Using Timestamps for Deterministic Distributed Software

Edward A. Lee

ISAE Supaero

Toulouse, France, Feb. 6, 2020

With thanks to:
Janette Cardoso
Marten Lohstroh
Martin Schoeberl
Andrés Goens
Mehrdad Niknami
Christopher Gill
Marjan Sirjani
Matt Weber
The major challenge: **Integrating complex subsystems** with adequate **reliability**, **repeatability**, and **testability**.
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”?

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”?
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, Thrift, …)
- 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?
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.

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).
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, …
Hewitt/Agha Actors

Data + Message Handlers

[Image of a diagram showing actors with message queues and handlers.]

- Messages In: X.A(args), X.B(args), X.A(args)
- Messages Out: Y.C(args), Y.D(args)

Example with Two Actors

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

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

Actor Door {
    handler open_door() {
        ...
    }
    handler disarm_door() {
        ...
    }
}
Example with Three Actors

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

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

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

Actor Door {
    handler open_door() {
        ...
    }
    handler disarm_door() {
        ...
    }
}
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)

Correct behavior is now defined: Process events in timestamp order.
Discrete Events (DE)

- Events that are processed in timestamp order.
- Widely used in simulation
- Foundation of hardware description languages.
- A deterministic concurrent MoC.
- But how to realize on distributed machines?

A few texts that use the DE MoC
Example: Google Spanner
A Globally Distributed Database

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

Update to a record comes in. Time stamp $t$.

Query for the same record comes in. Time stamp $r$. 
Semantics of the database is that it handles queries in timestamp order.

One Possible Approach: Chandy and Misra [1979]

- Assume events arrive reliably in timestamp order.
- Wait for events on each input.
- Process the event with the smaller timestamp.
- E.g. \( r_1 < t_1 \)
One Possible Approach: Chandy and Misra [1979]

- Deterministic
- Network traffic for “null messages.”
- Every node is a single point of failure.
Another Possible Approach: Jefferson: Time Warp [1985]

- Speculatively execute.
- If a message with an earlier timestamp later arrives...
Another Possible Approach: Jefferson: Time Warp [1985]

- Speculatively execute.
- If a message with an earlier timestamp later arrives…
- Backtrack!
Another Possible Approach: Jefferson: Time Warp [1985]

- No single point of failure.
- Can process events without network traffic
- Can’t backtrack side effects.
- Overhead: Snapshots
- Uncontrollable latencies.
A Third Possible Approach: High Level Architecture (HLA)

- Next event request (NER) with \( r \)
- Next event request (NER) with \( t \)
- If \( r < t \), then time advance grant (TAG) of \( q \leq r \)
- If \( q = r \), process event
A Third Possible Approach: High Level Architecture (HLA)

- Deterministic.
- RTI is a single point of failure.
- Works well for simulation, but not for online processing.
Ptides/Spanner Approach

- Local clock on each platform.
- \( t \) and \( r \) from local clocks.
- Bounded execution time \( W \).
- Bounded network latency \( L \).
- Event is known at B by time \( t + W + L \) (by clock at A).
- Bounded clock synchronization error \( E \).
- Event is known at B by time \( t + W + L + E \) (by clock at B).

- Event with timestamp \( r \) is safe to process time \( r + W + L + E \) (by clock at B).
Ptides/Spanner Approach

- No single point of failure.
- Can process events with *no network traffic*.
- Latencies are well defined.
- Time thresholds computed statically.
- Assumptions are clearly stated.

Ptides

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

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

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
  - These are becoming ubiquitous
- Bounded network latency
  - Violations are faults. They are detectable.
- Bounded execution times
  - Only needed in particular places.
  - Solvable with PRET machines (another talk).
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 any other component sends “open,” 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.

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"
Today: Lingua Franca

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

Lingua Franca Wiki

Topics

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

Papers

- EMSOFT 2019 work-in-progress paper.
- DAC 2019 paper on Reactors.

https://github.com/icyphy/lingua-franca/wiki
Hello World in Lingua Franca

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

- **Target language (currently C, C++, and TypeScript. Plans for Python, Rust, Java)**
- **Arbitrary code in the target language.**
- **Events of various kinds trigger reactions**
reactor A {
  output y;
  ...
}
reactor B {
  input x;
  ...
}
main reactor C {
  a = new A();
  b = new B();
  a.y -> b.x;
}
Application Sketch

![Diagram of Application Sketch]

SensorA → ComputationA → ComputationC → ActuatorA
SensorB → ComputationB → ActuatorB
Reactors

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

\begin{verbatim}
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);
    }
}
\end{verbatim}

Whether the two triggers are present simultaneously depends only on their timestamps, not 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.
        =}
}
Event-Triggered Behavior

```c
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, a_value);
    =}
}
```

Timestamp will be derived from the local physical clock.

ISR executes asynchronously, and `schedule()` function is thread safe.
reactor ActuatorA {
    input in:int;
    reaction(in) {
        perform actuation.
        => deadline 10 msec {
            handle deadline violation.
        }
    }
}
Still early, but evolving rapidly.
• Eclipse/Xtext-based IDE
• C, C++, and TypeScript targets
• Code runs on Mac, Linux, Windows, and bare iron
• Command-line compiler
• Regression test suite
• Wiki documentation

https://github.com/icyphy/lingua-franca
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 can work without clock synchronization by reassigning timestamps to network messages.
  - In this case, determinism is preserved within each multicore platform, but not across platforms.
Work in Progress

- Distributed execution based on Ptides.
- EDF scheduling on multicore.
- Targeting PRET machines for hard real time.
- Formal verification of Lingua Franca apps
- Leverage Google’s Protobufs and gRPC.
  - Complex datatypes
  - Polyglot systems
PRET targets

PRET machines: microarchitectures with precise timing control.

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

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

• 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