Preprint: To appear in "Concurrent Programming, Open Systems and Formal Methods: Essays Dedicated to Prof. Gul Agha to Celebrate his Scientific Career," José Meseguer, Carlos A. Varela, and Nalini Venkatasubramanian, Eds., Springer, LNCS, 2025

Logical Time in Actor Systems

Edward A. Lee[0000−0002−5663−0584]

UC Berkeley, Berkeley, CA, 94720, USA
eal@berkeley.edu

Abstract. The nondeterministic ordering of message handling in the original actor model makes it difficult to achieve the consistency across a distributed system that some applications require. This paper explores a number of mitigations, focusing primarily on the use of logical time to define a semantic ordering for messages. A variety of coordination mechanisms can ensure that messages are handled in logical time order, but they all come with costs. A fundamental tradeoff (the CAL theorem) makes it impossible to achieve consistency without paying a price in availability, where the price depends on the latencies introduced by network communication, computation overhead, and clock synchronization error. This paper shows how to use the Lingua Franca coordination language to navigate this tradeoff, and particularly how to ensure eventual consistency while bounding unavailability with manageable risk.

Keywords: actors · distributed systems · consistency vs. availability.

Prologue

This paper is dedicated to my friend, colleague, and mentor, Gul Agha, and offered as a contribution to his Festschrift. Gul's work on concurrent and distributed programming models has been and continues to be an inspiration.

1 Introduction

Gul Agha's actor model of computation [4,6,5,1,3,2] provides an elegant way to express concurrent and distributed programs. It has had enormous influence through realizations in Erlang [8], Akka [38], Rebeca [40], CAF [13], and many other software frameworks. The original model has inspired many variants, including several that strive for more deterministic computation. Several of these are realized in the Ptolemy II system [37], for example.

In the actor model, components (called actors) maintain local state and communicate via asynchronous message passing. When messages arrive, procedures (message handlers) are invoked in a mutually exclusive fashion. The principal source of nondeterminism in the model is that when messages arrive from a multiplicity of other actors, the order in which they are handled is not defined. While there are many applications for which such nondeterminism is not

problematic (and may even be a feature), some applications require more control over the possible behaviors. In particular, many applications require some form of *consistency*, which we can think of broadly as some form of agreement on shared knowledge across a distributed system.

Although it has proved difficult for the community to agree on the meaning of consistency [21], there is general agreement that inconsistencies often arise because distributed nodes see events in a different order. Consider two physically separated nodes that are each attempting to maintain state machines that track each other, as in, for example, a digital twin application, a replicated database, or redundant controllers for fault tolerance. Events that cause state changes can result in persistent inconsistent states if they are not handled in the same order. I develop here a very simple application that illustrates this problem and several of the many solutions that have been proposed and implemented.

The key problem addressed in this paper is that of maintaining consistency across a distributed system. Even when assuming reliable, in-order delivery of messages on a point-to-point connection, as realized for example by TCP, the widely used internet protocol, distributed components may see messages in different order when the arrive from diverse sources. This can result in inconsistent states across the system. Ensuring consistency unavoidably implies a timing penalty, a fundamental result known as the CAL theorem [29,30]. This paper shows how to augment the actor model with timestamps to ensure consistency, how to manage the resulting timing penalties, and how to keep timing penalties bounded in the presence of faults.

2 Motivating Example

Consider a brutally simplified example application that replicates data: a distributed banking system with just one account. I distill it down to the barest minimum, where there are two nodes at physically different locations. Each node manages an ATM machine that can dispense cash, and each node maintains a copy of the bank balance.

This application is perhaps the world's smallest distributed database. It maintains exactly one integer-valued record, the bank balance. It allows this record to be modified at either node, through deposits or withdrawals at the ATM machines. It requires at least *eventual consistency*, which means that if all transactions stop, then eventually the two nodes should agree on what the balance is. And it requires *availability*, in the sense that customers will not be happy if the ATM refuses to dispense cash because the network is down. But it also has some features that make it more interesting than a simple distributed database. In particular, overdrafts may be hard to prevent without sacrificing availability.

This application is typical of many distributed applications, where distributed nodes all monitor the same input data and evolve their state according to that data. The challenge is to maintain some form of consistency in their states.

We assume they will all eventually see the same input data, but not necessarily in the same order and possibly with considerable delays.

I will give a series of implementations, beginning with an actor implementation. In the actor implementation, the intrinsic nondeterminism in the model of computation is not problematic if overdrafts are not a problem. But if they are, the nondeterministic order in which messages are handled becomes a problem. Subsequent implementations offer a variety of ways to deal with the problems. I conclude by outlining the fundamental limits that this series of examples faces, arguing that there is no perfect solution. Every solution requires compromises, and which compromises to select depends very much on the application.

Each implementation is realized using the federated version [10] of the Lingua Franca (LF) coordination language [34]. I describe the features of LF as we go.

2.1 An Actor Solution

Figure 1 shows a Lingua Franca program together with its automatically-generated diagram.[1] This program uses features of LF to make the semantics of the program conformant with the actor model.

First, LF is a coordination language, not a programming language. It coordinates programs written in conventional languages. In this case, the language chosen is C, as indicated by the `target` directive on the first line. Line 19 declares the main program to be `federated`, which means that each component (called a reactor) that is instantiated within this federated reactor will become a standalone program. The programs can be run all on one machine or on several machines or cloud instances. The network communication and any required coordination are automatically generated.

In this program, there are four federate instances, instantiated on lines 20 through 23. Instances `w1` and `w2` are instances of an imported reactor class called `IntWebSocketServer`, which is not shown. That class realizes a simple web server that connects to the LF program via a web socket that sends a stream of integers to and from a browser. The browser stands in for an ATM machine, providing a simple HTML interface enabling a user to deposit and withdraw cash (see Figure 2). The details of the implementation of these reactors are not important. All we need to know is that `w1` outputs a positive integer when a user deposits cash and a negative integer when a user withdraws cash.

When the user deposits or withdraws cash (via the web interface), an integer output appears on the `received` output port of the web server reactor (see the diagram in Figure 1). This integer is forwarded to both account managers, and one of those account managers (presumably the local one) responds over the feedback connection with the balance in the account after the deposit or withdrawal is completed. The web server receives this input on its `response`

[1] Some details are omitted for brevity; full programs (and more) and all supporting files can be found at https://github.com/lf-lang/lf-demos/tree/main/federated-decentralized.

```
1  target C {coordination: decentralized}
2  import IntWebSocketServer from "lib/IntWebSocketServer.lf"
3
4  reactor ACIDAccountManager(STA: time = 0) {
5    input in1: int
6    input in2: int
7    output out: int
8    state balance: int = 0
9    reaction(in1, in2) -> out {=
10     if (in1->is_present) {
11       self->balance += in1->value;
12     }
13     if (in2->is_present) {
14       self->balance += in2->value;
15     }
16     lf_set(out, self->balance);
17   =}
18 }
19 federated reactor {
20   w1 = new IntWebSocketServer(...)
21   w2 = new IntWebSocketServer(...)
22   a1 = new ACIDAccountManager()
23   a2 = new ACIDAccountManager()
24
25   w1.received ~> a1.in1
26   w2.received ~> a2.in2
27   w1.received ~> a2.in1
28   w2.received ~> a1.in2
29   a1.out ~> w1.response
30   a2.out ~> w2.response
31 }
```

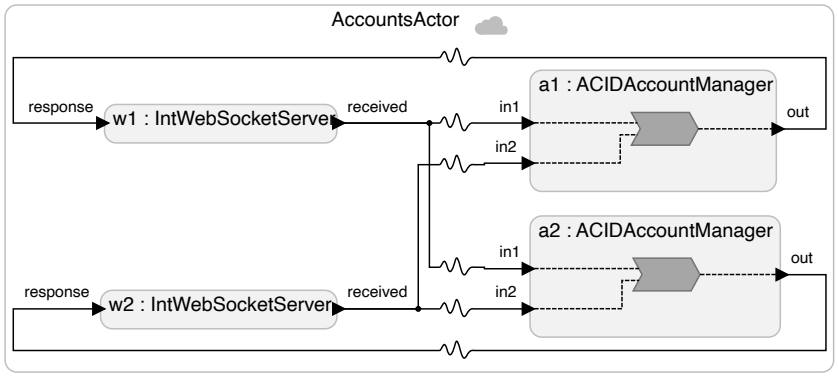

Fig. 1. A simple distributed banking system in the actor style realized in Lingua Franca.

input port. The web server reactor then sends this balance over a web socket to the browser, which will display it to the user as the reported balance.

The account manager instances `a1` and `a2` are instances of the reactor class called `ACIDAccountManager` defined starting on line 4. This class defines a single `reaction`, which is triggered by a message on either of its input ports, `in1` or `in2`. The code in the reaction body, surrounded by the delimiters `{= ... =}`, is C code. It uses LF C target API to check whether each input port has a message

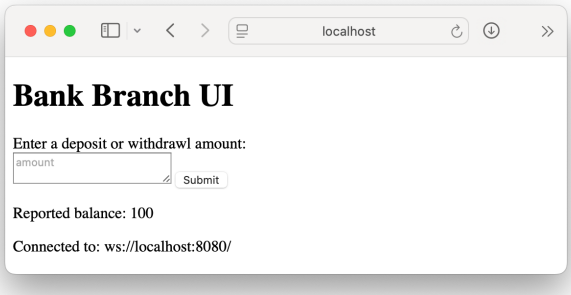

Fig. 2. Our crude browser-based user interface standing in for an ATM machine.

to process, and, if it does, to add the value of that message to its state variable named `balance`. It then produces the resulting balance on its output.

The connection statements on lines 25 through 30 establish connections between the reactor instances, as shown in the diagram. These connections are of a particular type in LF called a *physical connection*, created with the syntax ~> and depicted in the diagram with a squiggly line. A physical connection in LF uses TCP to reliably deliver messages in order, but otherwise there is no coordination between messages. Each account manager here has two inputs, one coming from a local web server (its ATM machine) and the other coming from a remote web server (a remote ATM machine). Because of the physical connections, there is no ordering semantics to these messages, and they will be handled in whatever order they happen to be received. Hence, the reaction starting on line 9 has similar semantics to a message handler in an actor language. It handles incoming messages in the order in which they are received.

Now, suppose these federates are deployed so that one web server and account manager are in Singapore and the other pair is in London. Each account manager maintains its own copy of the bank balance. It is reasonable to demand that these balances be consistent, but what exactly do we mean by that? A naive approach would be to insist that the two balances agree at an instant in time. But there is no such thing as an "instant in time" for a distributed system, even in theory [36], but much less in practice. And even if we could perfectly synchronize a clock in Singapore with one in London, which we cannot [19], communication delays would have to be accounted for.

The particular algorithm realized in the program, however, has some rather nice properties. Ignoring the possibility of integer overflow in C, the operations performed by the reaction starting on line 9 are associative and commutative. Because the network communication is realized with a TCP socket, the operation is effectively also idempotent. In distributed systems, idempotence means that if a message is delivered more than once, it has the same effect as if it had been delivered exactly once. The TCP connection guarantees exactly once delivery. Hence, handling of messages satisfies the properties that Helland and

Campbell call "ACID 2.0" [22], a revision of the classic ACID database properties (atomicity, consistency, isolation, and durability) to stand for associative, commutative, idempotent, and distributed. This means that regardless of network latencies and nondeterministic message ordering, as long as both account managers eventually see the same messages, they will eventually agree on the balance.

These ACID 2.0 properties have been identified by Shapiro et al. as realizing what they call a "conflict free replicated datatype" (CRDT) [39]. The `ACIDAccountManager` with its state variable `balance` and its operation defined by its reaction is, perhaps, the world's simplest CRDT.

A slightly tortured interpretation also allows us to see the operation of the account manager is being monotonic in a partial order, which by the CALM theorem [7,28] then guarantees eventual consistency. Under this interpretation, each account manager's balance is simply a compact representation of a multiset of all deposits and withdrawals that it has seen, and each operation simply adds a number to the multiset. Set union is a monotonic operation in a partial order defined by the subset relation, and hence the CALM theorem applies, albeit in a rather tortured way.

All of this gives us beautiful theory, but it does not give us something any bank would actually use. This design would permit customers to withdraw any amount of money irrespective of their balance. Unfortunately, fixing this flaw also loses most of these nice properties, as I explore next.

2.2 An Inconsistent Solution

Suppose that we replace the reaction in the account manager with the one shown in Figure 3. This reaction implements a particular (arbitrary) business logic that denies withdrawals greater than the balance and applies a $30 penalty for any attempt to overdraw the account. This operation is no longer commutative, so, when put in the structure shown in Figure 1, leads to nondeterministic disagreements about the balance in the account. Suppose that a deposit of $200 takes place in London nearly simultaneously with a withdrawal of $200 in Singapore. It is possible that one account manager will deem an overdraft to have occurred while the other did not. The two will now permanently disagree on the balance.

Fixing this inconsistency requires considerable sophistication in distributed computing. One relatively simple solution is to maintain the balance at exactly one node rather than in a distributed fashion. This has a cost of creating a single point of failure, and communication latencies or network outages can result in unavailability. Even this solution, however, requires paying careful attention to atomicity, for example using transactions [20,26]. The designer has to be careful that a withdrawal at another site cannot occur between querying for the balance and granting a local withdrawal, for example.

Distributed transactions offer a more sophisticated solution, but the algorithms are tricky and getting good performance is challenging [17]. Here, I offer

an alternative approach, which is to enhance the semantics of actors with timestamps that define an order for messages. I then explore infrastructure-level middleware techniques that enforce the ordering. These are not intrinsically simpler than other distributed computing techniques, but they make application development simpler by absorbing many of the complexities into the infrastructure. The application designer can focus on realizing the business logic rather than on the intricacies of distributed computing.

2.3 Using Timestamps

If a deposit occurs in London nearly simultaneously with a withdrawal in Singapore, how should we determine whether an overdraft has occurred? Fundamentally, it is not possible to unambiguously determine the order in which two closely-timed events occur at two distinct physical locations. The theory of relativity tells us that the ground-truth order may depend on the observer. Fortunately, we do not need to find a ground-truth order. We just need for the components of the distributed system to *agree* on the order and for the resulting order to appear reasonable to human observers.

For this purpose, let us assume that the computers in London and Singapore each have local clocks that are reasonably well synchronized with one another, using for example the network time protocol (NTP) [35], the precision time protocol (PTP) [19], or GPS, for example. We can use these physical clocks to assign logical timestamps to the deposits and withdrawals. A Lingua Franca program

```
reactor AccountManager(STA: time = 0) {
  input in1: int
  input in2: int
  output out: int
  state balance: int = 0

  reaction(in1, in2) -> out {=
    int in1_val = 0;
    int in2_val = 0;
    if (in1->is_present) {
      if (self->balance >= -in1->value) {
        self->balance += in1->value;
      } else {
        self->balance -= 30; // Apply penalty
      }
    }
    if (in2->is_present) {
      if (self->balance >= -in2->value) {
        self->balance += in2->value;
      } else {
        self->balance -= 30; // Apply penalty
      }
    }
    lf_set(out, self->balance);
  =}
}
```

Fig. 3. An inconsistent account manager.

```
1  federated reactor {
2    w1 = new IntWebSocketServer(...)
3    w2 = new IntWebSocketServer(...)
4    a1 = new AccountManager()
5    a2 = new AccountManager()
6
7    w1.received -> a1.in1
8    w2.received -> a2.in2
9    w1.received -> a2.in1
10   w2.received -> a1.in2
11   a1.out -> w1.response
12   a2.out -> w2.response
13 }
```

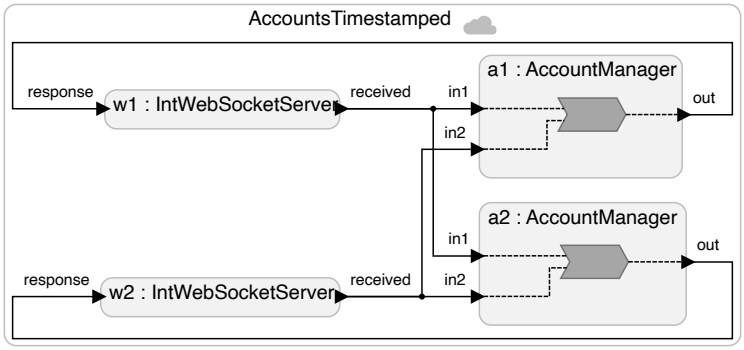

Fig. 4. Our distributed banking system with timestamped communication.

that does this is shown in Figure 4. This is identical to the program in Figure 1 except that the connections between components now carry timestamps, and the non-commutative account manager of Figure 3 is used. The use of timestamps is specified in the program using the connection syntax -> instead of ~> on lines 7 through 12, which results in the diagram shown in Figure 4. The diagram shows straight line connections instead of squiggly lines.

The Lingua Franca semantics now defines a correct execution of the program to be one where every reactor handles messages in timestamp order. Hence, a correct execution of the program in Figure 4 results in the two account managers agreeing on the value of the balance at any *timestamp*. The timestamp becomes a *logical time*, distinct from physical time [33] in that when Singapore learns about a deposit that occurred in London, that deposit will carry a timestamp that was assigned in London. The machine in Singapore will process that deposit in timestamp order relative to any other events, even local events that are assigned timestamps using its local physical clock.

The use of timestamps gives a semantics to message ordering. Timestamps are numbers, so one slight complication is that now there is a notion of two messages being *simultaneous*. They have the same timestamp. No such notion exists in the classical actor model, where messages are queued and handled one-at-a-time.

The `AccountManager` reactor defined in Figure 3 specifies a particular behavior when two inputs arrive with the same timestamp. In this case, there will be just one invocation of the reaction starting on line 7, and both `in1->is_present` and `in2->is_present` will be `true`. The logic given in this program, therefore, will give preference to a deposit or withdrawal on `in1`, honoring it first if there is sufficient balance. A withdrawal on `in2` will be honored only if the balance is sufficient after handling `in1`. Because of the way the connections are made in Figure 4, both account managers will give preference to the upper instance of the web server, and, hence, if one honors a withdrawal, so will the other, thereby maintaining consistency.

The use of timestamps here provides a stronger form of consistency than eventual consistency. This form of consistency asserts that two components agree on the value of a shared quantity *at a logical time*. As shown by Lee et al. [29,30], this consistency comes with an intrinsic cost in availability that is a function of the latency in communication (the CAL theorem). In this case, "availability" refers to the physical time one must wait before one can access the value of a shared quantity at a particular logical time. The longer one has to wait, the less available the shared quantity is. The waiting time is the price paid for ensuring a "correct" behavior, where all messages are handled in timestamp order. I now address the mechanisms provided by Lingua Franca for achieving a correct behavior and for relaxing consistency to improve availability.

3 Achieving Correct Behavior

Now that we have defined "correct behavior," how do we achieve it? In Lingua Franca, by default, when you create a federated program, the distributed execution relies on a single, centralized coordinator called the RTI, for runtime infrastructure [10]. When a node wishes to process a timestamped event, it queries the RTI for permission, and the RTI grants permission only when it can assure that the node has already received all messages with lesser timestamps. This coordination strategy ensures our strong form of consistency. However, it has considerable disadvantages. The RTI is a single point of failure, a potential bottleneck for communication, and a source of additional latency because of the required two-way communication. Consequently, I focus here on an experimental decentralized coordinator for federated Lingua Franca programs. I will show how this coordinator can be used to achieve a variety of strategies with different tradeoffs.

3.1 Decentralized Coordination in Lingua Franca

Lingua Franca offers an alternative coordination strategy that is decentralized [10]. A programmer declares to use the decentralized coordinator as shown on line 1 in Figure 1. Coordination is based on PTIDES [43,18], a technique that was developed in my group and independently reinvented at Google to become the backbone of Google Spanner, a globally distributed database system [14,15].

Like the (default) centralized coordinator, a runtime infrastructure (RTI) node orchestrates the startup (and shutdown) of the federation, but unlike the centralized coordinator, the RTI plays little role during the execution of the program. Its function is limited to handling requests to shut down the federation, and to performing runtime clock synchronization, if this is enabled. Many applications will disable this clock synchronization and rely instead on built-in mechanisms such as NTP, PTP, or GPS.

When using the decentralized coordinator, each federate makes its own decisions about when to advance to a timestamp and invoke reactions to events with that timestamp. To govern these decisions, there are two key parameters that a programmer can specify:

1. **STA**: The **safe to advance** offset is a physical time quantity that asserts that the federate can advance to a timestamp t when its local physical clock time T satisfies $T \geq t + \text{STA}$.
2. **STAA**: The **safe to assume absent** offset is a physical time quantity that asserts that the federate, if it has not received a message with timestamp t or larger, can assume an input to be absent at timestamp t when its local physical clock time T satisfies $T \geq t + \text{STA} + \text{STAA}$.

If we choose finite value for STA and STAA, these two thresholds are guaranteed to be satisfied eventually under only the assumption that physical time continues to advance. This can be used to ensure liveness in a distributed system. As we will see, however, sometimes it is useful to choose infinite values for STA or STAA, thereby ensuring consistency at the expense of liveness.

The STA is a property of a federate and is defined as a parameter for the reactor class:

```
reactor A(STA:time = <time value>) { ... }
```

The STAA is associated with one or more input ports, but it is declared on reactions to those input ports:

```
reactor A {
  input in:<type>
  reaction(in) {=
    <normal operation>
  =} STAA(<time value>) {=
    <fault handler>
  =}
}
```

Any network input port will have that STAA applied to it if it is a trigger for or is used by a reaction that declares an STAA handler. If more than one STAA is declared for the same input port, the minimum time value will be the one in effect.

If, for example, our account manager reactor has an STA of 100 ms, then, when it receives a deposit or withdrawal message from its local web server with timestamp t, it waits until its local clock hits $t + 100$ ms and then processes the message. In our application, this will ensure that messages are processed in a timely fashion (unavailability is limited to 100 ms), but messages will be processed "correctly" (in timestamp order) only under specific assumptions. In particular, we would have to ensure that any message with timestamp less than t from Singapore is received in London before London's physical clock hits $t +$ 100 ms. Assuming the timestamp in Singapore was assigned using Singapore's local physical clock, this implies that the sum of the clock synchronization error and the network latency (plus any processing overhead) cannot exceed 100 ms. This is quite a stringent requirement that may be difficult to achieve without dedicated networks and sophisticated clock synchronization (both used by Google Spanner).

What happens if this requirement is violated? In our particular application, it is quite possible that nothing bad will happen. Correctness is assured as long as messages are handled in timestamp order. If the deposits and withdrawals are sufficiently sparse, then it becomes extremely unlikely that two of them will be close enough in time time to trigger a fault. Nevertheless, if a message with timestamp t arrives after messages with timestamp t or larger have been processed, then the <`fault handler`> in the above code will be executed instead of the <`normal operation`> code in the reaction. If no <`fault handler`> is given for an input port that receives such an out-of-order message, then Lingua Franca issues a warning and proceeds to process the message at the wrong timestamp. For applications using CRDTs, this may not be a problem, although then it would be better to use the style of connections in Figure 1, which discard timestamps, so that no warnings are generated. But for our banking application, this strategy is not acceptable.

Choosing suitable STAs, STAAs, and fault handlers for a program is challenging and depends on many factors. I will continue to develop our running application to illustrate the reasoning that a system designer must perform. Our overarching goal is to enable business strategies that weigh the risks of failure and the quality of service while ensuring eventual consistency.

3.2 Optimistic Techniques

There is a long history, dating back at least to Jefferson's TimeWarp [23], of "optimistic" distributed computing. In such techniques, messages are processed when they are available, and, if an out-of-order message later arrives, the execution is rolled back in (logical) time and redone with messages processed in order. This requires that a snapshot state of the component be periodically saved and that inputs that arrive with timestamps greater than that of the snapshot also be saved. This strategy would be easy to implement with relatively small changes to our Lingua Franca program.

However, rolling back is not always possible. Cyber-physical systems, for example, perform actuation in the physical world that may be difficult or im-

possible to undo. It's hard to unlaunch a missile or claw back the dollars that have been dispensed by the ATM. Also, if the reactions use any legacy code or third-party libraries, creating snapshots of the state may be difficult, expensive, or impossible. As a consequence, optimistic techniques have mostly been used only for distributed *simulation*, not for actual system implementation, except in the much more limited form of transactions.

3.3 Transactions

Database-style transactions provide a more limited form of optimistic computation [20,26]. In our banking application, each node could tentatively make a decision to dispense cash, but only actually dispense cash (i.e. commit the transaction) after receiving confirmation from a quorum of other nodes. This is perhaps a reasonable approach for this application, and it can be implemented in Lingua Franca, but the logic is complex and it's easy to get the program wrong. Moreover, there will be considerable overhead, like with centralized coordination, that will increase latencies and network congestion. There are more attractive alternatives.

3.4 Conservative Techniques

There is a long history, dating back at least to Chandy and Misra [12] of conservative techniques that guarantee distributed processing of events in timestamp order. These have mostly been used for distributed simulation, but the same principles can be used in online applications like our banking system. The centralized coordinator in Lingua Franca provides such conservative coordination loosely based on the High-Level Architecture (HLA) [27,42]. But the decentralized coordinator is also capable of conservative coordination based on Chandy and Misra [12], as I will show now.

Suppose we set STA to `forever`, which is done in Lingua Franca as follows:

```
reactor A(STA:time = forever) { ... }
```

This tells any instance of this reactor that, in order to advance its logical time to some t and handle a message with that timestamp, it must know the status of *all* inputs up to and including timestamp t. If the input status is unknown, the federate must wait until it is known, with no upper bound on the wait time.

How does the status of an input port become known up to and including logical time t? First, recall that TCP underlies all communication between federated programs in LF. TCP provides a semantics of reliable, in-order delivery. Moreover, LF assures that messages sent along any connection between federates are sent in timestamp order, and that at most one message bears a particular timestamp t. Hence, when a federate receives a message with timestamp t, it is assured that it has seen all messages up to and including timestamp t for this connection.

```
target C {  coordination: decentralized }
import IntWebSocketServer from "lib/IntWebSocketServer.lf"
import AccountManager from "lib/AccountManager.lf"

reactor Server(hostport: int = 8080, initial_file: string = {= NULL =},
     null_message_period: time = 1 s) {
  input in: int
  output received: int
  timer t(0, null_message_period)
  w = new IntWebSocketServer(hostport = hostport, initial_file = initial_file)
  in -> w.response
  reaction (w.received, t) -> received {=
    if (w.received->is_present) {
      lf_set(received, w.received->value);
    } else {
      // Send a null message.
      lf_set(received, 0);
    }
  =}
}
federated reactor {
  w1 = new Server( ... )
  w2 = new Server( ... )
  a1 = new AccountManager(STA = forever)
  a2 = new AccountManager(STA = forever)
  w1.received -> a1.in1
  w2.received -> a2.in2
  w1.received -> a2.in1
  w2.received -> a1.in2
  a1.out -> w1.in
  a2.out -> w2.in
}
```

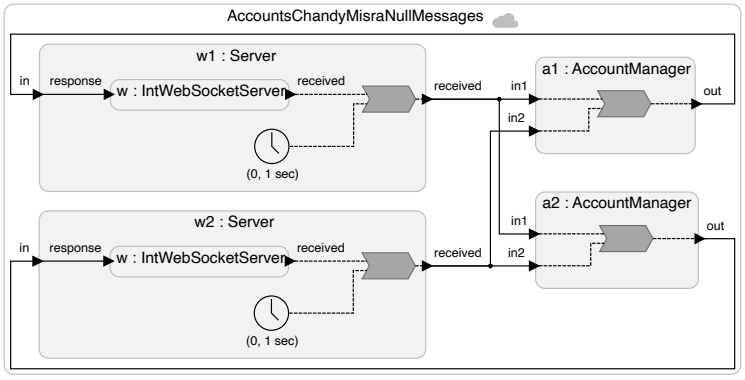

Fig. 5. Our distributed banking system with Chandy and Misra style coordination.

In this application, messages are sent only when a deposit or withdrawal is made. This means that if `a1` receives a withdrawal request on `in1` with timestamp t, it must wait until it receives a deposit or withdrawal on `in2` with timestamp at least t. This wait time is not bounded and will not be acceptable in this application. The Chandy and Misra technique, therefore, adds the notion of a "null message" that is sent periodically.

Figure 5 shows an implementation using the Chandy and Misra technique with null messages. The `IntWebSocketServer` reactors are wrapped in another

reactor that, in addition to deposit and withdrawal messages, sends periodic null messages (which in this case are just the integer zero, indicating no deposit or withdrawal). On lines 23 and 24, the STA is set to `forever` for the two account managers, which tells them that they must wait, with no upper bound on the wait time, until all inputs are known up to and including logical time t before processing any message with timestamp t.

The null messages are triggered by a Lingua Franca `timer`, declared on line 8, which has an offset of zero (the timer starts when the program starts) and a period of one second. The reaction that starts on line 11 is triggered by *either* a deposit or withdrawal from the web server or the timer (or both). If there is a real deposit or withdrawal, it simply forwards it to the output, and otherwise it sends a null message.

With this strategy, the wait time in the account managers is bounded. However, this strategy creates a tight coupling between the components. If one of the `Server` federates fails, for example, then the null messages will stop and the account managers will both be blocked. Hence, although this strategy guarantees strong consistency, it does so at a potentially unbounded cost in availability.

The CAL theorem tells us that there is no strategy that guarantees strong consistency without a potentially unbounded cost in availability [29,30]. To bound unavailability, we have to relax consistency. I explore how to do that next.

3.5 Trading Off Consistency and Availability

It is impossible to achieve strong consistency and bounded unavailability under all network behaviors. For this application, eventual consistency is a requirement for at least the `balance` state variable. All copies of the bank balance must eventually agree. But we can design a system that provides bounded unavailability for at least some transactions.

Figure 6 shows a design that responds in bounded time to deposits, but not necessarily to withdrawals. The `Account` reactors in this design are the same as the `AccountManager` reactors in Figure 5. They have STA set to `forever`, and hence will maintain strongly consistent balances. The `Server` reactors, like those in Figure 5, send null messages periodically.

The `QuickDeposit` reactors, which are defined by the code shown in the figure, have STA set to 30 ms. These reactors receive inputs from the local web server, and, if the input is positive, immediately process the deposit and send an acknowledgement back to the web server. Line 11 checks whether the transaction is a deposit, and it it is not, then the reactor ignores the input. This results in a quick response to a deposit regardless of network conditions.

The `QuickDeposit` reactors also have a `true_balance` input, which gets a message each time the `Account` reactors update their copy of the true balance. As shown on line 8, it simply updates its copy of the balance upon receiving this input. However, because the STA is only 30 ms for the `QuickDeposit` reactors and is `forever` for the `Account` reactors, it is possible for the `true_balance` input to receive a message late. Specifically, `true_balance` may receive an input with

timestamp t after the `QuickDeposit` federate has processed `deposit` inputs with larger timestamps. Hence, the `QuickDeposit` reactor is being asked to process messages out of timestamp order.

Line 16 specifies an STAA of zero. Together with the STA of 30 ms, as explained above, this means that if `QuickDeposit` receives a `deposit` input with timestamp t, and it has not received a `true_balance` input with timestamp t or greater, then when its physical clock reading T exceeds $t + \text{STA} + \text{STAA} = t + 30$ ms, it will assume that the `true_balance` input is absent and proceed to process the `deposit` input. If it later receives a `true_balance` input with timestamp less than or equal to t, then the requirement to handle inputs in timestamp order has been violated, and the exception code starting on line 17 will be invoked instead of the regular reaction.

The exception handling code starting on line 17 handles `true_balance` inputs just the same as if there were no error. The business logic here is that the

```
1  reactor QuickDeposit(STA: time = 30 ms) {
2    input deposit: int
3    input true_balance: int
4    output ack: int
5    state balance: int = 0
6    reaction(true_balance, deposit) -> ack {=
7      if (true_balance->is_present) {
8        self->balance = true_balance->value;
9      }
10     if (deposit->is_present) {
11       if (deposit->value >= 0) {
12         self->balance += deposit->value;
13         lf_set(ack, self->balance);
14       }
15     }
16   =} STAA(0) {=
17     if (true_balance->is_present) {
18       self->balance = true_balance->value;
19     }
20     if (deposit->is_present) {
21       // ERROR. Take ATM offline for service?
22     }
23   =}
24 }
```

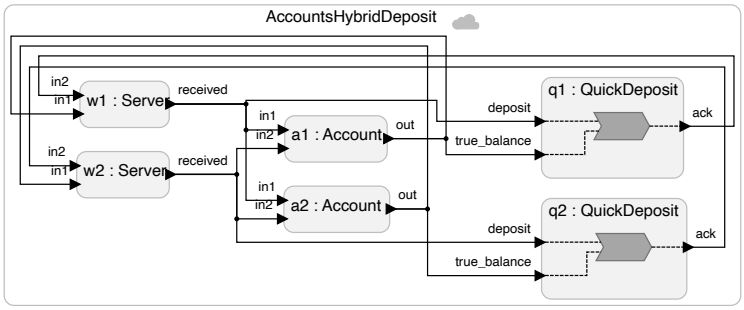

Fig. 6. Distributed banking system with strong consistency and guaranteed availability for deposits.

balance is used only to report back to the user what the balance is after the deposit, and if this balance is inaccurate because of delayed messages from the remote server, this will only result in transient errors. This code handles late `deposit` inputs as errors, however. The logic here is that the system is designed to give low delays between the server and the `QuickDeposit` reactors, and if the delays get large, something is wrong with the local system.

Note now that the design of `QuickDeposit` is governed more by *business* decisions than by the intricacies of distributed computing. An alternative design, for example, would also handle small withdrawals quickly and only delay the handling of large withdrawals. This is a business decision to accept some risk in exchange for better, more reliable customer service. Many other designs now become possible (and easy in Lingua Franca).

3.6 Fault Tolerance

Any federate with an STA of `forever` is vulnerable to blocking indefinitely if its upstream federate fails. For the example in Figure 6, this will only block withdrawals (or large withdrawals, if that option is chosen), which may be acceptable business logic. A better design, however, would use a smaller STA and use the exception handler to handle fault conditions. For example, the "true balance" could be defined by a quorum of nodes rather than by unanimous consent, as done here. (A quorum makes little sense with just two nodes, but a real application will have many more.) The exception handling code could make corrections late to the "true balance," but also alert human maintainers to instability in the system. As long as all nodes agree on which deposits and withdrawals have been actually executed, none of these strategies will compromise eventual consistency.

4 Zero-Delay Cycles

In the above examples, we have not had any particular need for the STAA parameter. Such a need arises only for programs that have zero-delay cycles, where a federate sends an output at tag t and expects a response from other federates at the same tag t. It is extremely tricky to support such cycles. The centralized coordinator in Lingua Franca does support them [16], but when using the decentralized coordinator, some programs become impossible to support without compromises.

The banking application above appears to have such a zero-delay cycle, in that the web server reactors produce a `received` output when a user begins a deposit or withdrawal, and they expect a response with the same timestamp t. However, the web server application has no particular need to process these responses in timestamp order relative the outputs. If another output is sent with timestamp $t' > t$ before the response with timestamp t has been received, no logic errors will result as long as the responses are processed in the same order as the `received` outputs. Hence, the STA and STAA for the web server federates

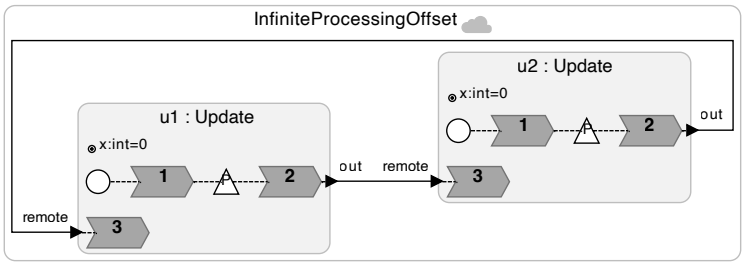

Fig. 7. A feedback system with infinite processing offsets.

can be set to zero, and the exception handler can simply process late responses identically to responses that are received on time. The pattern looks like this:

```
reaction(in) {=
  // Do something.
=} STAA(0) {=
  // Do the same something.
=}
```

Note that this pattern can also be implemented by just providing no STAA clause at all. But in such an implementation, LF will issue warnings stating that tag-order violations occurred and warning that there is no handler.

4.1 Impossible Consistency

However, not all applications are so forgiving. Consider a program with the pattern shown in Figure 7. Suppose that the two `Update` reactors maintain a replica of an integer quantity `x`. Each reactor can spontaneously update `x` at an arbitrary time using a Lingua Franca construct called a **physical action**, depicted in the diagram as a triangle with a "P" inside. The physical action asynchronously injects timestamped events when some external input is received, such as a message arrival. Suppose that reactions 2 and 3 in the `Update` reactors perform non-commutative updates to the value of `x`. Then, in order to maintain consistency, it becomes imperative to process the physical action events and `remote` input events in timestamp order in both federates. Only then can we assure that at any logical time t, the two `Update` reactors have the same value for `x`.

It is easy to see that there are no values for STA and STAA that make this program executable without timestamp-order violations. Suppose that the physical clocks have negligible clock synchronization error, and that communication between reactors takes 4 ms, at most. Then it might be tempting use an STA of, say, 5 ms in each reactor. But this will not work.

Suppose that the physical action in `u1` triggers at (elapsed) physical time 10 ms, according to the local physical clock. It assigns a logical time of 10 ms to the event. If the STA is 5 ms, then, in the absence of any signaling from u2, `u1`

will wait 5 ms (physical time) and then invoke reaction 2 at physical time 15 ms. Suppose now that u2's physical action had triggered at 9 ms. It too waits 5 ms, invoking its reaction 2 at physical time 14 ms. The message sent from u2 to u1 will arrive by physical time 18 ms. This will trigger a safe-to-process violation because the arriving message has timestamp 9 ms, but u1 has already advanced to logical time 10 ms and sent an output message with that timestamp.

It might be tempting to use an STA of zero and an STAA of 5 ms. But this too will not work. The same scenario as above will again trigger a timestamp-order violation.

This scenario is identified by Lee et al. [29] as one where communication network has a positive cycle mean, which results in the processing offsets diverging to infinity. In effect, each reactor seems to have to wait forever before advancing logical time.

4.2 A Solution

The centralized coordinator in LF is able to execute the program in Figure 7 without safe-to-process violations using the sophisticated techniques described by Donovan et al. [16]. We can use insights from that coordinator to augment the decentralized coordinator so that it too can execute this program. The key is that we can exploit some knowledge of the semantics of the physical action in Lingua Franca.

A physical action, when triggered, creates an event with a timestamp taken from the local physical clock. Suppose that we augment the decentralized coordination mechanism with a new kind of signal that is similar to the null messages used above, but with a critical difference. Each federate in Figure 7 could periodically send a null message *without advancing its logical time*. Now, each federate can use an STA of `forever` and the program will execute correctly with no safe to process violations, giving us a conservative strategy similar to that provided by the centralized coordinator, but without any centralized coordinator.

Consider three scenarios. First, assume that u1 has nothing to send for a long time. Suppose that every 5 ms, it sends a null message with logical times spaced by 5 ms, for example at (elapsed) logical times 0, 5 ms, 10 ms, etc. These messages are shown in blue in Figure 8. These null messages will flow in both directions, but to reduce clutter, we show only one direction until the symmetric case in Figure 8(c).

Suppose now that at physical time 12 ms, the physical action in u2 triggers, ① in Figure 8(a). Assume u2 has an STA of forever. Assume again perfectly synchronized clocks and a communication latency bound of 4 ms. By physical time 14 ms, u2 will receive a null message with timestamp 10 ms, ② in Figure 8(a), which is not sufficient to allow it to advance. By physical time 19 ms, however, u2 will receive a null message with timestamp 15 ms, ③ in Figure 8(a), which is sufficient. It can now commit and invoke reaction 2 at logical time 12 ms and physical time 19 ms. That reaction will send a message to u1, which will arrive at by physical time 23 ms and will have a lag of 11 ms, ④ in Figure 8(a), It is

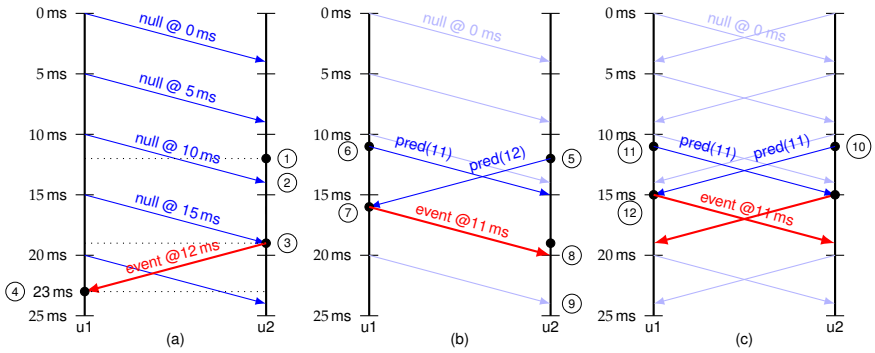

Fig. 8. Three scenarios where federates may send spontaneous messages to one another.

easy to see that in this case, the lag is bounded by 14 ms (twice the null message period plus the latency bound). Hence, with bounded latency, we achieve perfect consistency and bounded unavailability.

The second scenario we need to consider is where the two reactors' physical actions trigger close together in time. For example, as above, assume that at physical time 12 ms, the physical action in `u2` triggers, ⑤ in Figure 8(b), and at physical time 11 ms, the physical action in `u1` triggers, ⑥ in Figure 8(b). As above, `u1` will send a null message at 10 ms, but it cannot send one at 15 ms until it has processed the physical action. Similarly, `u2` will have sent a null message at 10 ms, but it too is blocked until it can process its event at 12 ms. How do we break the deadlock?

Simple! When a physical action triggers, at 11 ms in `u1` and 12 ms in `u2`, `u1` should send a null message with timestamp pred(11 ms) and u2 with timestamp pred(12 ms), where pred(t) is the largest timestamp less than t. This will break the deadlock. By physical time 16 ms, `u1` will receive `u2`'s null message with timestamp pred(12 ms), ⑦ in Figure 8(b). This timestamp is larger than its pending event timestamp of 11 ms, so it can safely advance its logical time to 11 ms and send a real message, shown in red in Figure 8(b). Federate `u2` will receive this message by physical time 20 ms, ⑧ in Figure 8(b), and it can immediately process the message because its own earliest pending message has timestamp 12 ms, which is greater than 11. To process its own pending event with timestamp 12, it will have to wait for another null message from `u1`, ⑨ in Figure 8(b), but the same mechanisms will work, and the lag will again be bounded.

The third scenario is where the two reactors' physical actions coincidentally trigger at exactly the same time, resulting in events with identical timestamps, say, 11 ms, ⑩ and ⑪ in Figure 8(c), Now, when `u1` receives `u2`'s null message with timestamp pred(11 ms), ⑫ in Figure 8(c), what should it do? It has a pending event with timestamp 11 ms, which is greater than pred(11 ms), so it seems it cannot safely advance. Here, fortunately, the semantics of Lingua Franca helps tremendously. First, note that `u1` *can* safely commit to advancing its logical time to 11 ms. It knows there will be no future input with timestamp *less than* 11 ms.

It *cannot*, however, safely assume anything about whether the input is present or absent *at* 11 ms.

First, once each federate has advanced its logical time to 11 ms, it can execute any reactions that do not depend on inputs to the federate and precede all reactions that do depend on an input. They can both, therefore, safely execute reaction 2 and send out their messages with timestamps 11 ms, red in Figure 8(c). But they must block before executing reaction 3 because they do not know the status of the input port at logical time 11 ms. They will become unblocked when they receive each others' messages.

In the decentralized coordinator, a federate with an STA of t_1 and an input port with an STAA of t_2 must wait until physical time $t + t_1 + t_2$ before it can assume that the input port is absent at logical time t. Because of this, we can set the STAA of the input port to 0. The STA is `forever`, so the federate will block indefinitely before executing reaction 3. Specifically, it will block until it receives the message with timestamp 11 ms from the other reactor, exactly the behavior we want!

The above strategy generalizes easily. It does not require perfectly synchronized clocks. Because the STA is `forever`, it delivers strong consistency. However, if latencies get large, this will come at a price in availability. As with the previous examples, we can compromise and choose a smaller STA and then provide fault handlers to deal with safe-to-process violations. How and whether to do that is entirely application dependent.

4.3 Using After Delays

A connection in Lingua Franca can have an `after` delay, as follows:

```
a.out -> b.in after 10 ms
```

This means that the timestamp at the receiving end will be 10 ms larger than the timestamp at the sending end. This feature can be used to realize a generalization of the logical execution time (LET) principle, as explained by Lee and Lohstroh [31]. This feature also relaxes consistency by a measured amount, allowing the source and destination to disagree on the value of a shared quantity for a bounded amount of logical time. As explained in Lee et al. [29], this can be used to improve availability.

It is tempting to assume that the subtleties we have dealt with occur only when a program demands strong consistency and features zero-delay cycles. However, the same problems arise in all programs whenever physical time latencies around a cycle exceed the logical time of the `after` delays. If we cannot enforce a strict bound on network latencies, then we inevitably have to deal with the possibility that this can occur. As I have shown, we can always choose either an approach emphasizing consistency, using STA of forever, or an approach emphasizing availability, using a finite STA and providing fault handlers. If we choose the approach emphasizing consistency, we can still provide fault handlers to deal with violations of our availability requirements using either Lingua Franca deadlines or watchdogs [9].

5 Related Work

Among the widely deployed variants that yield more deterministic behaviors are Kahn process networks [24], usually realized using the Kahn-MacQueen policy of blocking reads [25], and closely related dataflow models [32]. In the Kahn-MacQueen variant, for example, communication channels are assumed to have reliable in-order delivery of messages, and sequential processes block on channel reads until a message is available. Kahn and MacQueen showed that such processes realize prefix-monotonic functions on sequences of messages, and that a network of such processes defines a deterministic computation that is the least fixed point of a monotonic function constructed from the process functions.

Another family of widely deployed variants is based on discrete-event (DE) systems, which have historically been used for simulation [41,11], but can also be used as a deterministic execution model for actors [34]. In DE models, every message sent between actors has a tag, which is a value drawn from a totally-ordered set, and all messages are processed by actors in tag order. The model realized in Lingua Franca is closely related to these classic DE systems.

6 Conclusions

The nondeterministic ordering of message handling in the original actor model makes it difficult to achieve the consistency across a distributed system that some applications require. I have explored a number of mitigations, focusing primarily on the use of timestamps to define a semantic ordering and coordination mechanisms to ensure that messages are handled in timestamp order. A fundamental tradeoff (the CAL theorem) makes it impossible to achieve consistency without paying a price in availability, where the price depends on the latencies introduced by network communication, computation overhead, and clock synchronization error. Using the Lingua Franca coordination language, I have shown how to navigate this tradeoff, and particularly how to ensure eventual consistency while bounding unavailability with manageable risk.

Acknowledgments. The work in this paper was supported in part by the National Science Foundation (NSF), award #CNS-2233769 (Consistency vs. Availability in Cyber-Physical Systems), and the iCyPhy Research Center (Industrial Cyber-Physical Systems) at UC Berkeley.

Disclosure of Interests. The author has no competing interests to declare that are relevant to the content of this article.

References

1. Agha, G., Frolund, S., Kim, W., Panwar, R., Patterson, A., Sturman, D.: Abstraction and modularity mechanisms for concurrent computing. IEEE Parallel and Distributed Technology: Systems and Applications **1**(2), 3–14 (1993)

2. Agha, G.: ACTORS: A Model of Concurrent Computation in Distributed Systems. The MIT Press Series in Artificial Intelligence, MIT Press, Cambridge, MA (1986)
3. Agha, G.: Concurrent object-oriented programming. Communications of the ACM **33**(9), 125–140 (1990)
4. Agha, G.: Computing in pervasive cyberspace. Communications of the ACM **51**(1), 68–70 (2008). https://doi.org/10.1145/1327452.1327484
5. Agha, G.A.: Abstracting interaction patterns: A programming paradigm for open distributed systems. In: Najm, E., Stefani, J.B. (eds.) Formal Methods for Open Object-based Distributed Systems, IFIP Transactions. pp. 135–153. Chapman and Hall (1997). https://doi.org/10.1007/978-0-387-35082-0_10
6. Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming **7**(1), 1–72 (1997)
7. Alvaro, P., Conway, N., Hellerstein, J.M., Marczak, W.R.: Consistency analysis in bloom: a CALM and collected approach. In: 5th Biennial Conference on Innovative Data Systems Research (CIDR) (2011)
8. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent programming in Erlang. Prentice Hall, second edn. (1996)
9. Asch, B., Jellum, E., Lohstroh, M., Lee, E.A.: Software-defined watchdog timers for cyber-physical systems. Embedded Systems Letters (2024). https://doi.org/10.1109/LES.2024.3467332
10. Bateni, S., Lohstroh, M., Wong, H.S., Kim, H., Lin, S., Menard, C., Lee, E.A.: Risk and mitigation of nondeterminism in distributed cyber-physical systems. In: ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE) (2023). https://doi.org/10.1145/3610579.3613219
11. Cassandras, C.G.: Discrete Event Systems, Modeling and Performance Analysis. Irwin (1993)
12. Chandy, K.M., Misra, J.: Distributed simulation: A case study in design and verification of distributed programs. IEEE Trans. on Software Engineering **5**(5), 440–452 (1979)
13. Charousset, D., Hiesgen, R., Schmidt, T.C.: CAF - the C++ actor framework for scalable and resource-efficient applications. In: International Workshop on Programming based on Actors Agents and Decentralized Control (AGERE). pp. 15–28. ACM (2014). https://doi.org/10.1145/2687357.2687363
14. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's globally-distributed database. In: OSDI (2012). https://doi.org/10.1145/2491245
15. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's globally-distributed database. ACM Transactions on Computer Systems (TOCS) **31**(8) (2013). https://doi.org/10.1145/2491245
16. Donovan, P., Jellum, E., Jun, B., Kim, H., Lee, E.A., Lin, S., Lohstroh, M., Rengarajan, A.: Strongly-consistent distributed discrete-event systems. arXiv **2405.12117v1 [cs.DC]** (2024)
17. Dragojević, A., Narayanan, D., Nightingale, E.B., Renzelmann, M., Shamis, A., Badam, A., Castro, M.: No compromises: distributed transactions with consistency, availability, and performance. In: Symposium on Operating Systems Principles (SOSP). pp. 54–70 (2015). https://doi.org/10.1145/2815400.2815425

18. Eidson, J., Lee, E.A., Matic, S., Seshia, S.A., Zou, J.: Distributed real-time software for cyber-physical systems. Proceedings of the IEEE (special issue on CPS) **100**(1), 45–59 (2012). https://doi.org/10.1109/JPROC.2011.2161237
19. Eidson, J.C.: Measurement, Control, and Communication Using IEEE 1588. Springer (2006)
20. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Mateo, CA (1993)
21. Helland, P.: Don't get stuck in the "con" game: Consistency, convergence and confluence are not the same! Queue **19**(330), 16–35 (2021). https://doi.org/10.1145/3475965.3480470
22. Helland, P., Campbell, D.: Building on quicksand. In: Conference on Innovative Data Systems Research (CIDR). ACM (2009), https://arxiv.org/abs/0909.1788
23. Jefferson, D.: Virtual time. ACM Trans. Programming Languages and Systems **7**(3), 404–425 (1985)
24. Kahn, G.: The semantics of a simple language for parallel programming. In: Proc. of the IFIP Congress 74. pp. 471–475. North-Holland Publishing Co. (1974)
25. Kahn, G., MacQueen, D.B.: Coroutines and networks of parallel processes. In: Gilchrist, B. (ed.) Information Processing. pp. 993–998. North-Holland Publishing Co., Amsterdam (1977)
26. Kossmann, D.: The state of the art in distributed query processing. ACM Computing Surveys (CSUR) **32**(4), 422–469 (2000). https://doi.org/10.1145/371578.371598
27. Kuhl, F., Weatherly, R., Dahmann, J.: Creating Computer Simulation Systems: an Introduction to the High Level Architecture. Prentice Hall PTR (1999)
28. Laddad, S., Power, C., Milano, M., Cheung, A., Crooks, N., Hellerstein, J.M.: Keep CALM and CRDT on. arXiv **2210.12605v1 [cs.DB]** (2022). https://doi.org/10.48550/arXiv.2210.12605
29. Lee, E.A., Akella, R., Bateni, S., Lin, S., Lohstroh, M., Menard, C.: Consistency vs. availability in distributed cyber-physical systems. ACM Transactions on Embedded Computing Systems (TECS) **22**(5s), 1–24 (2023). https://doi.org/10.1145/3609119, presented at EMSOFT, September 17-22, 2023, Hamburg, Germany
30. Lee, E.A., Bateni, S., Lin, S., Lohstroh, M., Menard, C.: Trading off consistency and availability in tiered heterogeneous distributed systems. Intelligent Computing **2**(Article 0013), 1–23 (2023). https://doi.org/10.34133/icomputing.0013
31. Lee, E.A., Lohstroh, M.: Generalizing logical execution time. In: Raskin, J.F., Chatterjee, K. (eds.) Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday. vol. LNCS 13660, pp. 1–22. Springer Nature (2022). https://doi.org/10.1007/978-3-031-22337-2_8
32. Lee, E.A., Matsikoudis, E.: The Semantics of Dataflow with Firing. Cambridge University Press (2009), http://ptolemy.eecs.berkeley.edu/publications/papers/08/DataflowWithFiring/
33. Lohstroh, M., Lee, E.A., Edwards, S., Broman, D.: Logical time for reactive software. In: Workshop on Timing-Centric Reactive Software (TCRS), in Cyber-Physical Systems and Internet of Things Week (CPSIoT). ACM (2023). https://doi.org/10.1145/3576914.3587494
34. Lohstroh, M., Menard, C., Bateni, S., Lee, E.A.: Toward a lingua franca for deterministic concurrent systems. ACM Transactions on Embedded Computing Systems (TECS) **20**(4), Article 36 (2021). https://doi.org/10.1145/3448128
35. Mills, D.L.: Computer Network Time Synchronization — The Network Time Protocol. CRC Press, Boca Raton, FL (2006)
36. Muller, R.A.: Now — The Physics of Time. W. W. Norton and Company (2016)

37. Ptolemaeus, C.: System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org, Berkeley, CA (2014), http://ptolemy.org/books/Systems
38. Roestenburg, R., Bakker, R., Williams, R.: Akka in Action. Manning Publications, 2nd edn. (2023)
39. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. Report, INRIA (2011), https://pages.lip6.fr/Marc.Shapiro/papers/RR-7687.pdf
40. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundamenta Informaticae **63**(4), 385–410 (2004). https://doi.org/10.3233/FUN-2004-63405
41. Zeigler, B.: Theory of Modeling and Simulation. Wiley Interscience, New York (1976)
42. Zeigler, B.P., Lee, J.S.: Theory of quantized systems: Formal basis for DEVS/HLA distributed simulation environment. In: SPIE Conference on Enabling Technology for Simulation Science. vol. SPIE Vol. 3369, pp. 49–58 (1998). https://doi.org/10.1117/12.319354
43. Zhao, Y., Lee, E.A., Liu, J.: A programming model for time-synchronized distributed real-time systems. In: Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 259 – 268. IEEE (2007). https://doi.org/10.1109/RTAS.2007.5