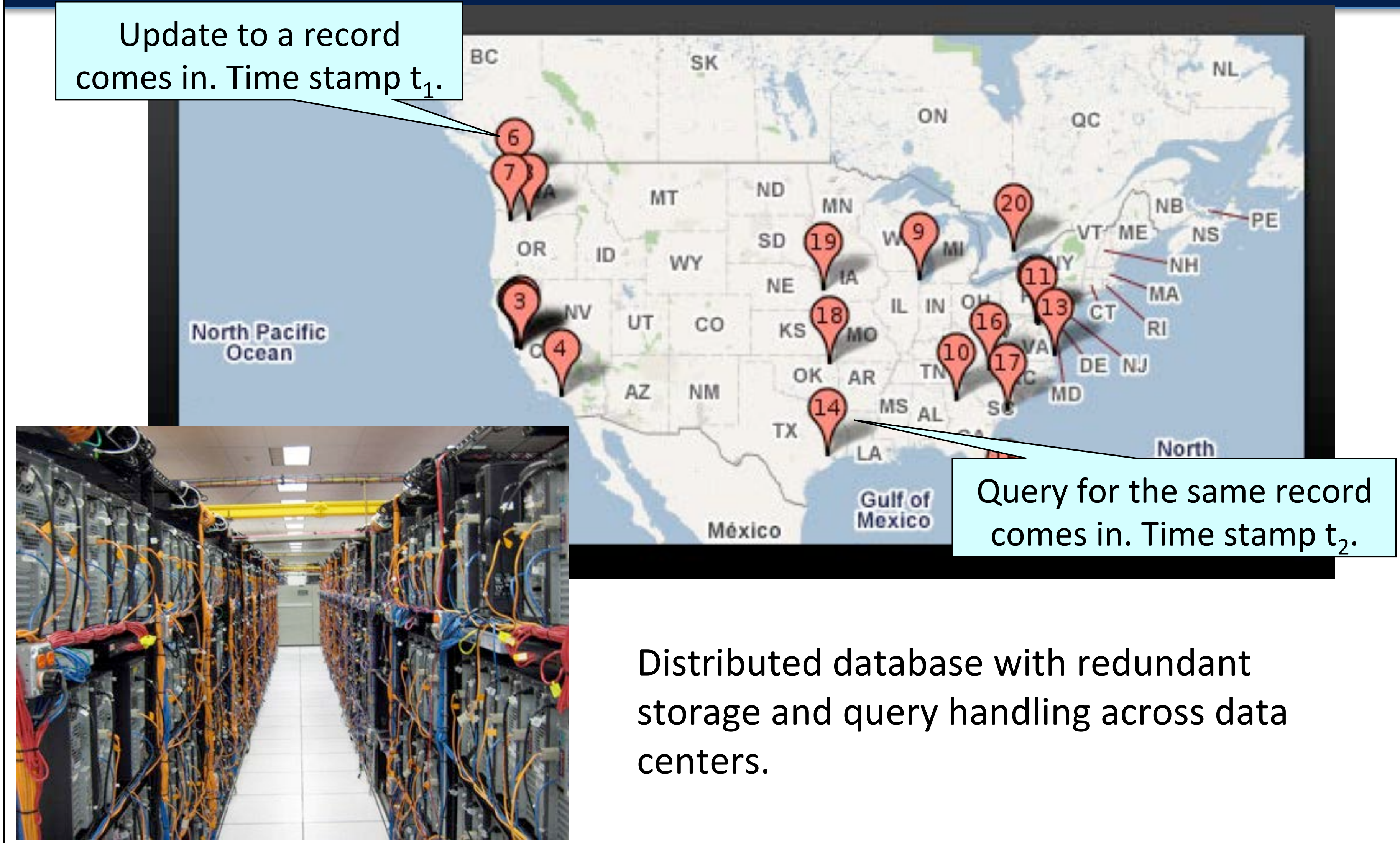
Spanner is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions. This paper describes how Spanner is structured, its feature set, the rationale underlying various design decisions, and a novel time API that exposes clock uncertainty. This API and its implementation are critical to supporting external consistency and a variety of powerful features: non-blocking reads in the past, lock-free read-only transactions, and atomic schema changes, across all of Spanner.

tency over higher availability, as long as they can survive 1 or 2 datacenter failures.

Spanner's main focus is managing cross-datacenter replicated data, but we have also spent a great deal of time in designing and implementing important database features on top of our distributed-systems infrastructure. Even though many projects happily use Bigtable [9], we have also consistently received complaints from users that Bigtable can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication. (Similar claims have been made by other authors [37].) Many applications at Google

Proceedings of OSDI 2012

Google Spanner – A Reinvention of PTIDES



Update to a record comes in. Time stamp t_1.

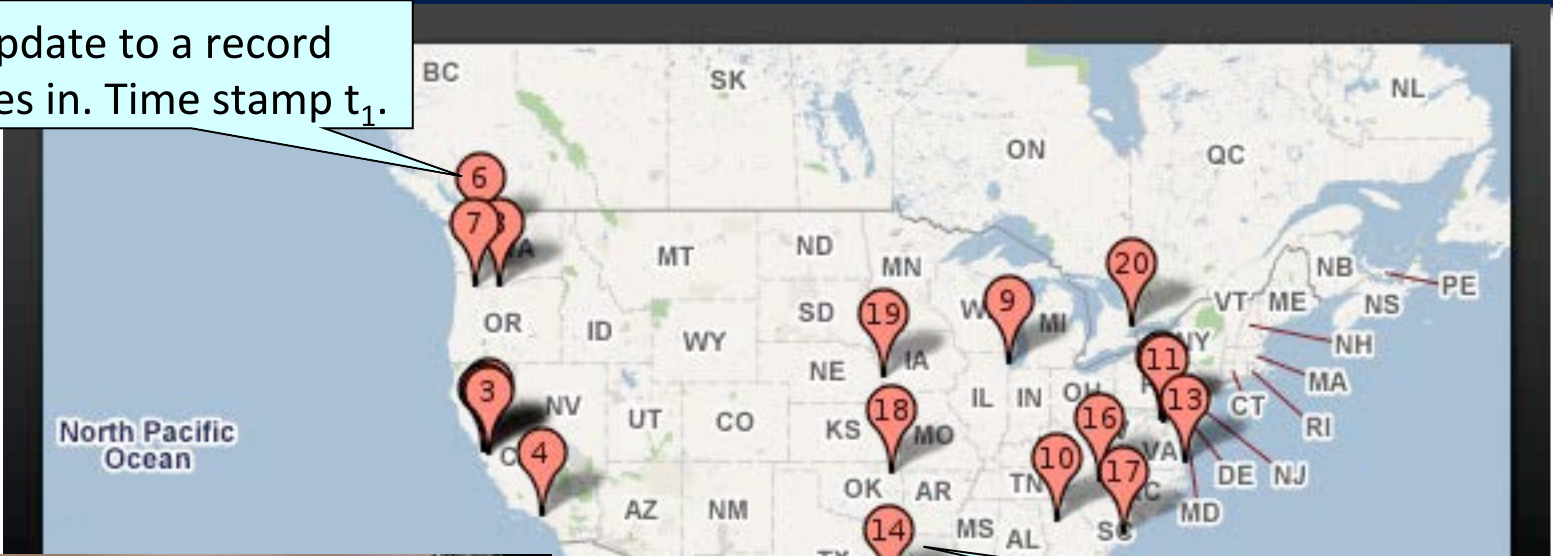
Query for the same record comes in. Time stamp t_2.

Distributed database with redundant storage and query handling across data centers.

Lee, Berkeley

Google Spanner – A Reinvention of PTIDES

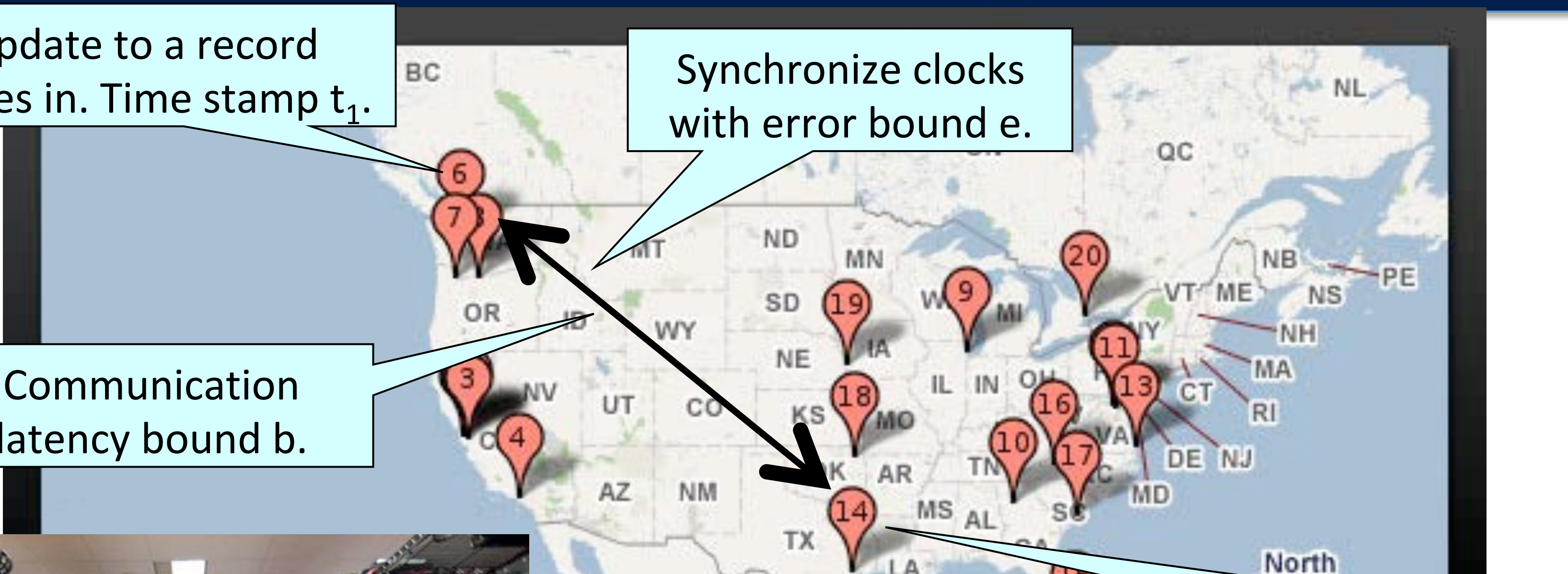
Update to a record comes in. Time stamp t_1.



Query for the same record comes in. Time stamp t_2.

If $t_2 < t_1$, the query response should be the pre-update value. Otherwise, it should be the post-update value.

Google Spanner: When to Respond?

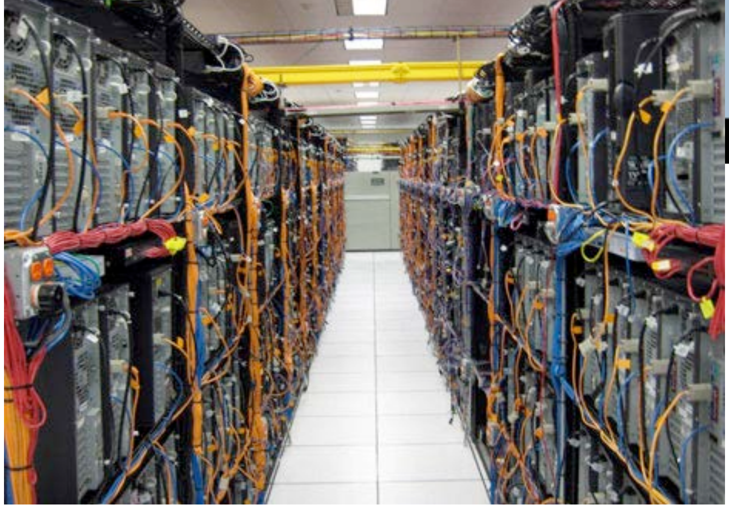


Update to a record comes in. Time stamp t_1.

Synchronize clocks with error bound e.

Communication latency bound b.

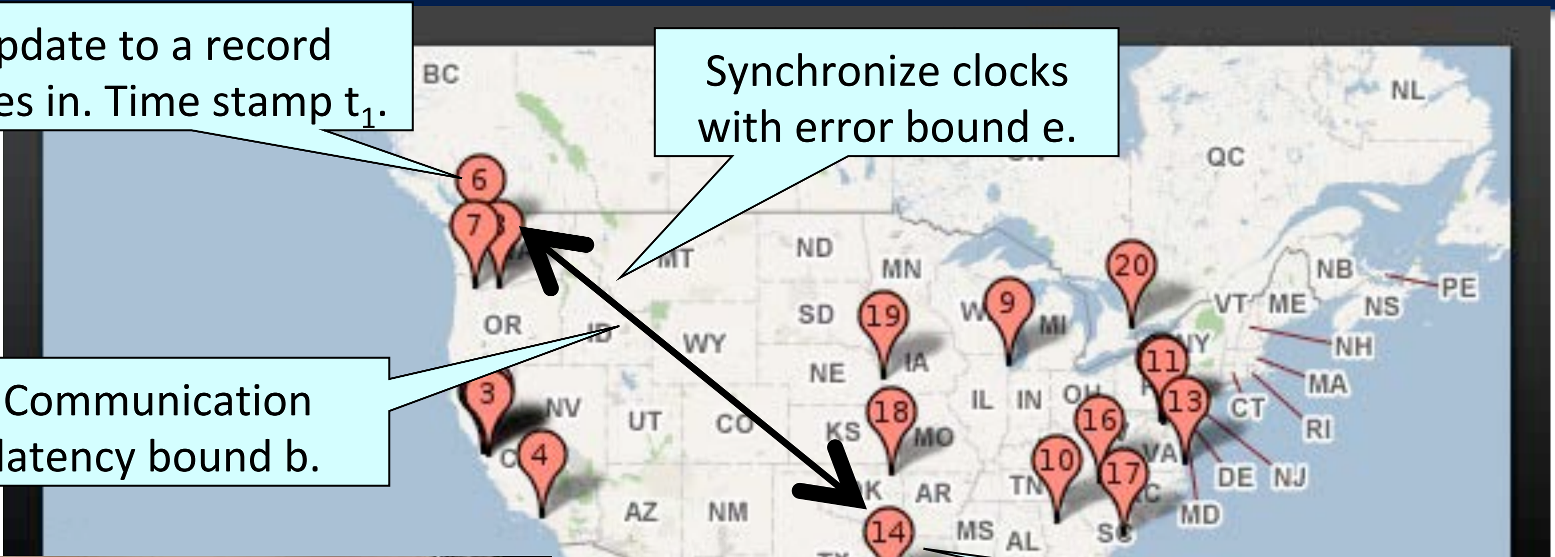
Query for the same record comes in. Time stamp t_2.



When the local clock time exceeds t_2 + e + b, issue the current record value as a response.

Lee, Berkeley

Google Spanner: Fault!



Update to a record comes in. Time stamp t_1.

Synchronize clocks with error bound e.

Communication latency bound b.

Query for the same record comes in. Time stamp t_2.

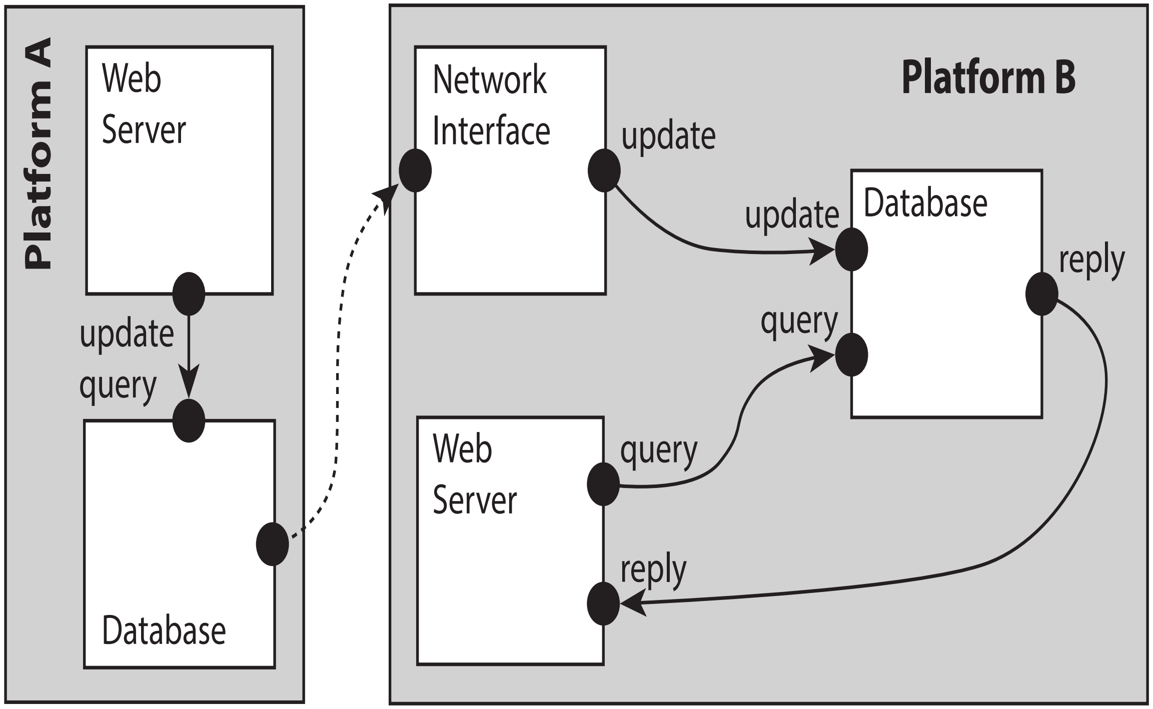
If after sending a response, we receive a record update with time stamp $t_1 < t_2$ declare a fault. Spanner handles this with a transaction schema.

Lee, Berkeley

Google Spanner as a Lingua Franca Program

Spanner's assumptions:

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



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

What can be verified with the PTIDES/Spanner approach?

1. If Door receives “open,” it will ~~eventually~~ open the door in bounded time, even if all other components fail.
2. If any component sends “disarm” ~~before~~ with a timestamp earlier than any other component’s “open” message 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]

The Essential Change in the MoC (compared to Actors, Pub/Sub, SoA, …)

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

Deterministic Distributed Real-Time

Assume bounds on:

- *clock synchronization error*
- *network latency*

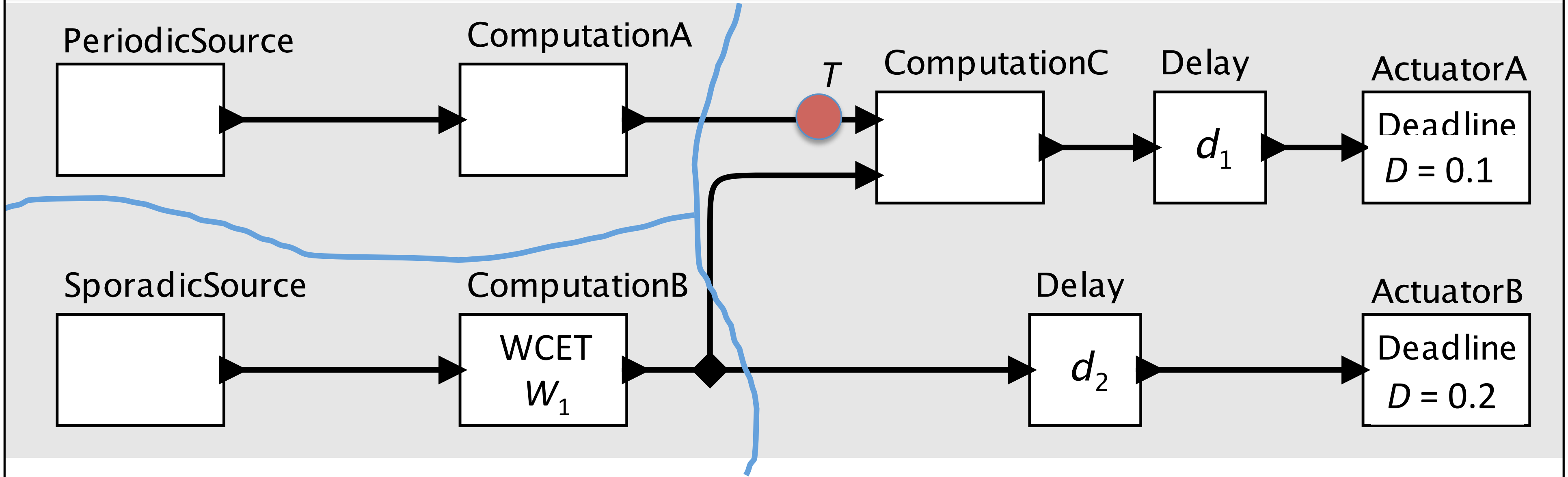
then ***events are processed in time-stamp order*** at every component. If in addition we assume

- *bounds on execution time*

then events are delivered to actuators on time.

See http://chess.eecs.berkeley.edu/ptides

Recall:
Networked Scheduling: PTides



When is this “safe to process”?

When $\tau \geq T + W_1 + E + N$, where

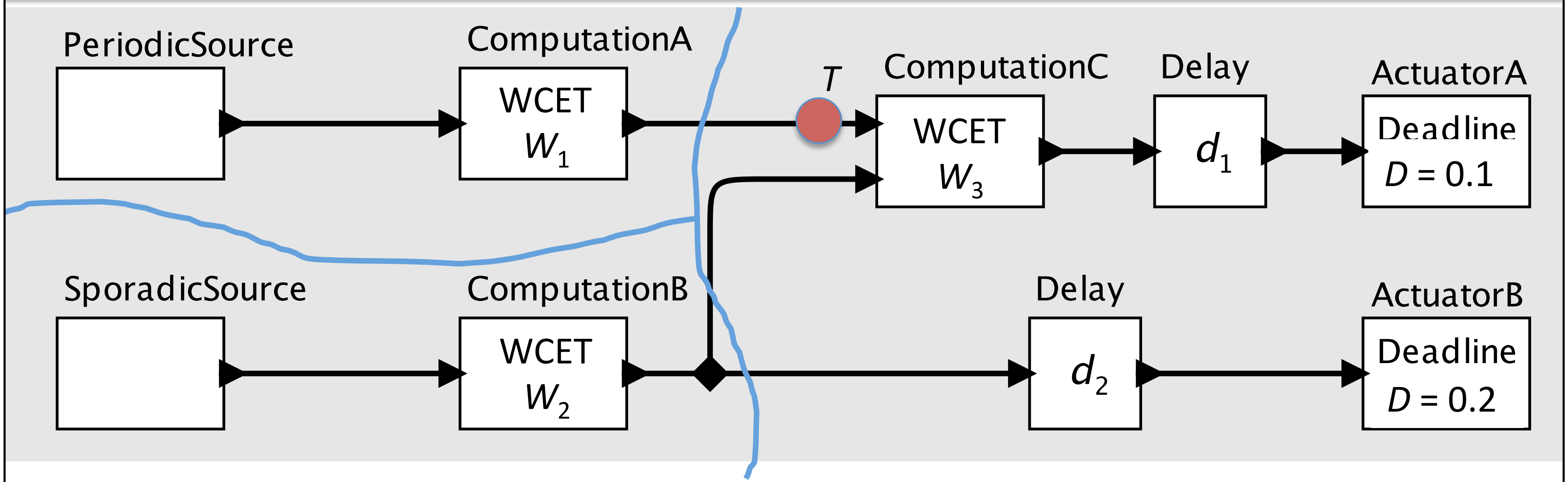
- τ is the local physical clock time
- W_1 is worst-case execution time
- E is the bound on the clock synchronization error
- N the bound on the network delay

[Zhao et al., 2007]

[Edison et al., 2012]

[Corbett et al., 2012]

Networked Scheduling: PTides

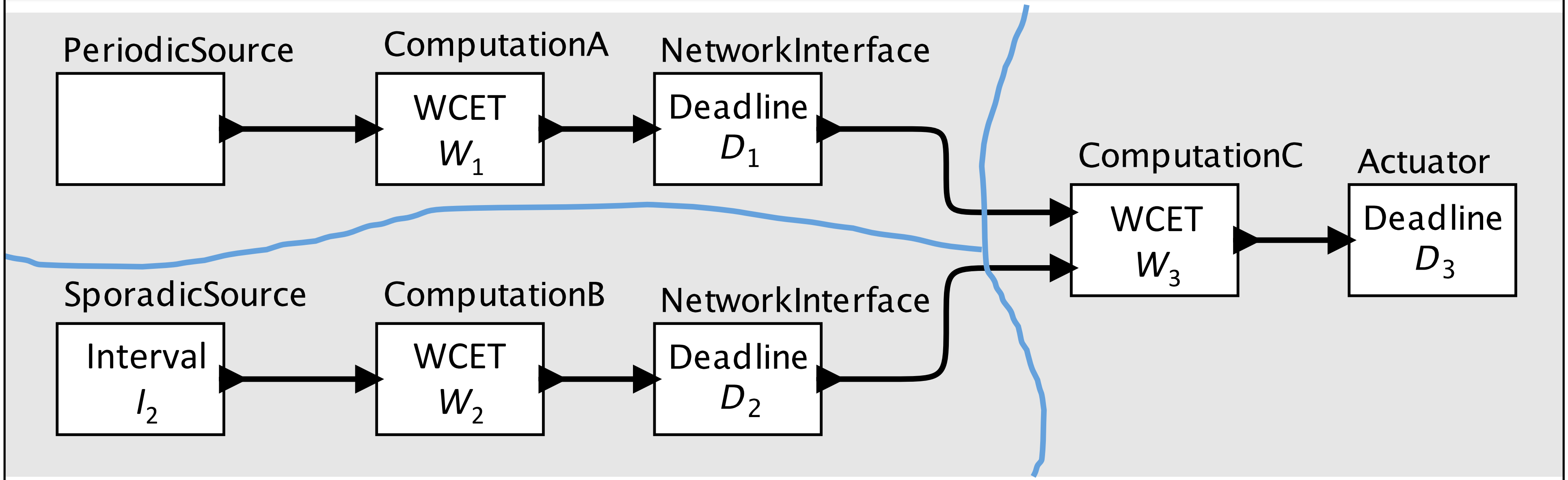


Can the deadline at ActuatorA be met?

Yes if $D + d_1 \geq \max(W_1, W_2) + E + N + W_3$

[Zhao et al., 2007]
[Edison et al., 2012]

Decoupling Real-Time Analysis with Networked Scheduling



Imposing deadlines on network interfaces decouples the real-time analysis problem. Each execution platform can be individually verified for meeting deadlines.

E.g., $I_2 \geq W_2$, $D_2 \geq W_2$, $D_3 \geq \max(D_1, D_2) + W_3 + N + E$, …

[Zhao et al., 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, clearly stated 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”

Testability

System responds in a known way to specified inputs.

Possible victim of unintended acceleration.

NASA's Toyota Study released by Dept. of Transportation in 2011 found that Toyota software was “untestable.”

At What Cost Determinism?

- Synchronized clocks
  - These are becoming ubiquitous
- Bounded network latency
  - Violations are *faults*. They are detectable.
- Bounded execution times
  - Only needed in particular places.
  - Solvable with FlexPRET HRT threads.

PTIDES Requires Synchronized Clocks with Bounded Error

Every engineered design makes assumptions about its execution platform.

Ubiquitous clock synchronization gives us a new and powerful tool.

Outline

- Motivating Problems
- Why Existing Methods Fall Short
- Ports, Hierarchy, Models of Computation
- The Lingua Franca Language
- Distributed Execution
- Clock Synchronization
- Conclusion

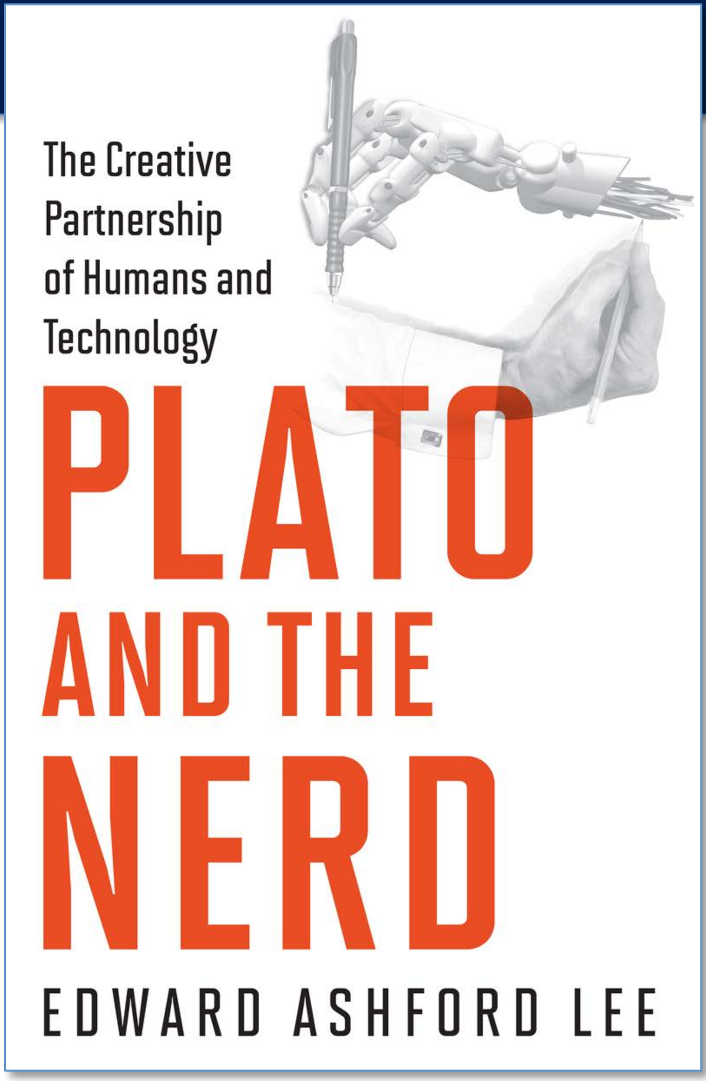
Clock Synchronization

- 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 can work without clock synchronization by reassigning timestamps to network messages.
  - Determinism is preserved within each multicore platform, but not across platforms.
- With synchronized clocks, global determinism under clearly stated assumptions becomes possible.

Recall

- 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.

Maybe we should do less science and more engineering.



The Creative Partnership of Humans and Technology

PLATO AND THE NERD

EDWARD ASHFORD LEE

http://platoandthenerd.org

Using Synchronized Clocks in Practice

Despite using TCP/IP on Ethernet, this network achieves highly reliable bounded latency.

TSN (time-sensitive networks) is starting to become pervasive…

This Bosch Rexroth printing press is a cyber-physical factory using Ethernet and TCP/IP with **high-precision clock synchronization (IEEE 1588**) on an **isolated LAN**.

Predicting the Future

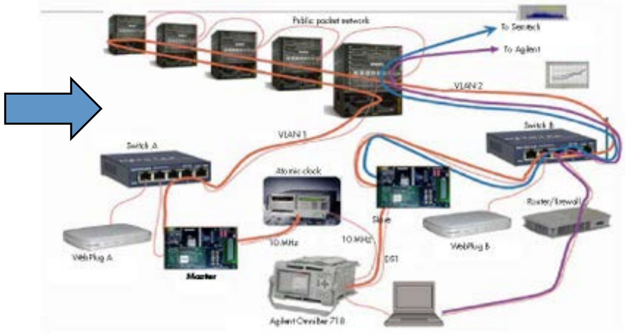
Clock synchronization is going to change the world (again)

Gregorian Calendar (BBC history)

1500s
days

Lackawanna Railroad Station, 1907, Hoboken.
Photograph by Alicia Dudek

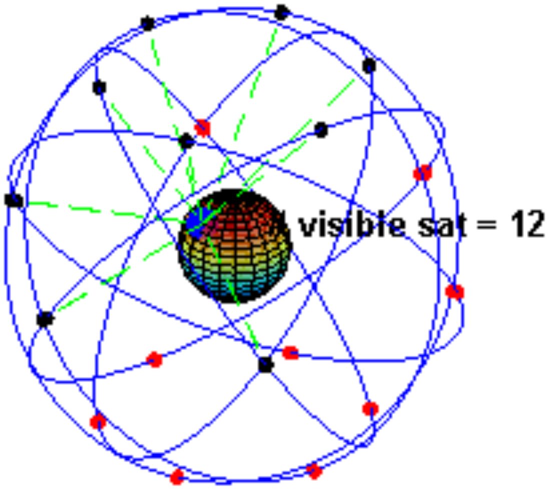
1800s
seconds



2005: first IEEE 1588 plugfest

2000s
nanoseconds

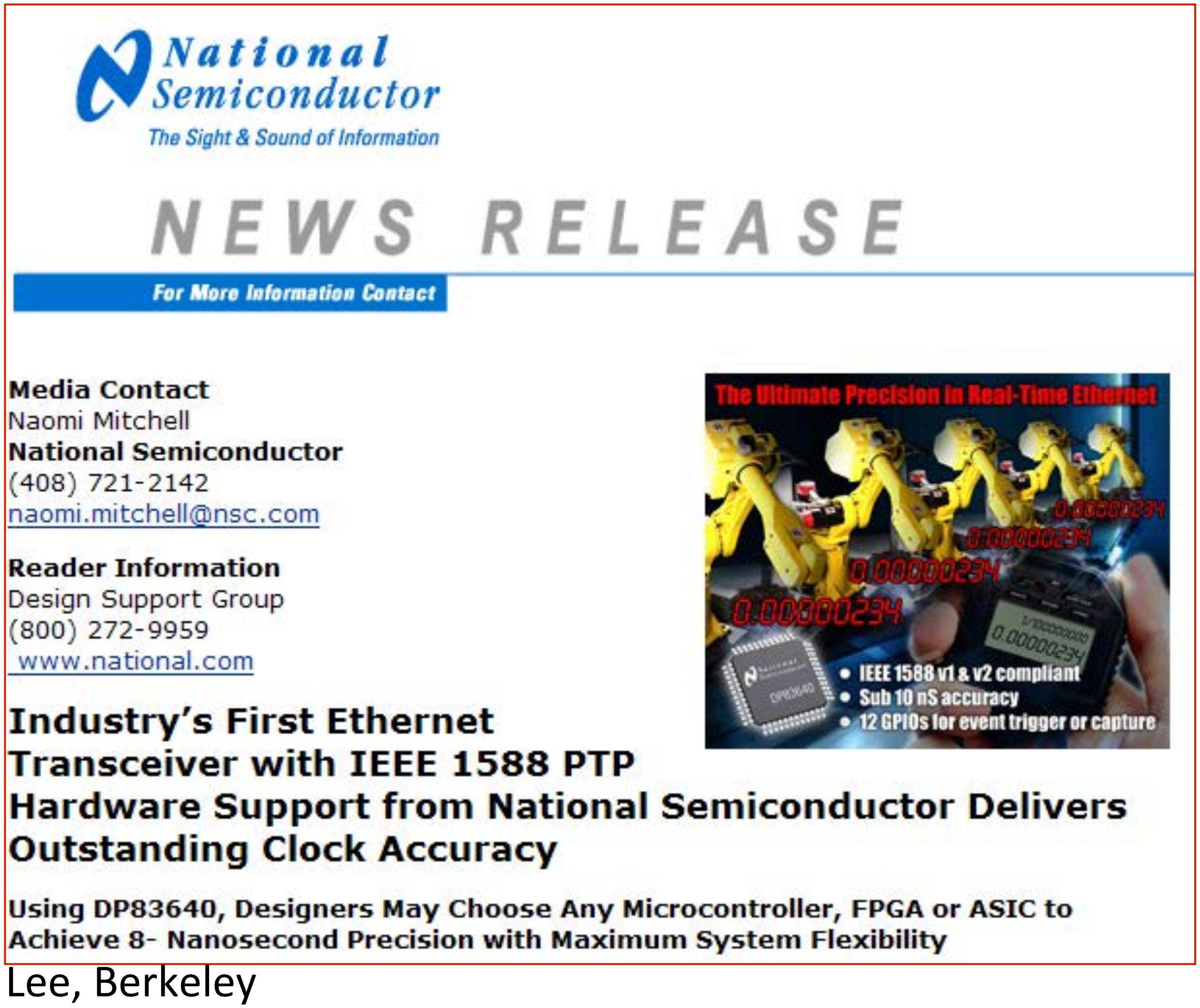
Global Positioning System



Provides ~100ns accuracy to devices with outdoor access.

Precision Time Protocols (PTP) IEEE 1588 on Ethernet

Press Release October 1, 2007



National Semiconductor
The Sight & Sound of Information

NEWS RELEASE

For More Information Contact

Media Contact
Naomi Mitchell
National Semiconductor
(408) 721-2142
naomi.mitchell@nsc.com

Reader Information
Design Support Group
(800) 272-9959
www.national.com

The Ultimate Precision in Real-Time Ethernet
0.00000234
0.00000234
0.00000234
0.00000234
1/1000000000
0.00000234
DP83640
• IEEE 1588 v1 & v2 compliant
• Sub 10 nS accuracy
• 12 GPIOs for event trigger or capture

Industry's First Ethernet Transceiver with IEEE 1588 PTP Hardware Support from National Semiconductor Delivers Outstanding Clock Accuracy

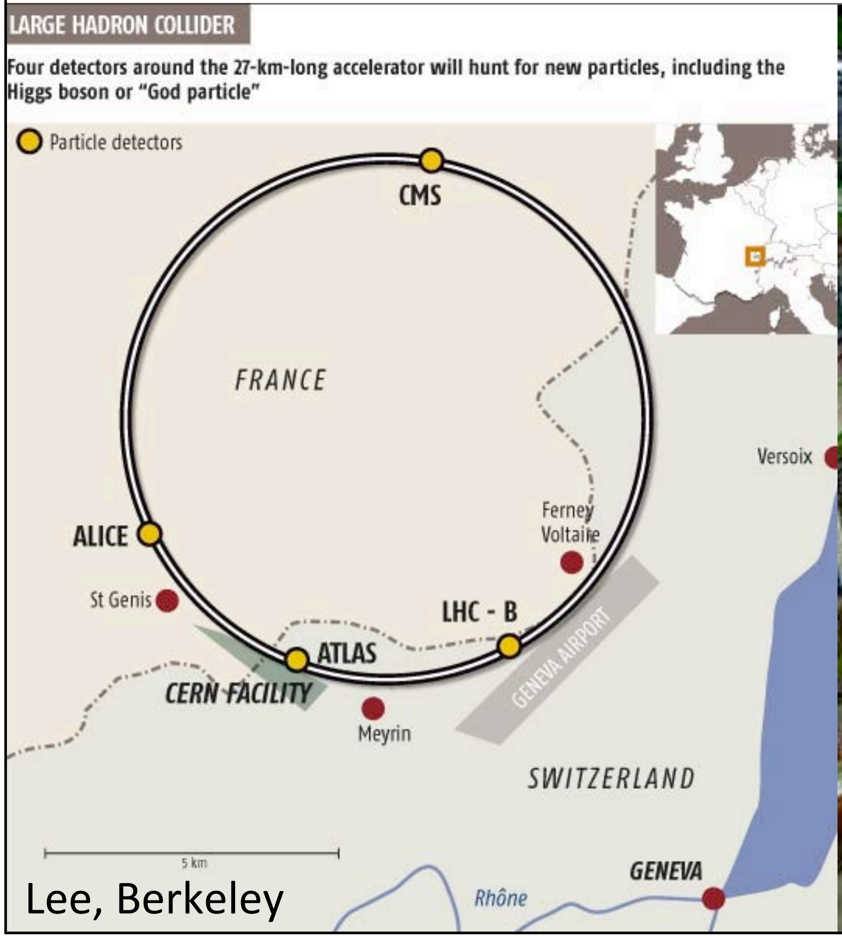
Using DP83640, Designers May Choose Any Microcontroller, FPGA or ASIC to Achieve 8- Nanosecond Precision with Maximum System Flexibility

It is routine for physical network interfaces (PHY) to provide hardware support for PTPs.

With this first generation PHY, clocks on a LAN agree on the current time of day to within ns, far more precise than GPS older techniques like NTP.

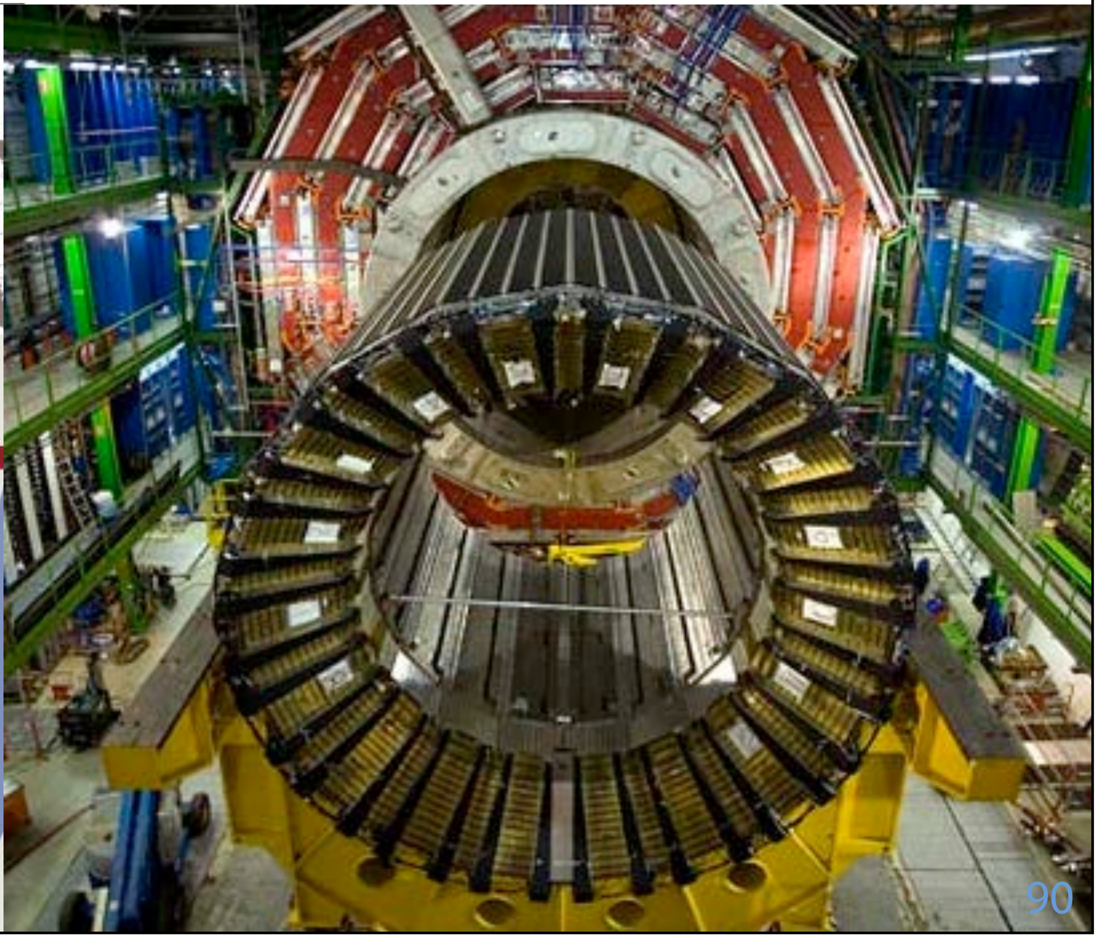
An Extreme Example: The Large Hadron Collider

The WhiteRabbit project at CERN is synchronizing the clocks of computers 10 km apart to within 10s of psec using a combination of GPS, IEEE 1588 PTP and synchronous ethernet.

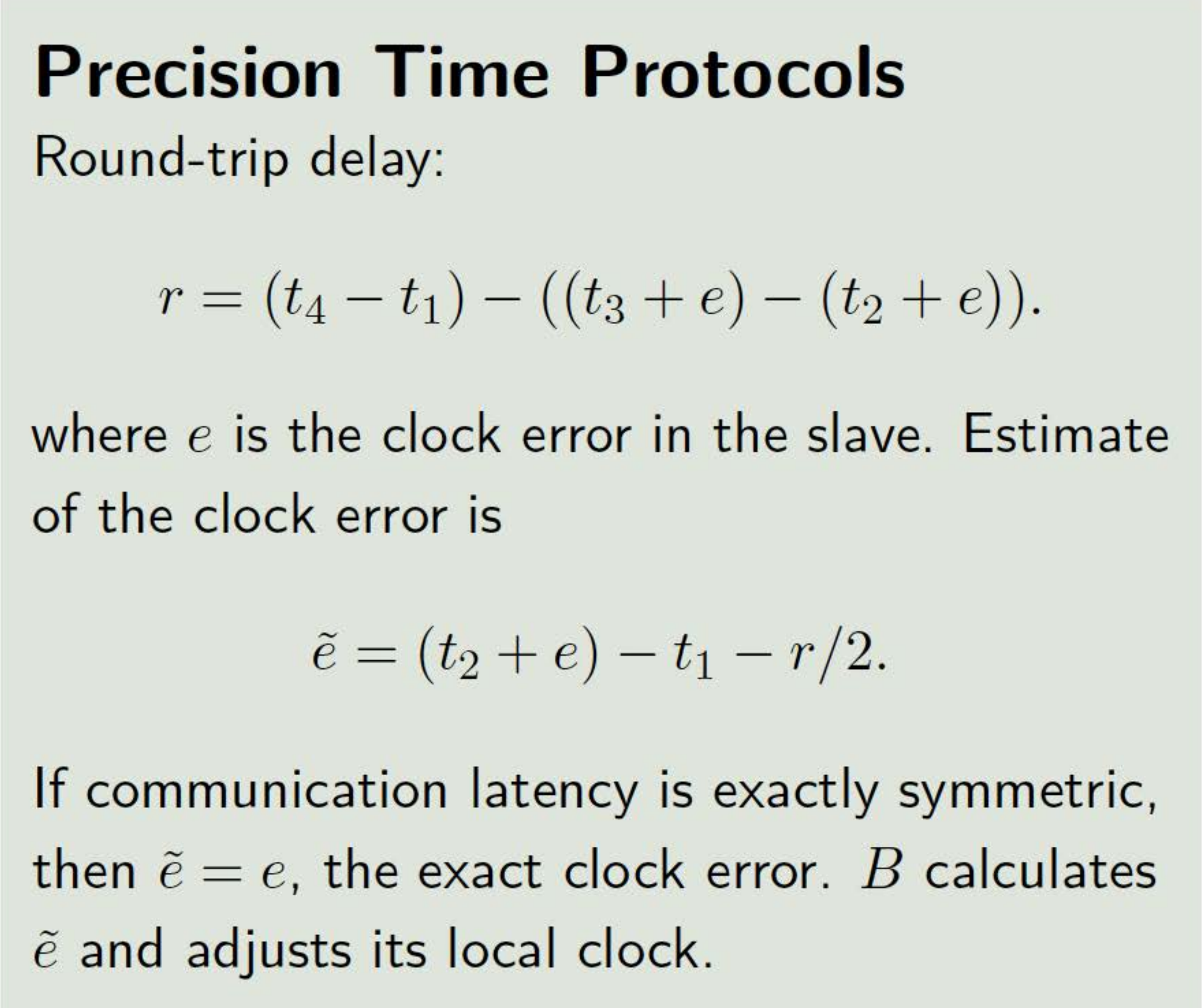


LARGE HADRON COLLIDER

Four detectors around the 27-km-long accelerator will hunt for new particles, including the Higgs boson or "God particle"



How PTP Synchronization works



Precision Time Protocols

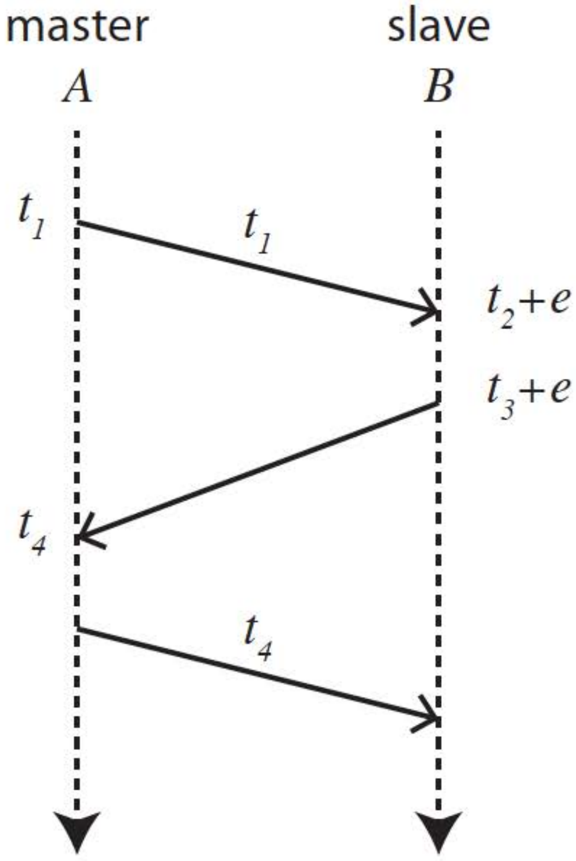
Round-trip delay:

$$r = (t_4 - t_1) - ((t_3 + e) - (t_2 + e)).$$

where e is the clock error in the slave. Estimate of the clock error is

$$\tilde{e} = (t_2 + e) - t_1 - r/2.$$

If communication latency is exactly symmetric, then $\tilde{e} = e$, the exact clock error. B calculates $\tilde{e}$ and adjusts its local clock.



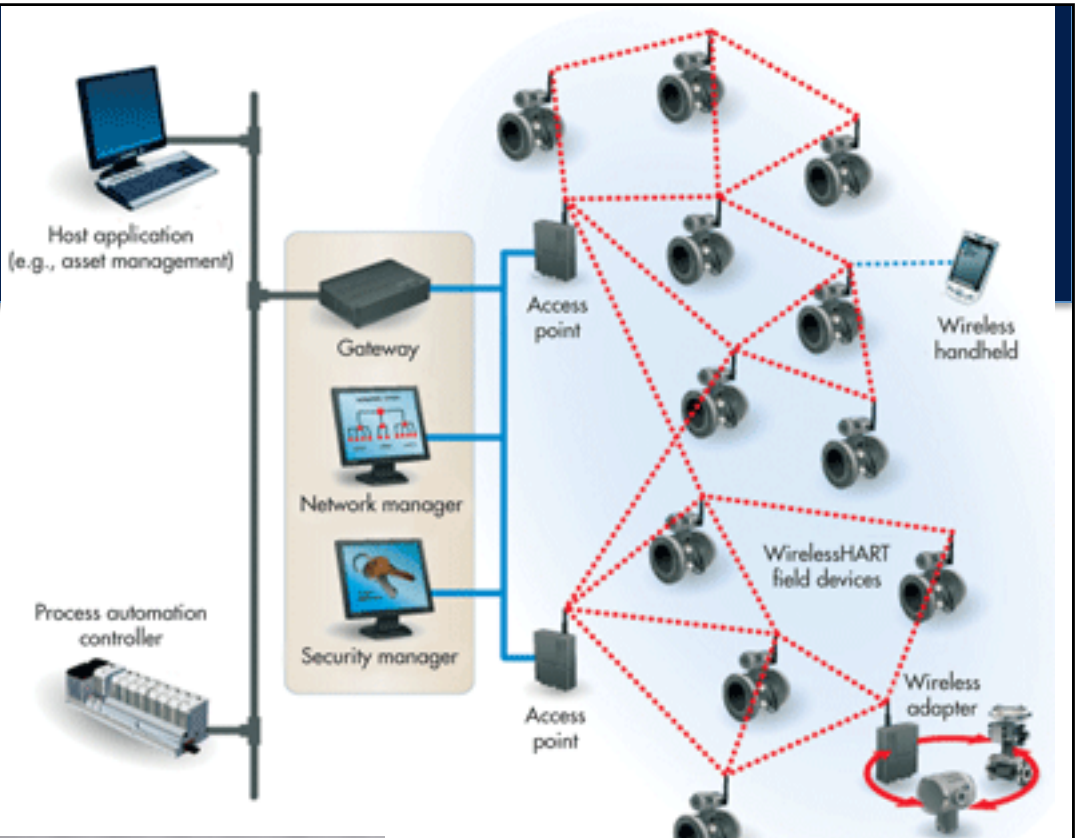
IEEE 1588,
IEEE 802.1AS

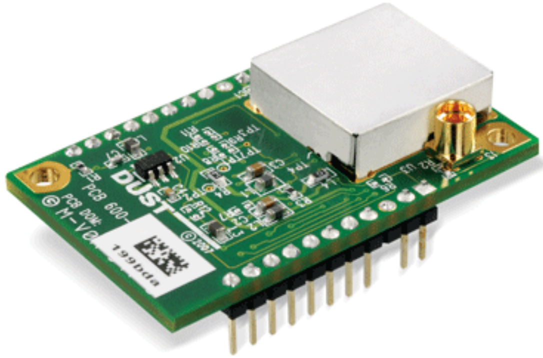
Clock Synchronization Enables:

- Energy efficiency
- Coordination
- Security
- Resource management
- Determinism

Energy Efficiency

Wireless HART uses Time Synchronized Mesh Protocol (TSMP) in a Mote-on-Chip (MoC), from Dust Networks Inc.





Clock Synchronization Enables:

- Energy efficiency
- Coordination
- Security
- Resource management
- Determinism

AVB – Audio Video Bridge

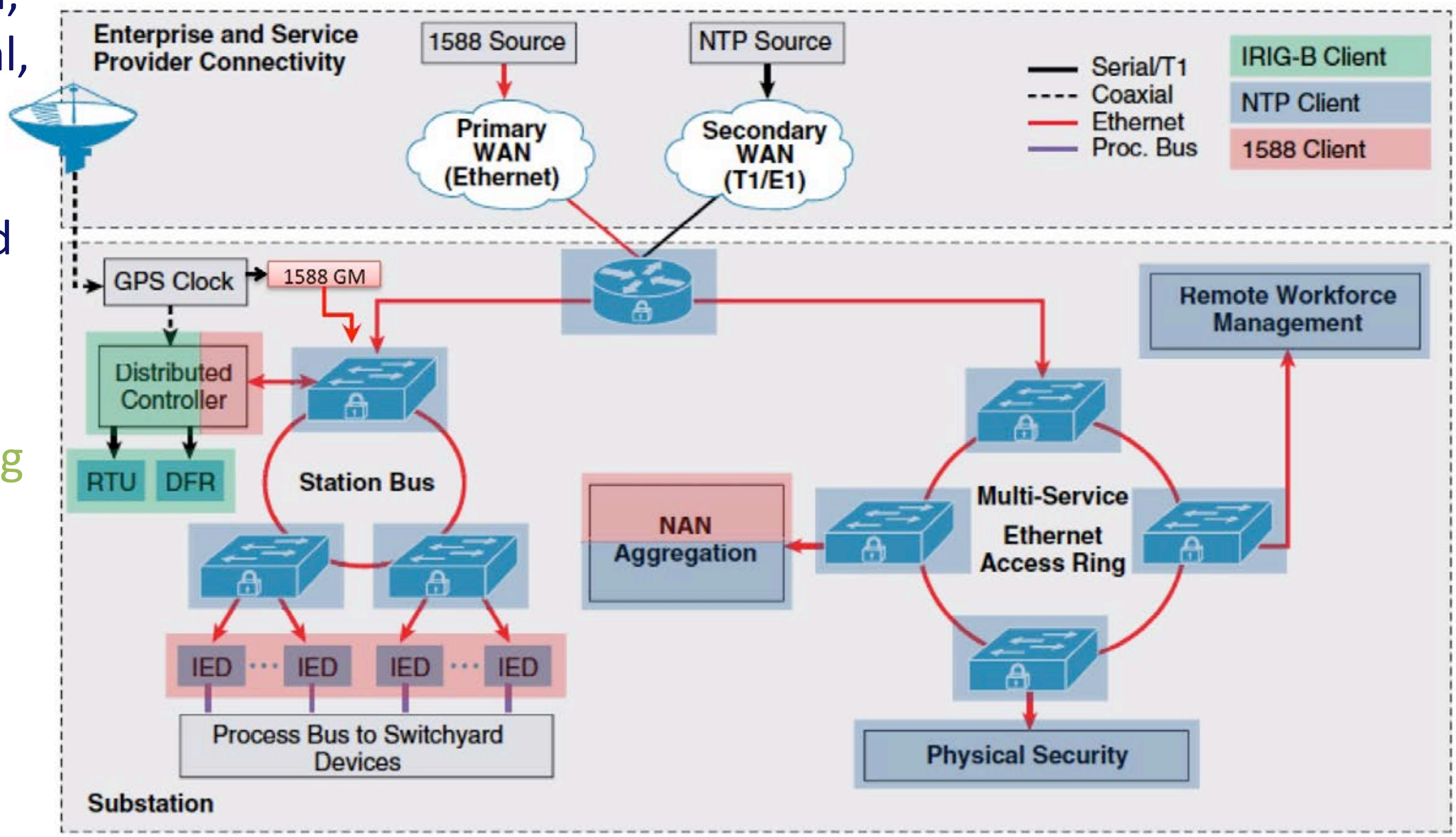
IEEE 802.1AS: Precise Synchronization

Meyer Sound CAL
(Column Array Loudspeaker),
based on research at CNMAT
(UC Berkeley)

Electric Power Generation and Distribution

Distributed power generation (wind, solar, geothermal, etc.) requires synchronized access to the grid

http://developer.cisco.com/web/tad/sample-solutions-2



Substation Timing Synchronization Using IEEE-1588 Power Profile (Cisco)

Lee, Berkeley

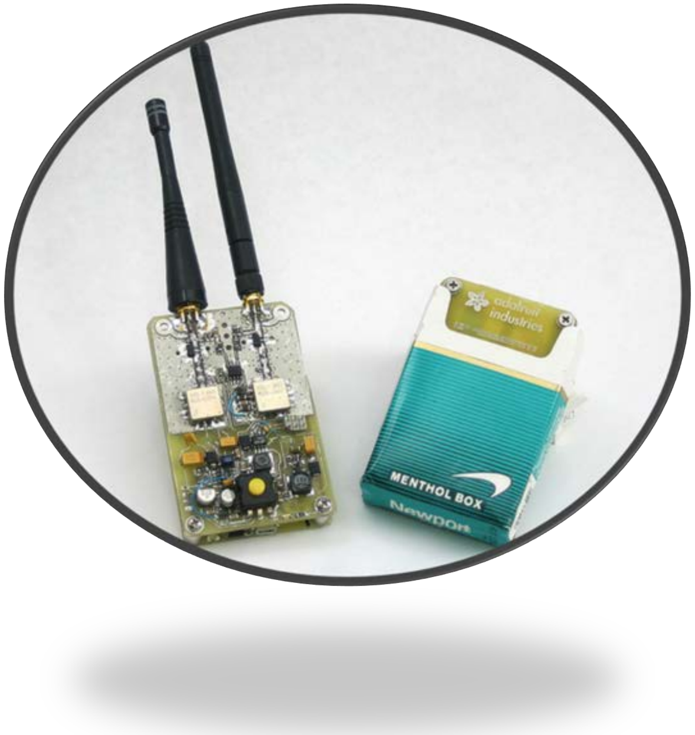
Clock Synchronization Enables:

- Energy efficiency
- Coordination
- Security
- Resource management
- Determinism

Security

- Increased vulnerability
  - – Denial of service attacks (DoS)
  - – Spoofing PTP
  - – Spoofing and jamming GPS
- Decreased vulnerability
  - – Coordination without communication
  - – Detection of DoS
  - – Detection of spoofing

Security: GPS Jammers



courtesy of
Kyle D. Wesson, UT Austin

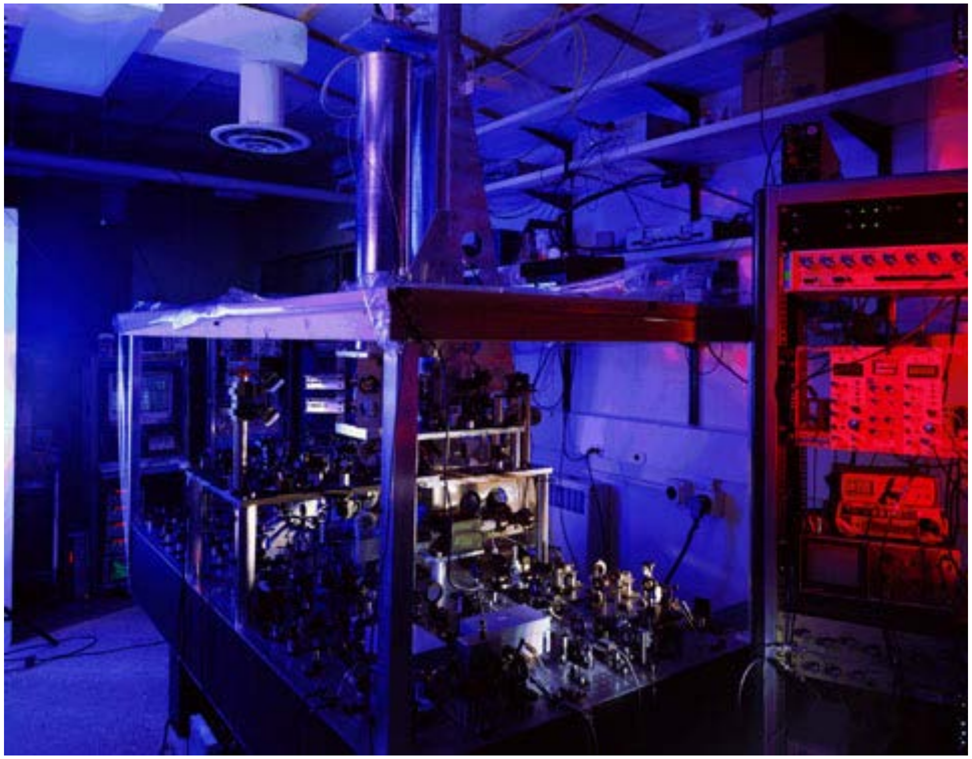
Security: GPS Spoofer

Todd Humphreys' GPS spoofing UAV (UT Austin)

Security: Stable Clocks

For a price, local clocks can be made arbitrarily stable.

NIST-F1 may drift one second in 20 million years



Coordination without Communication

With stable local clocks you can:

- Prevent packet losses.
- Detect hardware failures.
- Detect denial of service.
- Detect GPS and PTP spoofing.
- Coordinate w/out communicating.

Clock Synchronization Enables:

- Energy efficiency
- Coordination
- Security
- Resource management
- Determinism

Resource Management

Suppose that we did not all agree on the time of day (no watches or clocks).

How would you manage use of these tennis courts?

ALOHA?

The last big time synchronization push (1800s) was driven by resource allocation problems

"On August 12, 1853, two trains on the Providence & Worcester Railroad were headed toward each other on a single track. The conductor of one train thought there was time to reach the switch to a track to Boston before the approaching train was scheduled to pass through. But the conductor's watch was slow. As his speeding train rounded a blind curve, it collided head-on with the other train—fourteen people were killed. The public was outraged. All over New England, railroads ordered more reliable watches for their conductors and issued stricter rules for running on time."

Source: National Museum of American History

Today’s Networks

Today’s general-purpose networks manage resources without coordinated time.

Clock Synchronization Enables:

- Energy efficiency
- Coordination
- Security
- Resource management
- Determinism

Other Possible Topics

- Physical actions
- Superdense time
- Memory management (reference counting)

Outline

- Motivating Problems
- Why Existing Methods Fall Short
- Ports, Hierarchy, Models of Computation
- The Lingua Franca Language
- Distributed Execution
- Clock Synchronization
- Conclusion

Summary

- Lingua Franca programs are **testable** (timestamped inputs -> timestamped outputs)
- LF programs are **deterministic** under *clearly stated assumptions*.
- Violations of assumptions are **detectable** at run time.
- Actors, Pub/Sub, SoA, and shared memory have **none** of these properties.

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

Engineering Models for Real-Time Cyber-Physical Systems

- **PRET**: time-deterministic architectures
  - http://chess.eecs.berkeley.edu/pret
- **PTIDES**: distributed real-time software
  - http://chess.eecs.berkeley.edu/ptides
- **Lingua Franca**: a programming model
  - https://github.com/icyphy/lingua-franca

These enable models with tightly controlled timing and deterministic behaviors.

We have shown that that these models are practically realizable at reasonable cost.