Intelligent Computing
A SCIENCE PARTNER JOURNAL

RESEARCH ARTICLE

Trading Off Consistency and Availability in Tiered Heterogeneous Distributed Systems

Edward A. Lee[1*], Soroush Bateni[1], Shaokai Lin[1], Marten Lohstroh[1], and Christian Menard[2]

[1]UC Berkeley, Berkeley, CA, USA. [2]TU Dresden, Dresden, Germany.

*Address correspondence to: eal@berkeley.edu

Citation: Lee EA, Bateni S, Lin S, Lohstroh M, Menard C. Trading Off Consistency and Availability in Tiered Heterogeneous Distributed Systems. *Intell. Comput.* 2023;2:Article 0013. https://doi.org/10.34133/icomputing.0013

Submitted 2 November 2022
Accepted 20 December 2022
Published 15 February 2023

Tiered distributed computing systems, where components run in Internet-of-things devices, in edge computers, and in the cloud, introduce unique difficulties in maintaining consistency of shared data while ensuring availability. A major source of difficulty is the highly variable network latencies that applications must deal with. It is well known in distributed computing that when network latencies rise sufficiently, one or both of consistency and availability must be sacrificed. This paper quantifies consistency and availability and gives an algebraic relationship between these quantities and network latencies. The algebraic relation is linear in a max-plus algebra and supports heterogeneous networks, where the communication latency between 2 components may differ from the latency between another 2 components. We show how to make use this algebraic relation to guide design, enabling software designers to specify consistency and availability requirements, and to derive from those the requirements on network latencies. We show how to design systems to fail in predictable ways when the network latency requirements are violated, by choosing to sacrifice either consistency or availability.

Introduction

Brewer's well-known CAP theorem states that in the presence of network partitioning (P), a distributed system must sacrifice at least one of availability (A) or consistency (C) [1–3]. Consistency is where distributed components agree on the value of shared state, and availability is the ability to respond to user requests using and/or modifying that shared state. All networks have latency, and "network partitioning" is just an extreme case, where latency becomes unbounded.

We have recently discovered that consistency, availability, and network latency can all be quantified, and that there is a simple algebraic relationship between them [4]. We call this relationship the Consistency, Availability, apparent Latency (CAL) theorem, replacing "Partitioning" with "Latency". The relation is a linear system of equations in a max-plus algebra, where the structure of the equations reflects the communication topology of an application. In particular, the matrix form of the equations enables compact modeling of heterogeneous networks, where the latencies between pairs of nodes can vary considerably.

It is becoming increasingly common to design systems to operate in tiered, heterogeneous networks. Internet-of-things and embedded devices, such as factory robots, medical devices, autonomous vehicle controllers, and security systems, may connect to edge computers over wired or wireless links. Those edge computers, in turn, may connect to cloud-based services that enable wide area aggregation and scalability, for example, for machine learning. The various networks involved may have widely varying characteristics, yielding enormously different latencies and latency variability. A time-sensitive network (TSN) [5] on a factory floor, for example, may yield reliable latencies on the scale of microseconds between edge computers, whereas wide-area networks (WANs) may yield highly variable latencies that can extend up to tens of seconds [6]. Moreover, any of these networks can fail, yielding unbounded latencies, and systems need to be designed to handle such failures gracefully.

The CAL theorem will allow us to model a heterogeneous network topology interconnecting a wide variety of nodes. We will use the Lingua Franca (LF) coordination language [7] to specify programs that explicitly define availability and consistency requirements for a distributed application. We can then use the CAL theorem to derive the network latency bounds that make meeting the requirements possible. This can be used to guide decisions about which services must be placed in the end devices, which can be placed on an edge computer, and which can be put in the cloud. Moreover, we will show how, once such a system is deployed, violations of the network latency requirements, which will make it impossible to meet the consistency and availability requirements, can be detected. System designers can build in to the application fault handlers that handle such failures gracefully.

The relationships between consistency, availability, and latency can be used to derive networking requirements from availability and/or consistency requirements, or to derive availability and/or consistency properties from assumptions about the network. In practice, any set of networking requirements or assumptions about network latencies may be violated in the field. We will show how systems can be designed to fail in predictable ways when this occurs.

The CAP theorem itself is rather obvious and very much part of the folklore in distributed computing. By quantifying

it and relating it to individual point-to-point latencies, the CAL theorem elevates the phenomenon from folklore to an engineering principle, enabling rigorous design with clearly stated assumptions. In this paper, we show how to carry out such rigorous design using the LF coordination language [7], which supports explicit representations of availability and consistency requirements. Moreover, we show how to detect situations where the networking requirements that are implied by the availability and consistency requirements cannot be met, for example when the network fails. We show how LF can provide exception handlers that enable the designer to explicitly choose how to handle such fault conditions, for example, by reducing accuracy [6] or by switching to fail-safe modes of operation.

Motivating application

Kuhn [8] gives a simple application that maintains bank balances in a distributed database and accepts deposits and withdrawals at distributed locations like automatic teller machines (ATMs). Suppose that a customer shows up and wants to withdraw w dollars. This needs to be compared against the current balance x in the account to prevent an overdraft. However, there may be near-simultaneous withdrawals occurring at other locations. A conservative approach would have each ATM access a centralized database before dispensing cash. However, the CAP theorem tells us that a consequence of this design choice is that the ATM will become unavailable in the event of a network failure. The CAL theorem enables us to calculate the loss of availability as a function of network latencies. In practice, network latencies over a WAN can get into tens of seconds [6], which may still be acceptable for the ATM application. But it it not hard to imagine applications with more stringent timing requirements, such as control of autonomous vehicles that share state through a centralized database. We will use Kuhn's example because of its simplicity and understandability, but please understand that the principles apply over a wide range of timing requirements.

If the CAL theorem yields unacceptable availability for realistic network latencies, then a design alternative could use edge computers, e.g., at bank branch offices, with replicas of the database that can respond with much lower network latencies. But then consistency becomes a major issue. If multiple deposits and withdrawals can occur on the same account at different locations around the world, then a strong form of consistency requires agreement on the order in which these deposits and withdrawals occur. If all locations agree on this order, the bank can assure that the balance never drops below zero by denying any withdrawal that would make it so. The CAL theorem will reveal that, in such an architecture, availability will still depend on WAN latencies and may still yield unacceptable availability.

Here, system designers face a business choice. They can relax consistency, thereby improving availability at the cost of some business risk, or they can enforce strong consistency, risking irate customers when the network underperforms. We will show how to build programs that explicitly relax consistency by a measured amount, and we will use the CAL theorem to show how this improves availability for a given profile on network latencies.

Relaxing consistency by a measured amount means that inconsistencies are temporary. If such a relaxation of bank policy allows the balance to drop below zero, all branches will eventually agree on the number of times that this has occurred, so they will all agree on what overdraft charges to apply.

The programs we give can also explicitly specify availability requirements. For example, the program may give the ATM computer a deadline of 500 ms to respond to any user request. With the consistency requirements also specified in the program, the CAL theorem can be used to derive the point-to-point network latency bounds beyond which maintaining the specified availability and consistency becomes impossible. When network latency gets larger than these bounds, the program requirements can no longer be met. We describe a distributed coordinator that detects such violations and enables the system designer to handle them as faults. How to handle such faults is again a business decision, so it is important to use a software framework that enables detection and handling of such faults. By enabling explicit design choices, the programmer can specify whether an ATM will deny dispensing cash because of a temporary network failure. How big is the risk to the bank if the ATM dispenses cash on the basis of possibly inconsistent data? The trade-off between risk and customer service is a business decision.

Kuhn's application admits a whole range of design choices, ranging from strong consistency to highest availability. Moreover, the technique generalizes to many kinds of applications. For example, the merge operations that combine updates to a shared variable are associative and commutative if overdrafts are allowed, and not otherwise. Our technique accommodates merge operations that are associative, commutative, both, or neither. Hence, this example enables exploration with a variety of patterns.

Related work

Gilbert and Lynch [9] proved 2 variants of the CAP theorem, one strong result for asynchronous networks [10, chapter 8] and one weaker result for partially synchronous networks. Abadi [11] argues that CAP is irrelevant when there are no network partitions, but Brewer [2] points out that network partitions are not a binary property; all networks have latency, and a complete communication failure is just the limiting case when the latency becomes unbounded.

The only prior attempt we are aware of to quantify the CAP theorem was done by Yu and Vahdat [12], who quantified availability and consistency and show a trade-off between them. Their quantifications, however, are in terms of fractions of satisfied accesses (availability) and fractions of out-of-order writes (inconsistency), and they show that finding the availability as a function of consistency is NP-hard. In contrast, the CAL theorem defines these quantities as time intervals and gives a strikingly simpler relationship, one that is linear in a max-plus algebra.

Organization of the paper

In the "Materials and Methods" section, to make this paper self contained, we review the role of time and the use of timestamps; formally define inconsistency, unavailability, and network latency; and give the CAL theorem, which relates these three quantities [4]. In the "Trading off consistency and availability in practice" section, we explain how a system designer can use the CAL theorem to guide system design, give complete programs illustrating the tradeoffs implied by the CAL theorem, and show how the LF coordination language supports being explicit about these tradeoffs. In the "Results and Discussion" section, we describe two implementations: one that bounds inconsistency and one that bounds unavailability in the face of network partitions. The table above provides a summary of notation and acronyms used in the paper.

Table. Notation and acronyms used in this paper.

$\overline{A}_i$	Unavailability at node i	ATM	Automatic teller machine
$\overline{C}_{ij}$	Inconsistency from j to i	CAL	Consistency, Availability, apparent Latency
D	Logical delay	CAP	Consistency, Availability, Partitioning tolerance
E_{ij}	Clock sync error from j to i	HLA	High-level architecture
g	Tag	LF	Lingua Franca
$\mathbb{G}$	Set of tags	NTP	Network Time Protocol
Γ	Topology matrix	PTAG	Provisional tag advance grant
$\mathbb{I}$	Set of time intervals	PTP	Precision Time Protocol
L_{ij}	Network latency from j to i	RTI	Run-time infrastructure
$\mathcal{L}_{ij}$	Apparent latency from j to i	STA	Safe to advance
$\overline{O}_i$	Processing offset at i	STAA	Safe to assume absent
t	Logical time	TAG	Tag advance grant
T	Physical time	TAN	Time advance notice
$\mathbb{T}$	Set of times	TSN	Time-sensitive network
$\mathcal{T}$	Map from tag to time	WAN	Wide-area network
X_{ij}	Execution time from j to i		

Materials and Methods

Logical and physical times

We use two distinct notions of time, logical and physical. A physical time $T \in \mathbb{T}$ is an imperfect measurement of time taken from some clock somewhere the system. The set $\mathbb{T}$ contains all the possible times that a physical clock can report. We assume that $\mathbb{T}$ is totally ordered and includes two special members: $\infty, -\infty \in \mathbb{T}$, larger and smaller than any time any clock can report. We will occasionally make a distinction between the set $\mathbb{I}$ of time intervals (differences between two times) and time values $T \in \mathbb{T}$. It is often convenient to have the set $\mathbb{T}$ represent a common definition of physical time, such as Coordinated Universal Time (UTC) because, otherwise, comparisons between times will not correlate with physical reality. (In LF, $\mathbb{T}$ and $\mathbb{I}$ are both the set of 64-bit integers. Following the Portable Operating System Interface (POSIX) standard, $T \in \mathbb{T}$ is the number of nanoseconds that have elapsed since midnight, 1970 January 1, Greenwich mean time. The largest and smallest 64-bit integers represent ∞ and $-\infty$, respectively. As a practical matter, these numbers will overflow in systems running near the year 2270.)

For logical time, we use an element that we call a tag g of a totally ordered set $\mathbb{G}$. Each event in a distributed system is associated with a tag $g \in \mathbb{G}$. From the perspective of any component of a distributed system, the order in which events occur is defined by the order of their tags. If two distinct events have the same tag, we say that they are logically simultaneous. We assume that the tag set $\mathbb{G}$ also has largest and smallest elements. (In LF, $\mathbb{G} = \mathbb{T} \times \mathbb{U}$, where $\mathbb{U}$ is the set of 32-bit unsigned integers representing the microstep of a superdense time system [13–15]. Following the tagged-signal model [16], we use the term tag rather than timestamp to allow for such a richer model of logical time. For the purposes of this paper, however, the microsteps will not matter, and hence, you can think of a tag as a timestamp and ignore the microstep.) We will consistently denote tags with a lower case $g \in \mathbb{G}$ and measurements of physical time $T \in \mathbb{T}$ with upper case.

To combine tags with physical times, we assume a monotonically nondecreasing function $\mathcal{T}: \mathbb{G} \rightarrow \mathbb{T}$ that gives a physical time interpretation to any tag. For any tag g, we call $\mathcal{T}(g)$ its timestamp. (In LF, for any tag $g = (t,m) \in \mathbb{G}, \mathcal{T}(g) = t$. Hence, to get a timestamp from a tag, you just have to ignore the microstep.) The set $\mathbb{G}$ also includes largest and smallest elements such that $\mathcal{T}(\infty_{\mathbb{G}}) = \infty_{\mathbb{T}}$ and $\mathcal{T}(-\infty_{\mathbb{G}}) = -\infty_{\mathbb{T}}$, where the subscripts disambiguate the infinities.

An external input, such as a user attempting a withdrawal at an ATM, will be assigned a tag g such that $\mathcal{T}(g) = T$, where T is a measurement of physical time taken from the local clock where the input first enters the system. (In LF, this tag is normally given microstep 0, $g = (T, 0)$.) For these tags to be meaningful globally, some effort must be put into clock synchronization. The extent to which clocks must be synchronized is also application dependent. We will show that clock synchronization error is indistinguishable from network latency, and hence the CAL theorem can also provide guidance on the extent to which clocks must be synchronized.

The CAL theorem

Following Schwartz and Mattern [17], assume we are given a trace of an execution of a distributed system consisting of N sequential processes, where each process is an unbounded sequence of (tagged) events. Although the theory is developed for traces, the CAL theorem can be used for programs, not just traces because a program is formally a family of traces. The k-th event of a process is associated with a tag g_k and a physical time T_k. The physical time T_k is the reading on a local clock at the time where the event starts being processed. The events in a process are required to have nondecreasing tags and increasing physical times. That is, if g_k is the tag and T_k is the physical time of the k-th event, then $g_k \leq g_{k+1}$ and $T_k < T_{k+1}$.

For exposition, we focus on events that read and update a single shared variable x, such as the balance of a bank account.

Within each process, a read event with tag g yields the value of a shared variable x. The shared variable x is stored as a local copy, which has previously acquired a value via a a write event or an accept event in the same process. An accept event receives an updated value of the variable from the network. A read event with tag g will yield the value assigned by the write or accept event with the largest tag g' where $g' \leq g$. If $g' = g$, then we require that $T' < T$, where T' is the physical time of the write or accept event and T is the physical time of the read event. This requirement ensures that a read event reads a value that was written at an earlier physical time.

A send event is where a process launches into the network an update to a shared variable x, for example the amount of a deposit or withdrawal. Like a read event, the send event has a tag greater than or equal to that of the write or accept event that it is reporting and a physical time greater than that of the write or accept event. An accept event that receives the update sent by the send event has a tag greater than or equal to that of the send event. The physical time of the accept event relative to the originating send event is unconstrained, however, because these times likely come from distinct physical clocks.

Definition 1. *Inconsistency*. For each write event on process j with tag g_j, let g_i be the tag of the corresponding accept event on process i or ∞ if there is no corresponding accept event. The inconsistency $\overline{C}_{ij} \in \mathbb{I}$ from j to i is defined to be

$$\overline{C}_{ij} = \max(\mathcal{T}(g_i) - \mathcal{T}(g_j)), \tag{1}$$

where the maximization is over all write events on process j. If there are no write events on j, then we define $\overline{C}_{ij} = 0$.

It is clear that $\overline{C}_{ij} \geq 0$. If $\overline{C}_{ij} = 0$ for all i and j, we have strong consistency. We will see that this strong consistency comes at a price in availability; in particular, network failures can result in unbounded unavailability. If $\overline{C}_{ij}$ is bounded, then we have eventual consistency, and the bound quantifies "eventual".

Unavailability, $\overline{A}$, is a measure of the time it takes for a system to respond to user requests [18]. A user request is any external event that originates from outside the distributed system that requires reading the shared variable to provide a response. Assume that a user request triggers a read event in process i with tag g_i such that its timestamp $\mathcal{T}(g_i)$ is the reading of a local clock when the external event occurs. Let T_i be the physical time of the read event, i.e., the physical time at which the read is processed. Hence, $T_i \geq \mathcal{T}(g_i)$.

Definition 2. *Unavailability*. For each read event on process i, let g_i be its tag and T_i be the physical time at which it is processed. The unavailability $\overline{A}_i \in \mathbb{I}$ at process i is defined to be

$$\overline{A}_i = \max\big(T_i - \mathcal{T}(g_i)\big), \tag{2}$$

where the maximization is over all read events on process i that are triggered by user requests. If there are no such read events on process i, then $\overline{A}_i = 0$.

Because we are considering only read events that are triggered from outside the software system, $\mathcal{T}(g_i) \leq T_i$, so $\overline{A}_i \geq 0$. If $\overline{A}_i = 0$, then we have maximum availability (minimum unavailability). This situation arises when external triggers cause immediate reactions.

We require that each process handle events in tag order. This gives the overall program a formal property known as causal consistency, which is analyzed in depth by Schwartz and Mattern. They define a causality relation, written $e_1 \rightarrow e_2$, between events e_1 and e_2 to mean that e_1 can causally affect e_2. The phrase "causally affect" is rather difficult to pin down (see Lee [19, Chapter 11] for the subtleties around the notion of causation), but, intuitively, $e_1 \rightarrow e_2$ means e_2 cannot behave as if e_1 had not occurred. Put another way, if the effect of an event is reflected in the state of a local replica of a variable x, then any cause of the event must also be reflected. Put yet another way, an observer must never observe an effect before its cause.

Formally, the causality relation of Schwartz and Mattern is the smallest transitive relation such that $e_1 \rightarrow e_2$ if e_1 precedes e_2 in a process, or e_1 is the sending of a value in one process (event type s_x) and e_2 is the acceptance of the value in another process (event type a_x). If neither $e_1 \rightarrow e_2$ nor $e_2 \rightarrow e_1$ holds, then we write $e_1 \| e_2$ or $e_2 \| e_1$ and say that e_1 and e_2 are incomparable. The causality relation is identical to the "happened before" relation of Lamport et al. [20], but Schwartz and Mattern prefer the term "causality relation" because even if e_1 occurs unambiguously earlier than e_2 in physical time, they may nevertheless be incomparable, $e_1 \| e_2$.

The causality relation is a strict partial order. Schwartz and Mattern use their causality relation to define a "consistent global snapshot" of a distributed computation to be a subset S of all the events E in the execution that is a downset, meaning that if $e' \in S$ and $e \rightarrow e'$, then $e \in S$ (this was previously called a "consistent cut" by Mattern [21]).

To maintain causal consistency, it is sufficient that each process handle events in order of nondecreasing tags. For this reason, in a trace, a read or write event triggered by an external input may have a physical time T that is significantly larger than its tag's timestamp $\mathcal{T}(g)$. While $\mathcal{T}(g)$ is determined by the physical clock at the time the external input appears, the physical time at which the event is actually processed may have to be later to ensure that all events with earlier tags have been processed. This motivates the following definition:

Definition 3. *Processing Offset*. For process i, the processing offset $O_i \in \mathbb{I}$ is

$$O_i = \max\big(T_i - \mathcal{T}(g_i)\big) \tag{3}$$

where T_i and g_i are the physical time and tag, respectively, of a write event on process i that is triggered by a local external input (and hence assigned a timestamp drawn from the local clock). The maximization is over all such write events in process i. If there are no such write events, then $O_i = 0$.

The processing offset closely resembles the unavailability of Definition 2, but the former refers to write events and the latter refers to read events. The processing offset, by definition, is greater than or equal to zero.

When a write to a shared variable occurs in process j, some time will elapse before a corresponding accept event on process i triggers a write to its local copy of the shared variable. This motivates the following definition:

Definition 4. *Apparent Latency*. Let g_j be the tag of a write event in process j that is triggered by an external input at j

(so $\mathcal{T}(g_j)$ is the physical time of that external input). Let T_i be the physical time of the corresponding accept event in process i (or ∞ if there is no such event). (If $i = j$, we assume T_i is the same as the physical time of the write event.) The apparent latency or just latency $\mathcal{L}_{ij} \in \mathbb{I}$ for communication from j to i is

$$\mathcal{L}_{ij} = \max(T_i - \mathcal{T}(g_j)), \tag{4}$$

where maximization is over all such write events in process j. If there are no such write events, then $\mathcal{L}_{ij} = 0$.

Note that T_i and $\mathcal{T}(g_j)$ are physical times on two different clocks if $i \neq j$, so this apparent latency is an actual latency only if those clocks are perfectly synchronized. Unless the two processes are actually using the same physical clock, they will never be perfectly synchronized. Hence, the apparent latency may even be negative. Note that despite these numbers coming from different clocks, if tags are sent along with messages, this apparent latency is measurable.

The apparent latency is a sum of four components,

$$\mathcal{L}_{ij} = O_j + X_{ij} + L_{ij} + E_{ij}, \tag{5}$$

where X_{ij} is execution time overhead at node j for sending a message to node i, L_{ij} is the network latency from j to i, and E_{ij} is the clock synchronization error. The three latter quantities are indistinguishable and always appear summed together, so there is no point in breaking apparent latency down in this way. Moreover, these latter three quantities would have to be measured with some physical clock, and it is not clear what clock to use. The apparent latency requires no problematic measurement since it explicitly refers to local clocks and tags.

The clock synchronization error can be positive or negative, whereas O_j, X_{ij}, and L_{ij} are always nonnegative. If E_{ij} is a sufficiently large negative number, the apparent latency will itself also be negative. Because of the use of local clocks, the accept event will appear to have occurred before the user input that triggered it. This possibility is unavoidable with imperfect clocks.

We have derived the overall unavailability for node i from the above definitions [4], yielding the following theorem:

Theorem 1. Given a trace, the unavailability at process i is, in the worst case,

$$\overline{A}_i = \max\left(O_i, \max_{j \in N}\left(\mathcal{L}_{ij} - \overline{C}_{ij}\right)\right), \tag{6}$$

where O_i is the processing offset, $\mathcal{L}_{ij}$ is apparent latency (which includes O_j), and $\overline{C}_{ij}$ is the inconsistency.

The proof of this theorem follows immediately from the carefully constructed definitions.

This theorem can be put in an elegant form using max-plus algebra [22]. Let N be the number of processes, and define an $N \times N$ matrix Γ such that its elements are given by

$$\Gamma_{ij} = \mathcal{L}_{ij} - \overline{C}_{ij} - O_j = X_{ij} + L_{ij} + E_{ij} - \overline{C}_{ij}. \tag{7}$$

That is, from Eq. 5, the i, j-th entry in the matrix is an assumed bound on $X_{ij} + L_{ij} + E_{ij}$ (execution time, network latency, and clock synchronization error), adjusted downwards by the specified tolerance for inconsistency $\overline{C}_{ij}$ on the communication from node j to node i.

Let A be a column vector with elements equal to the unavailabilities $\overline{A}_i$ and O be a column vector with elements equal to the processing offsets O_i. Then, the CAL theorem Eq. 6 can be written as

$$A = O \oplus \Gamma O, \tag{8}$$

where the matrix multiplication is in the max-plus algebra and $\oplus$ is addition in the max-plus algebra. This can be rewritten as

$$A = (I \oplus \Gamma)O, \tag{9}$$

where I is the identity matrix in max-plus, which has zeros along the diagonal and $-\infty$ everywhere else. Hence, unavailability is a simple linear function of the processing offsets, where the function is given by a matrix that depends on the network latencies, clock synchronization error, execution times, and specified inconsistency in a simple way.

The matrix Γ encodes pairwise latencies between each pair of nodes in a system adjusted by a specified tolerance for inconsistency. These latencies include not just network latencies but also execution times that affect availability and clock synchronization error, which is indistinguishable from network latency. Because it encodes these latencies pairwise, the max-plus formulation Eq. 9 compactly accounts for tiered, heterogeneous networks. It can combine lower latencies for communication with edge devices, higher latencies for communication over a WAN, and heterogeneous mixtures of clock synchronization technologies, such as NTP (Network Time Protocol) [23,24] on cloud services and PTP (Precision Time Protocol) [25,26] on edge devices. It can also accommodate heterogeneous networking technologies, such as wireless links to mobile devices, TSNs (Time Sensitive Networks) [5] linking edge devices, and the open internet connecting cloud devices.

Terminology

Some caution is in order when relating our work to prior interpretations of the CAP theorem. Kleppmann [18] critiques this prior work for loose definitions of consistency, availability, and partitioning. He points out that availability is loosely defined as the "proportion of time during which a service is able to successfully handle requests, or the proportion of requests that receive a successful response". He then critiques this definition, pointing out that "it is nonsensical to say that some software package or algorithm is 'available' or 'unavailable'" and suggests replacing availability with a measure of the time it takes to respond to user requests, which he calls "latency". We have adopted that suggestion here, but we reserve the word "latency" for our quantitative measure of delays due to networking and execution time. Hence, agreeing with Kleppmann's critique, our "unavailability" is not a proportion of time the system is down but rather is the time it takes to produce a response to a user. When a user request fails altogether, we define the unavailability to be infinite. Similarly, when the network fails altogether (partitioning), the "latency" in CAL becomes infinite.

Trading off consistency and availability in practice

LF [7] is a polyglot coordination language that orchestrates concurrent and distributed programs written in any of several target languages (as of this writing, C, C++, Python, TypeScript, and Rust). LF supports a full range of explicit tradeoffs between availability and consistency for a wide variety of applications. In this section, we give a small collection of complete LF programs that illustrate these trade-offs using Kuhn's ATM example [8].

In LF, asynchronous external inputs to the software system manifest as events that are assigned tags based on a local clock that measures physical time. These events (and any further events triggered by them) will be processed in tag order by software components called reactors [27,28]. At every tag, the dependencies between the reactions of each reactor are known to the runtime system, and hence simultaneous events are handled deterministically, and causal consistency is ensured.

For an LF program that is distributed across a network (called a federated program), the LF runtime system provides a default clock synchronization mechanism to ensure that tags assigned at different nodes are at least approximately based on a common physical time line. Once the tags are assigned, however, they lie on a logical time line that is common across the entire system.

If the business decision is that consistency has priority over availability, then the LF program is simple, as we will see (we develop variants of the program in this section). Each node, representing an ATM, receives user input representing a deposit or withdrawal, assigns it a tag, and broadcasts it to all other nodes. It also accepts as inputs such broadcasts from all other nodes. The LF program is arranged so that before the node responds to a user input, it processes all updates from other nodes that have tags less than or equal to the tag assigned to its own user input. We can then use the CAL theorem to derive the resulting unavailability, which is a bound on the time it takes to respond to the user. The result is satisfyingly intuitive. For this particular communication pattern (all-to-all broadcast), the unavailability at any node is the maximum apparent latency from other nodes, or zero if all those apparent latencies are negative.

If the business decision is that availability is more important than consistency, then a small change to the LF program yields an alternative design. In LF, a programmer can manipulate the tags of events, adding a logical delay to a message sent from one node to another. If the programmer adds a logical delay D to each broadcast, then the CAL theorem tells us that the unavailability for this program structure becomes

$$\overline{A}_i = \max\left(0, \max_{j \in N}\left(\mathcal{L}_{ij} - D\right)\right), \tag{10}$$

at the i-th node. If D is larger than all apparent latencies, then the unavailability is minimized to zero.

An intermediate design choice may be preferred. For example, a system designer may start from a requirement that unavailability not exceed, say, 100 ms, and that inconsistency not exceed, say 1 s. From this, the CAL theorem gives the designer a bound on apparent latency. Setting $D = 1$ s and $\overline{A}_i = 100$ ms in Eq. 10, we derive the requirement that

$$\mathcal{L}_{ij} \leq 1.1 \text{ s}.$$

From Eq. 5, we see that apparent latency includes four components. The processing offsets for this application are all zero, $O_i = 0$, as we will see in the "Results and Dicussion" section, so the remaining terms, network latency, clock synchronization error, and execution time overhead remain. We now have a formal requirement on these quantities that can guide the choice of a clock synchronization technology (the LF default will probably be sufficient in this case) and service-level agreements with network providers.

Once we have such a design, if our (now explicit) assumptions about network behavior are met in the field, then the behavior of the system will be exactly as defined. There is always the possibility, however, that these assumptions will be violated. In the "Results and Dicsussion" section, we describe two distinct coordination mechanisms that we have built on top of LF, one of which maintains consistency and the other of which maintains availability when the assumptions are violated. The CAL theorem tells that, in the presence of such violations, one of the two requirements must be sacrificed. Which one to sacrifice is again a business decision.

More complicated program structures are easily supported by our formalism. Kuhn's banking example has a simple all-to-all broadcast structure, but more interesting programs may have a much less uniform communication structure. That structure gets encoded in the Γ matrix, but, otherwise, the mathematical formulation remains the same. A system designer can even apply nonuniform trade-offs, emphasizing availability for some particular service while emphasizing consistency for another.

In the next subsection, we give a brief overview of LF, followed by complete programs realizing Kuhn's ATM example [8]. (Download these programs from https://cal.lf-lang.org/.)

Brief introduction to LF

LF is a coordination language where applications are defined as concurrent compositions of components called reactors [27,28]. (The LF compiler and documentation can be found at https://lf-lang.org. The source code for the compiler is available at https://repo.lf-lang.org.) Figure 1 outlines the structure of an LF program. One or more reactor classes are defined with input ports (line 3), output ports (line 5), state variables (line 7), and timers and actions. We will not need timers here and will elaborate on actions later. If a reactor class is instantiated within a federation, as shown on line 18, then the instance is called a federate, and tagged inputs will arrive from the network at the input ports and be handled in tag order. Inputs are handled by reactions, as shown on line 11. Reactions declare their triggers, as on line 11, which can be input ports, timers, or actions. If a reaction lists an output port among its effects, then it can produce tagged output messages via that output port. The routing

```
1  target L;
2  reactor ReactorClass {
3      input name:type;
4      ...
5      output name:type;
6      ...
7      state name:type(init);
8      ...
9      ... timers, actions, if any ...
10     ...
11     reaction(trigger, ...) -> effect, ... {=
12         ... code in language L ...
13     =}
14     ... more reactions ...
15 }
16 ...
17 federated reactor {
18     instance = new ReactorClass();
19     ...
20     instance.name -> instance.name;
21     ...
22 }
```

Fig. 1. Structure of a federated LF program for target language L.

of messages is specified by connections, as shown on line 20. The syntax and semantics will become clearer as we develop our specific applications.

Commutative and associative replica

We begin with a version that has an associative and commutative merge operation. The first part of this version is shown in Fig. 2. This defines a reactor class. The first line defines the target language, which is the language of the program that the LF code generator will produce and the language in which the business logic of the software component is written. To minimize dependencies, we give our examples here with C as the target language.

The second line declares a new reactor class called CAReplica ("CA" for Commutative and Associative). This reactor has three inputs, defined on lines 3 through 5. The first input accepts a local update, an integer that is positive for a deposit and negative for a withdrawal. The second input accepts a remote update, which will come from some other machine somewhere on the network (we will generalize this later to accept an arbitrary number of remote updates). The third input accepts a query for the current balance.

Line 7 defines a local state variable, an integer that is the local copy of the balance. Line 9 defines an integer output, which will be a response to a query for the balance. In a strongly consistent design (which we will see how to construct), this balance will be agreed upon by all replicas at each tag.

Lines 11 through 18 give the business logic of how to handle local or remote updates. The code between the delimiters {= … =} is ordinary C code making use of mechanisms provided by the LF code generator to access the inputs and state. This code simply checks to see which inputs are present and adjusts the balance accordingly. Because the operation is commutative and associative, it does not matter in which order these inputs are handled.

The figure to the right is automatically generated by the LF tools. (The diagram synthesis feature was created by Alexander Schulz-Rosengarten of Kiel University using the graphical layout tools from the KIELER Lightweight Diagrams framework [29].) The chevrons in the figure represent reactions, and their dependencies on inputs and their ability to produce outputs are shown using dashed lines.

User

The code in Fig. 3 defines an LF reactor that stands in for an ATM through which a customer can make deposits or withdrawals. This component listens for the user to type a number on a terminal and then produces that number on its output port. To keep things simple, there is no authentication and not much error checking. While obviously critical to a real ATM design, those aspects are irrelevant to our discussion here, so we leave them out.

Upon startup, on line 33, this reactor creates a thread that executes concurrently with the LF program. That thread, defined on lines 5 to 24, repeatedly blocks on line 10 waiting for the user to type something. If the user input is a valid number, then on line 18, the thread calls a built-in thread-safe function lf_schedule_int, passing it a pointer to the physical action named "r" and the amount entered by the user (the 0 argument is irrelevant to the current discussion).

The physical action r, declared on line 29, is an LF construct for providing external, asynchronous inputs to an LF program. The key is that when lf_schedule_int is called, a tag g based on a local measurement T of physical time, such that $\mathcal{T}(g) = T$, is assigned to the event, which is then injected into the program to be handled in tag order.

The LF program reacts to the event created by the call to lf_schedule_int by executing the reaction given on lines 35 through 37. This sets the output named deposit to the amount entered by the user.

The physical action declared on line 29 of Fig. 3 deserves more scrutiny. First, LF, by default, uses the system clock on the machine that runs each federate to assign the timestamp part $\mathcal{T}(g)$ of the tag. As described by Bateni et al. [30], when a federated program is started, it performs a clock synchronization round using the

```
1  target C;
2  reactor CAReplica {
3      input local_update:int;
4      input remote_update:int;
5      input query:bool;
6
7      state balance:int(0);
8
9      output response:int;
10
11     reaction(local_update, remote_update) {=
12         if (local_update->is_present) {
13             self->balance += local_update->value;
14         }
15         if (remote_update->is_present) {
16             self->balance += remote_update->value;
17         }
18     =}
19
20     reaction(query) -> response {=
21         lf_set(response, self->balance);
22     =}
23 }
```

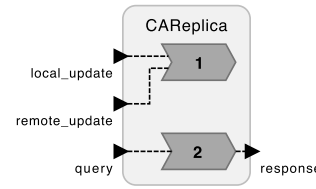

Fig. 2. LF code defining a reactor class that is a replica in a replicated database storing a bank balance and accepting queries and updates. This version is commutative and associative.

```
1  target C;
2  reactor UserInput {
3     preamble {=
4        // Define a function to read user input.
5        void* get(void* r) {
6           int amt;
7           char buf[20];
8           while(1) {
9              // Read a character input.
10             char* ln = fgets(buf, 20, stdin);
11             // Exit if no more input.
12             if (ln == NULL) return NULL;
13             // Parse an input integer.
14             int n = sscanf(ln, "%d", &amt);
15             // If one integer was parsed...
16             if (n == 1) {
17                // Schedule an event.
18                lf_schedule_int(r, 0, amt);
19             } else {
20                // Request another input.
21                printf("Please enter a number.\n");
22             }
23          }
24       }
25    =}
26    input balance:int;
27    output deposit:int;
28
29    physical action r:int;
30
31    reaction(startup) -> r {=
32       pthread_t id;
33       pthread_create(&id, NULL, &get, r);
34    =}
35    reaction(r) -> deposit {=
36       lf_set(deposit, r->value);
37    =}
38    reaction(balance) {=
39       printf("Balance: %d\n", balance->value);
40    =}
41 }
```

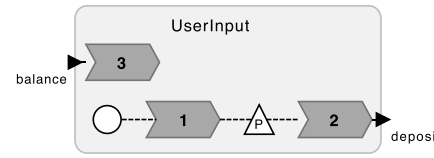

Fig. 3. LF component that gets user input to provide deposits and make withdrawals.

technique of Geng et al. [31]. This ensures that even if the system clock is set manually to some arbitrary value, when the distributed program starts up, all nodes will agree on the current physical time within a few milliseconds. LF also provides a facility for performing ongoing clock synchronization that can correct for clock drifts, but in many systems, it is not really necessary to enable this. We can rely instead on a built-in NTP realization, if that is sufficiently precise for the application.

Once a tag is assigned, the handling of the update throughout the distributed system is a deterministic function of that tag. This gives a clear semantics to the behavior of the system when the actual order of events originating throughout the system is unknown, unknowable, or ambiguous. Moreover, it enables rigorous regression testing, where the UserInput reactor of Fig. 3 is replaced by event generators driven by logical clocks. Those logical clocks can generate events on distributed nodes that are deterministically ordered or even simultaneous.

Composition

We can now put together the components to get a complete, executable program. In Fig. 4, we define an ATM reactor that contains one instance each of UserInput from Fig. 3 and CAReplica from Fig. 2. We then define a federated reactor named ConsistencyFirst that creates two instances of ATM and connects them. Hopefully, the program, with the help of the diagram, is self-explanatory, with the possible exception of the new syntax on line 10. To the left of the arrow, the output port u.deposit is surrounded with (...)+, which, in LF syntax, indicates to use the port as many times as necessary to satisfy all the destinations given to the right of the arrow. In other words, it is a compact syntax for multicast. The reaction defined on lines 13 through 17 simply prevents publishing zero-valued deposits. Hence, a deposit equal to zero can be used to query the current balance and will not generate network traffic.

Execution

When the top-level reactor in an LF program is federated, as it is in Fig. 4, then the code generator, instead of producing a single program, produces as many programs as there are instances of reactors within the top level reactor. In this case, there are two reactors within the top level, so two programs will be generated.

An execution of the program in Fig. 4 is shown in Fig. 5, where there is one terminal for each of the two ATM instances. The top line of each window shows the command that starts each instance. In the lower window, user *b* begins by querying the

```
1  target C;
2  import CAReplica from "CAReplica.lf";
3  import UserInput from "UserInput.lf";
4
5  reactor ATM {
6      input update:int;
7      output publish:int;
8      u = new UserInput();
9      r = new CAReplica();
10     (u.deposit)+ -> r.query, r.local_update;
11     r.response -> u.balance;
12     update -> r.remote_update;
13     reaction(u.deposit) -> publish {=
14         if (u.deposit->value != 0) {
15             lf_set(publish, u.deposit->value);
16         }
17     =}
18 }
19 federated reactor ConsistencyFirst {
20     a = new ATM();
21     b = new ATM();
22     b.publish -> a.update;
23     a.publish -> b.update;
24 }
```

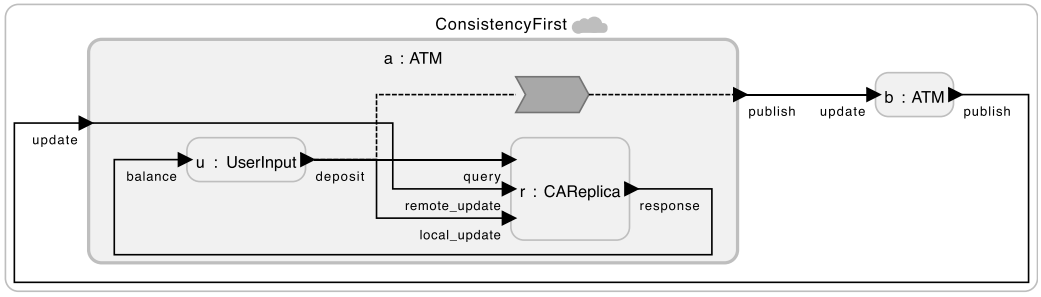

Fig. 4. Federated LF program with two automatic teller machines (ATMs) that can provide deposits and make withdrawals.

current balance by entering "0". The balance is zero. User *b* then deposits 100 dollars. User *a* then withdraws 20 dollars twice. User *b* then queries the balance again, discovering that it is now 60.

The deposits and withdrawals are handled by both ATM federates in the same order, defined by tags that are assigned when schedule is called on line 18 in Fig. 3. If two deposits occur with the same tag, then the reported balance by each user will reflect the aggregate of the two operations. That is, the two deposits are semantically simultaneous.

This simultaneity feature is hard to test with an interactive program like this, but in LF, it is easy to create a regression test that replaces the UserInput with timer-driven inputs, which gives precise control over the tags. The ability to construct such deterministic, distributed regression tests is one of the key advantages of LF.

Variants of the running example

Lest the reader conclude that we are only talking about one rather oversimplified example, we will now point out several variants of this design that are easy to build. First, the example in Fig. 4 has only two federates. LF has a convenient syntax, shown in Fig. 6, for scaling this program to any number of federates. The top-level ConsistencyFirstN reactor has a parameter N defined on line 5 (with default value 4) that specifies how many instances of the ATM reactor to create. Those instances are created on line 7, and each instance is assigned the value N to its own parameter, defined on line 11, which happens to also have the name "N".

The CAReplicaN reactor is similar to CAReplica in Fig. 2, with the only difference being its multiport input, defined on line 29, which can accept *N* input connections. The reaction on lines 34 through 40 iterates over these inputs and adds to the balance any values it finds.

For the example in Fig. 6, it does not matter in what order simultaneous updates are applied because the updates are commutative and associative and no replica reads the result until all updates have been applied. Many distributed applications with shared data, however, do not naturally have commutative and associative merge operations.

Figure 7 shows a variant where, on line 28, an update overwrites the shared value. Such an operation is associative but not commutative. Here, each user update is broadcast to all nodes and, as before, applied before any query for the value is processed. If two updates are logically simultaneous, then both updates will appear in deterministic order at the multiport input (line 19) of the replica instance. Because the order in which these updates appear is deterministic, a priority scheme can be used to determine which update prevails. In this implementation, a bank of *N* nodes is created on line 5, where *N* is the parameter of the federated reactor, which defaults to 4. This parameter is passed down to instances of Node (line 5), each of which in turn passes it down to instances of ReplicaN (line 12). Assume that these nodes have indexes 0 to $N - 1$. Nodes with higher indices have

```
$ bin/ConsistencyFirst_a
Federate 0: Connected to RTI at localhost:15045.
Federate 0: ---- Start execution at time Mon Aug  2 08:39:41 2021
---- plus 125041000 nanoseconds.
Federate 0: Starting timestamp is: 1627918786216634000.
-20
Balance: 80 at elapsed time 27089056000
-20
Balance: 60 at elapsed time 30879016000
0
Balance: 60 at elapsed time 43546736000
```

```
$ bin/ConsistencyFirst_b
Federate 1: Connected to RTI at localhost:15045.
Federate 1: ---- Start execution at time Mon Aug  2 08:39:45 2021
---- plus 216355000 nanoseconds.
Federate 1: Starting timestamp is: 1627918786216634000.
0
Balance: 0 at elapsed time 5382774000
100
Balance: 100 at elapsed time 9567247000
0
Balance: 60 at elapsed time 39196844000
```

Fig. 5. An execution of the LF program in Fig. 4 (slightly elaborated to display (elapsed) logical time $\mathcal{T}(g)$ (in nanoseconds).

priority over nodes with lower ones simply because the iteration on line 25 reads inputs from other members of the bank in the same order as their index. Any write by a node with a higher index will overwrite a write by a node with a lower index that is simultaneous. Hence, simultaneous updates yield deterministic results prioritized by index.

Trading off consistency and availability in LF

Unavailability is a measure of the time it takes for a system to respond to user requests. If it takes a long time (or it never responds), then the system is unavailable, whereas if responses are instantaneous, then the system is highly available.

In the ATM application in Fig. 6, a user request at the i-th ATM is assigned a tag g_i on line 18 of Fig. 3 based on the local physical clock. Hence, $\mathcal{T}(g_i)$ is a good measure of the physical time at which the user has initiated a request. The user's request turns into a tagged deposit output from the UserInput reactor, which gets sent to the query input of the CAReplicaN reactor. That reactor sends back the value of the shared variable at the tag g_i, which, in this design, reflects all updates throughout the system with tags g_i or less.

Note that a read of the value is handled locally on each node. However, this read will have latency that depends on the time it takes for updates to traverse the network. In Fig. 6, notice that the reaction to query on line 41 is defined after the reaction to local and remote updates. In LF semantics, this ensures that the reaction to query is not invoked at tag g_i until after all local and remote updates with tags g_i or less have been processed. It is this property that gives this program strong consistency.

The key question becomes, when can the reaction to the query input on line 41 of Fig. 6 be executed? The physical time between $\mathcal{T}(g_i)$ and the time of that reaction invocation becomes our measure of unavailability.

This scenario matches exactly the scenario leading to the CAL theorem in the "The CAL theorem" section. Hence, the unavailability is given by Eq. 6. This result is intuitive. We will shortly show that the processing offsets can be zero in this case, so assume $O_i = O_j = 0$. If execution times are negligible, then $\mathcal{L}_{ij}$ is just the sum of the network latency and clock synchronization error, and the unavailability at node i due to possible updates at node j is

$$\overline{A}_{ij} = \max\left(L_{ij} + E_{ij}, 0\right). \tag{11}$$

Recall that even though clock synchronization error can be negative, the above maximization ensures that the unavailability is non-negative. If we further assume that clock synchronization errors are negligible compared to network latency, then Eq. 11 tells us that the unavailability at i due to possible updates at j is equal to the network latency from j to i, a satisfyingly intuitive result.

We can easily modify the program to improve availability at the cost of consistency. Specifically, if we replace line 8 of Fig. 6 with this:

```
8 (bank.publish)+ -> bank.updates after 100 ms;
```

then the inconsistency is specified to be $\overline{C} = 100$ ms. The **after** keyword specifies a logical time offset between the sender's tag and the receiver's tag. In other words, it specifies a logical delay between the initiation of an update by a user and the recording of that update in a state variable of each replica. This is exactly

```
target C;
import UserInput from "UserInput2.lf";

federated reactor ReplicatedDataStore(N:int(4)) {
    nodes = new[N] Node(N = N);
    (nodes.publish)+ -> nodes.updates;
}
reactor Node(N:int(2)) {
    input[N] updates:int;
    output publish:int;
    u = new UserInput();
    r = new ReplicaN(N = N);
    u.update -> r.query;
    u.update -> publish;
    r.response -> u.current_value;
    updates -> r.updates;
}
reactor ReplicaN(N:int(2)) {
    input[N] updates:int;
    input query:int;
    output response:int;
    state record:int(0);

    reaction(updates) {=
        for (int i = 0; i < self->N; i++) {
            if (updates[i]->is_present) {
                // Overwrite simultaneous updates on lesser input channels.
              self->record = updates[i]->value;
            }
        }
    =}
    reaction(query) -> response {=
        lf_set(response, self->record);
    =}
}
```

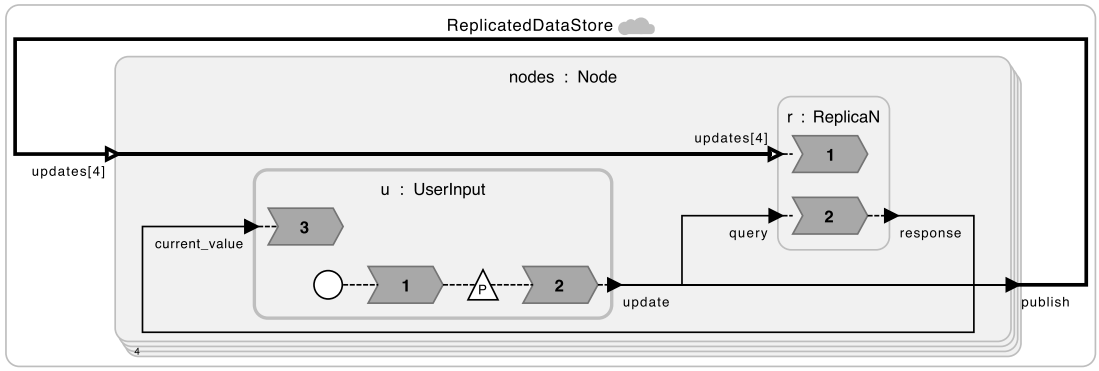

Fig. 7. Federated LF program with any number of nodes that can update a shared data value.

the tag manipulation considered in the "The CAL theorem" section, so, from Theorem 1, the unavailability at node i becomes

$$\overline{A}_i = \max\left(0, \max_{j \in N}\left(\mathcal{L}_{ij} - \overline{C}_{ij}\right)\right),$$

where we have again assumed the processing offsets are zero. Again, if clock synchronization error and execution times are negligible compared to network latencies, then this states that the unavailability is the largest difference between network latency and logical delays. With the choice of $\overline{C} = 100$ ms, if the network latency is less than 100 ms, then unavailability becomes zero. The system can respond instantaneously to user requests.

Even with the logical delay, this design assures eventual consistency because all updates are applied in the same order at all nodes. Note that even local updates are logically delayed, and hence, the same design can be applied even if the merge operation is not associative and commutative, as in the example in Fig. 7.

The price for improving availability in this ATM example is that all queries for the value of the shared variable yield a result that is (logically) 100 ms old. This means that a query for x may not even reflect a recent local update. If the operations are commutative and associative, however, then local updates need not be delayed. We can use the structure of Fig. 4 and apply the logical delay only on the connections that broadcast local updates. We leave it as an exercise for the reader to modify the programs in

```
target C;
import UserInput from "UserInput.lf";

federated reactor ConsistencyFirstN(
    N:int(4)
) {
    bank = new[N] ATM(N = N);
    (bank.publish)+ -> bank.updates;
}
reactor ATM(
    N:int(2)
) {
    input[N] updates:int;
    output publish:int;
    u = new UserInput();
    r = new CAReplicaN(N = N);
    u.deposit -> r.query;
    r.response -> u.balance;
    updates -> r.updates;
    reaction(u.deposit) -> publish {=
        if (u.deposit->value != 0) {
            lf_set(publish, u.deposit->value);
        }
    =}
}
reactor CAReplicaN(
    N:int(2)
) {
    input[N] updates:int;
    input query:int;
    output response:int;
    state balance:int(0);

    reaction(updates) {=
        for (int i = 0; i < updates_width; i++) {
            if (updates[i]->is_present) {
              self->balance += updates[i]->value;
            }
        }
    =}
    reaction(query) -> response {=
        lf_set(response, self->balance);
    =}
}
```

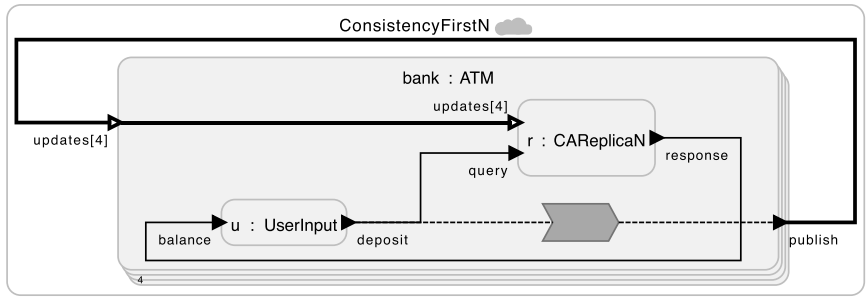

Fig. 6. Federated LF program with any number of ATMs that can provide deposits and make withdrawals.

Figs. 4 and 6 to accomplish this form of bounded inconsistency by inserting after delays. If we were to change the design to apply local updates immediately in situations where the merge operation is neither associative nor commutative, such as the program in Fig. 7, then we would have to do some additional work to ensure eventual consistency, using for example a sorted replace [4].

What happens if the apparent latency exceeds 100 ms? There are two possibilities. We can delay handling of events, thereby increasing unavailability, or we can proceed with processing events as if the inputs are absent, thereby increasing inconsistency. In the "Results and Discussion" section, we describe two coordination mechanisms that we have implemented for LF,

```
1
2  target C;
3  reactor Sensor {
4      output y:int;
5      timer t(0, 10 ms);
6      reaction(t) -> y {=
7          // code in C: produce output y
8      =}
9  }
10 reactor Actuator {
11     input x:int;
12     reaction(x) {=
13         // code in C: Use input x
14     =}
15 }
16 reactor Analytics {
17     input in:int;
18     output out:int;
19     reaction(in) -> out {=
20         // code in C: read in, write out
21     =}
22 }
23 federated reactor {
24     i1 = new Sensor();
25     i2 = new Analytics();
26     i3 = new Actuator();
27     i1.y -> i2.in;
28     i2.out -> i3.x;
29 }
```

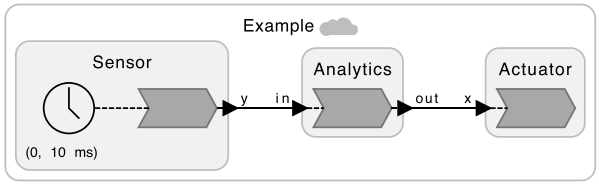

Fig. 8. Structure of an LF program for a simple pipeline.

one of which emphasizes consistency and the other of which emphasizes availability.

Another alternative is to remove the bound on inconsistency altogether while still preserving the property that if the network is repaired, we get eventual consistency. A one-line change in the LF program of Fig. 6 can realize this strategy. If we change line 5 to this subtly different version:

```
8 (bank.publish)+ ~> bank.updates;
```

then there is no upper bound on the inconsistency $\overline{C}_{ij}$. The subtle change is to replace the logical connection -> with a physical connection ~> . In LF, this is a directive to assign a new tag g_i at the receiving end i based on a local measurement of physical time T_i when the message is received such that $\mathcal{T}(g_i) = T_i$. The original tag is discarded. If all connections between federates are physical connections, then the federation no longer has any need for clock synchronization. However, the price we pay is that the order in which updates are applied is now dependent on apparent latencies. We preserve eventual consistency only if the merge operation is associative and commutative. Using physical connections is a draconian measure because it also sacrifices determinacy. This makes it much harder to define regression tests because a correct execution of the program admits many behaviors. The use of logical delays, together with the coordination mechanisms given in the "Results and Discussion" section, offers more control.

Pessimistic evaluation of processing offsets

The processing offsets O_i and O_j are physical time delays incurred on nodes i and j before they can begin handling events. Specifically, node i can begin handling a user input (specifically a write event) with tag g_i at physical time $T_i = \mathcal{T}(g_i) + O_i$. In the absence of any further information about a program, we can use our Γ matrix to calculate these offsets. However, the result is pessimistic because it does not use dependency information that is present in an LF program. The conservative analysis we give here will result in infinite processing offsets for the strongly consistent ATM example. We will show in Results and Discussion that, by using information present in the structure of the LF program, we can derive less conservative processing offsets that turn out to be zero for the ATM application, assuming that the execution times of reactions are negligible.

First, consider nodes that have the possibility of new events appearing asynchronously with timestamps given by the local physical clock, like those in our ATM application. In such a node i, it is generally not safe to process an event with tag g_i until the physical clock T_i exceeds $\mathcal{T}(g_i)$, i.e., $T_i > \mathcal{T}(g_i)$. Otherwise, there is a possibility of processing events out of order. Define a column vector $\boldsymbol{Z}$ such that

$$Z_i = \begin{cases} 0, & \text{if node } i \text{ has local physically time} - \text{stamped inputs,} \\ -\infty, & \text{otherwise.} \end{cases} \tag{12}$$

With this, we require at least that $O \geq Z$. In addition, to ensure that node i processes events in tag order, it is sufficient to ensure that node i has received all network input events with tags less than or equal to g_i before processing any event with tag g_i. With this (conservative) policy, $O_i \geq \max_j\left(\mathcal{L}_{ij} - \overline{C}_{ij}\right)$. The smallest processing offsets that satisfy these two constraints satisfy

$$O = Z \oplus \Gamma O. \tag{13}$$

This is a system of equations in the max-plus algebra. From Baccelli et al. [22] (Theorem 3.17), if every cycle of the matrix Γ has weight less than zero, then the unique solution of this equation is

$$O = \Gamma^* Z, \tag{14}$$

where the Kleene star is (Theorem 3.20 [22]) $\Gamma^* = \boldsymbol{I} \oplus \Gamma \oplus \Gamma^2 \oplus \cdots$. Baccelli et al. show that this reduces to $\Gamma^* = \boldsymbol{I} \oplus \Gamma \oplus \cdots \oplus \Gamma^{N-1}$, where N is the number of processes.

The requirement that the cycle weights be less than zero is intuitive but overly restrictive. It means that along any communication path from a node i back to itself, the sum of the logical delays D_{jk} must exceed the sum of the execution times, network latencies, and clock synchronization errors along the path. This implies that we have to tolerate a nonzero inconsistency somewhere on each cycle. For the strongly consistent ATM example, every node sends messages to every other node, so every pair of nodes requires a nonzero inconsistency in order to satisfy this cycle-mean constraint. For the strongly consistent case, where there are no logical delays, there is no finite solution to Eq. 13.

In practice, programs may have zero or positive cycle means. Theorem 3.17 of Baccelli et al. [22] shows that if all cycle weights are nonpositive, then there is a solution, but the solution may not be unique. If there are cycles with positive cycle weights, then there is no finite solution for O. For the strongly consistent ATM example, there are no logical delays at all, and all cycle weights become positive. In this case, the only solution to

Eq. 13 sets all the processing offsets to ∞. Every node must wait forever before handling any user input. This is, of course, the ultimate price in availability.

Equation 15 is pessimistic because, in the absence of more information about the application logic, we must assume that any network input at node i with tag g_i can causally affect any network output with tag g_i or larger and that network inputs may come from anywhere. In Results and Discussion, we use the fact that LF exposes more information about causal relationships to derive much less pessimistic processing offsets while still preserving causal and eventual consistency.

An example

Figure 8 shows a simple example where the pessimistic evaluation of the processing offsets works and permits us to derive the unavailability in a simple way. This program could realize a simple Internet-of-things application where a local sensor sends data to a cloud service for analysis, and the results of the analysis are used to drive an actuator. This example is a rather trivial application of the CAL theorem, which yields exactly the results we would expect.

There are three federates and hence three communicating nodes. The Γ matrix is given by

$$\Gamma = \begin{bmatrix} 0 & -\infty & -\infty \\ \Gamma_{21} & 0 & -\infty \\ -\infty & \Gamma_{32} & 0 \end{bmatrix} \tag{15}$$

where

- $\Gamma_{21} = X_{21} + L_{21} + E_{21}$,
- $\Gamma_{32} = X_{32} + L_{32} + E_{32}$,

and

- X_{21} is the execution time for the reaction in Sensor,
- L_{21} is the network latency from Sense to Analytics,
- E_{21} is the clock synchronization error from i1 to i2,
- X_{32} is the execution time for the reaction in Analytics,
- L_{32} is the network latency from Analytics to Actuator, and
- E_{32} is the clock synchronization error from i2 to i3.

The $-\infty$ entries in the matrix are a consequence of a lack of communication.

First, we can use the analysis of the "Pessimistic evaluation of processing offsets" section to evaluate the processing offsets. For this example, $N = 3$, so Γ^* in Eq. 14 reduces to

$$\Gamma^* = \mathbf{I} \oplus \Gamma \oplus \Gamma^2.$$

It is straightforward to evaluate this to get

$$\Gamma^* = \begin{bmatrix} 0 & -\infty & -\infty \\ \Gamma_{21} & 0 & -\infty \\ \Gamma_{21}+\Gamma_{32} & \Gamma_{32} & 0 \end{bmatrix}.$$

Intuitively, this matrix captures the fact that the Actuator reactor indirectly depends on the Sensor reactor, something not directly represented in the Γ matrix.

The Z vector of (12) is $Z = [0, -\infty, -\infty]^T$ because the Sensor reactor has a timer, and hence, logical time cannot get ahead of physical time; the other two reactors have no local physically time-stamped inputs and hence can advance logical time ahead of physical time as long as dependencies are respected. We can now evaluate Eq. 14 to get

$$\boldsymbol{O} = \Gamma^* \boldsymbol{Z} = \begin{bmatrix} 0 \\ \Gamma_{21} \\ \Gamma_{21}+\Gamma_{32} \end{bmatrix} \tag{16}$$

Next, we evaluate Eq. 9 to get the unavailability at each node,

$$\boldsymbol{A} = (\boldsymbol{I} \oplus \Gamma)\boldsymbol{O} = \begin{bmatrix} 0 \\ \Gamma_{21} \\ \Gamma_{21}+\Gamma_{32} \end{bmatrix} \tag{17}$$

In this simple case, the unavailability is equal to the processing offsets, which means that the processing offsets capture all the waiting that needs to be done to realize the semantics of the program.

These unavailability numbers are exactly what we would expect without help from the CAL theorem. First, note that the Sensor can react to external stimulus (specifically timer input) immediately. It has no network inputs to worry about, so $\boldsymbol{A}_0 = 0$. The Analytics reactor, however, can react to an input stimulus with timestamp t only at physical time $T = t + X_{21} + L_{21} + E_{21}$. Note that since the Analytics reactor uses its local physical clock to determine this time, and it is possible for E_{21} to be negative, it can even be that $T < t$, something we might not have expected. Similarly, the Actuator reactor can respond when physical time T exceeds $\Gamma_{21} + \Gamma_{32}$. Again, the consequences of clock synchronization error might be unexpected, but otherwise, these results are rather obvious.

These results can be used to determine whether the Analytics reactor can safely be put in the cloud because these results give us the resulting end-to-end delay from sensing to actuation. If the Sensor and Actuator are put on the same node and hence share the same physical clock, then the clock synchronization errors in $\Gamma_{21} + \Gamma_{32}$ will cancel out. This sum becomes the end-to-end latency, the sum of execution times and network latencies, exactly what we would have derived without the CAL theorem.

You can elaborate this example in various ways, and the CAL theorem will reveal subtleties that may have been harder to see. For example, if the Analytics or Actuator reactor have their own asynchronous inputs via physical actions, then the $\boldsymbol{Z}$ vector will change, and the effect of clock synchronization errors will become more complex. More interestingly, the availability of reactions to those asynchronous inputs may be degraded by the network latencies. Even more interestingly, these degradations can be mitigated by putting after delays on the connections between reactors. We leave these calculations to the reader as an exercise.

Determinism, idempotence, and causal consistency

An LF program that has only logical connections and no physical connections has deterministic semantics, in the sense that once tags are assigned, there is exactly one correct execution of the program. The runtime infrastructure is responsible for ensuring

that every reactor is presented with inputs in tag order, that messages are delivered to reactors exactly once, and that reactions to messages with identical tags are invoked according to the order specified by the code. Moreover, because reactors have explicit input and output ports, LF has a notion of a communication channel, a connection between two ports. Each port is guaranteed to have at most one message at any tag. These properties, taken together, make it much easier to design consistent distributed programs and to trade off consistency against availability. These properties automatically deliver what is sometimes called CALM, meaning consistency as logical monotonicity [32,33].

The ACID 2.0 database principle of Helland and Campbell [34] includes an assumption, called idempotence, that operations that are applied more than once have the same effect as operations that are applied exactly once. Given the use of tags in LF, this assumption is automatically met by the infrastructure, so it need not be a concern of the application developer. That is, if any communication fabric is used that retransmits messages, it is up to the infrastructure to ensure that those messages are delivered to a reactor exactly once.

LF also ensures causal consistency. The runtime infrastructure, described next in Results and Discussion, uses the topology of interconnection between reactors together with tags to ensure that no reaction that reads an input or a state variable is invoked until all precedent reactions have been invoked. Note that this does not require that messages be globally ordered! It only requires that each component (each reactor in LF) see messages in tag order and that simultaneous messages (those with the same tag) are handled in precedence order. We discuss in the "Results and Discussion" section how this is achieved in LF.

Results and Discussion

We now describe the two available distributed coordination mechanisms, which we have implemented as an extension of LF, that support arbitrary trade-offs between consistency and availability as network latency varies [30]. With centralized coordination, inconsistency remains bounded by a chosen numerical value at the cost that unavailability becomes unbounded under network partitioning. With decentralized coordination, unavailability remains bounded by a chosen numerical quantity at the cost that inconsistency becomes unbounded under network partitioning. Our centralized coordination mechanism is an extension of techniques that have historically been used for distributed simulation, an application where consistency is paramount. Our decentralized coordination mechanism is an extension of techniques that have been used in distributed databases when availability is paramount.

Centralized coordination

Centralized coordination is based on high-level architecture (HLA) [35] and other distributed simulation frameworks [36,37], with significant extensions that we describe here. Distributed simulation is a relevant problem because, usually, consistency trumps availability. A distributed implementation of a simulation is expected to yield the same results as a nondistributed version, only faster. The HLA is designed for distributed simulation of discrete-event systems, where events have timestamps, and hence addresses a similar problem.

We face two complications, however, that are not present in distributed simulation applications. The first is that, in our context, unlike simulation, events may materialize out of nowhere with tags derived from the local physical clock. Our context, in other words, has users interacting with the system, and hence availability becomes a concern. Simulation has no such users. In LF, we use physical actions to realize asynchronous stimulus from users. A second problem is that the programs in Figs. 6 and 7 have cycles without logical delays, which are not allowed in HLA.

The HLA, like other distributed simulation frameworks, uses a centralized controller called the run-time infrastructure (RTI). Each node that wishes to process a tagged event consults with the RTI, which grants permission to advance its current tag to that tag only when the RTI can assure the node that no event with a lesser tag will later appear. The existence of physical actions and zero-delay cycles in LF complicates this assurance and requires extending the protocols used in HLA.

Our RTI, like those in distributed simulation frameworks, realizes a mechanism similar to vector clocks [38]. Schwartz and Mattern [17] show that any mechanism that preserves causal consistency fundamentally has a complexity of at least that of vector clocks. However, because LF exposes information about which federates communicate with which, we have realized significant optimizations. Our RTI keeps track of the tag to which each federate has advanced, and uses that information, together with network topology information, to regulate the advancement of the current tag at downstream federates based on the activity of their upstream federates. A federate that has no network inputs, for example, can advance its current tag without consulting the RTI because there is no risk of later seeing an incoming message that has a tag less than the tag to which it has advanced. A federate with network inputs, however, must receive an assurance from the RTI, a tag advance grant (TAG) or provisional tag advance grant (PTAG), before it can advance its current tag.

Our first extension over HLA supports zero-delay cycles by introducing a PTAG, where the RTI assures a federate that there will be no future message with tags less than some g but makes no promises about messages with tags equal to g. This permits a federate to advance its current tag to g and execute any reactions with no dependence, direct or indirect, on network inputs. When it has executed such reactions and the next reaction in the reaction sequence depends on network inputs, then the federate is required to block until it either accepts messages on those network inputs or receives an assurance that no message is forthcoming with tag g on those inputs. Such an assurance is similar to the null messages of Chandy and Misra [39].

In some cases, providing such an assurance is easy. If the upstream federates have all advanced their own current tag beyond g and have informed the RTI of this fact, then the RTI can provide the required assurance to the downstream federate. As long as that assurance message is sent along the same order-preserving message channel as tagged messages, then when a federate receives the assurance, it knows it has received all relevant tagged messages and hence can proceed. However, if there are cycles between federates that lack logical delays, a federate may need to send a null message, an indicator that no message with tag g is forthcoming, even before it has completed processing of all events with tag g. It can send such a null message as soon as it has executed or chosen not to execute all reactions that are capable of producing the relevant network output. Such null messages are similar to those of Chandy and Misra [39] but are only needed in particular circumstances.

When there are physical actions, however, things are still a bit more complicated. Consider the program in Fig. 7, focusing

particularly on the graphical rendition at the bottom. This example instantiates four federates, each an instance of the Node reactor class. Each federate has four input channels on its updates input port. Under centralized coordination, a federate cannot advance its current tag to g until it receives either a TAG or a PTAG from the RTI with value g. It also cannot advance to g until its physical clock exceeds $\mathcal{T}(g)$ because it has a physical action. When can the RTI provide a TAG or PTAG message? This depends on how each federate produces network outputs. Each federate has a physical action in its UserInput reactor that triggers a network output on the publish port. The tag g of that output will have $\mathcal{T}(g)$ taken from the local physical clock. Hence, as soon as the physical clock of a federate exceeds $\mathcal{T}(g)$, downstream federates can be assured that there will be no forthcoming message with timestamp g or less.

Unlike decentralized coordination (see the "Decentralized coordination" section below), centralized coordination does not rely on clock synchronization except to give a meaning to tags originated by distributed physical actions. Instead of relying on clock synchronization, in centralized coordination, each federate that has a physical action that can result in network outputs must notify downstream federates as its physical clock advances. This is done by periodically sending to the RTI a time advance notice (TAN) message with a time t; this message is a promise to not produce future messages with tags g where $\mathcal{T}(g) \leq t$ and hence is also similar to the null messages of Chandy and Misra [39].

Unlike Chandy and Misra's technique, in LF, null messages are only required when communication between federates forms a cycle without logical delays and when physical actions trigger network outputs. Unfortunately, all of the applications considered in the "Trading off consistency and availability in practice" section have such cycles and physical actions and therefore require null messages. LF provides a mechanism to control the frequency of the TAN messages, thus controlling the overhead, but as the frequency of messages decreases, the cost in unavailability increases. This overhead is avoided in decentralized coordination, explained next, but at the cost that consistency is sacrificed under network partitioning.

Decentralized coordination

Decentralized coordination extends a mechanism first described by Lamport [40], first applied to explicitly time-stamped distributed systems in Programming Temporally Integrated Distributed Embedded Systems (PTIDES) [41], and reinvented at Google to form the core of Google Spanner [42]. All three of these use timestamps to define the logical ordering of events and physical clocks to determine when it is safe to process time-stamped events. The physical clocks are assumed to be synchronized with a bound on the clock synchronization error. All three also assume a bound on network latency. If these assumptions are met at run time, then all messages will be processed in timestamp order without any centralized coordination.

Relying on physical clocks has an advantage with respect to availability because we can assume that, even in the presence of complete network partitioning, physical clocks continue to advance. If progress is governed by the physical clocks, then unavailability can be bounded even with no network connectivity. This contrasts with centralized coordination, where inconsistency can be bounded, but loss of network connectivity leads to loss of availability.

Safe to advance offset

In our implementation of decentralized coordination in LF, each federate can have an optional safe-to-advance (STA) offset given by the programmer. The meaning of the STA offset is that if a federate has an earliest pending event with tag g, then it can advance its current tag to g when current physical time T satisfies $T \geq \mathcal{T}(g) + \text{STA}$. Hence, to handle a user request that gets assigned tag g, the federate needs to wait at least until physical time exceeds $\mathcal{T}(g)$ by the STA offset. Put another way, the STA offset is a time interval beyond $\mathcal{T}(g)$ that a federate needs to wait before it can assume that it will not later receive any input messages with tags less than g. By default, STA = 0. The STA offset is closely related to the safe-to-process offset of Zhao et al. [41] but is more provisional. It gives a time threshold for committing to a tag advance but not necessarily fully processing that tag advance. Put another way, it gives a time threshold at which the federate can assume it has seen all messages with tags less than g, but it cannot necessarily assume it has seen all messages with tags equal to g. This distinction turns out to be important for the replicated data store examples we have seen.

Obviously, the STA offset affects availability and is clearly closely related to the processing offset of Definition 3. There is, however, a subtle but important distinction. The processing offset of Definition 3 is a property of a trace, an actual execution, whereas the STA is a specification. The LF code generator generates code where, when executed, every trace will have the property that

$$O_i \geq \text{STA}_i \tag{18}$$

for each federate i. If the STA offset is not sufficiently large for a particular program, then the consistency requirements of the program will not be met. Our task, therefore, is to determine sufficiently large STA offsets such that, if the observed apparent latencies are within our assumed constraints, the program will process all events in tag order, thereby achieving the desired consistency. Only when the apparent latencies exceed our assumed constraints will the program sacrifice consistency in order to maintain availability.

The STA offsets depend on assumed bounds on apparent latency, and vice versa, the assumed bounds on apparent latency depend on the STA offsets, which brings us to a second subtlety. In Definition 4, apparent latency is also a property of a trace, whereas, to use it to derive the STA offsets, we need to use it as a bound on all reasonable traces. Any assumed bound may be exceeded in practice (e.g., the network becomes partitioned), and the strategy of decentralized coordination is to sacrifice consistency rather than availability when this occurs. This condition will be detectable, and LF supports specification of fault handlers for such conditions (see the "Adaptation" section).

A third subtlety is that, in Definition 4, the apparent latency is a property of a pair of processes, sequential procedures where one sends updates to another. LF, however, is a more richly structured language. Federates themselves may be concurrent, running in parallel on multicore machines, for example, and communication between federates is mediated by input and output ports that, pairwise, give specific communication channels over which messages with monotonically increasing tags flow. As a consequence, apparent latency between one federate and another may vary depending on which communication channel between the two is used. Moreover, each pair of send-receive ports may have a different logical delay.

Causality

To analyze an LF program, we need to redo that analysis of the "The CAL theorem" section using the structure of the program. Specifically, it is possible to tell by looking at the program whether an event at one port can result in an event at another port, and we can find bounds on the relationship between the tags of these two events. This analysis must be done carefully, however, because we have to distinguish whether an event at one port can cause an event at another from whether it can influence an event at another port.

To determine whether a message with a certain tag can exist, we need to analyze the counterfactual causality properties of the LF program. Counterfactual causality [43] is a relation between events e_1 and e_2 where e_2 would not occur were it not for the occurrence of e_1. We distinguish this from causal influence, where event e_1 can causally affect e_2 [20]. In an LF program, any input to a reactor with tag g can causally affect any output with tag larger than g and some outputs with tag equal to g because a reaction to that input can change the state of the reactor. However, only some inputs counterfactually cause particular outputs, which then in turn counterfactually cause other inputs. Specifically, consider a reactor like this:

```
1 reactor DirectFlowThrough {
2 input in:int;
3 output out:int;
4 reaction(in) -> out {=
5 ...
6 =}
7 }
```

Because of the reaction signature, we assume that an event at input named in can counterfactually cause an event at the output named out. We do not need to analyze the body of the reaction (which is written in the target language) to determine this fact. In contrast, consider:

```
1 reactor CausalInfluence {
2 input in1:int;
3 input in2:int;
4 output out:int;
5 state s:int(0);
6 reaction(in1) {=
7 ...
8 =}
9 reaction(in2) -> out {=
10 ...
11 =}
12 }
```

Here, input in1 causally influences output out (because the first reaction can change the state, and the second reaction can use that updated state), but it does not counterfactually cause the output. For the output to occur, a message must arrive on in2.

In the above example, an input with tag g on in1 can causally influence any output with tag g or larger. If the reactions were given in the opposite order, then it would only be able to causally influence an output with tag larger than g. This is because, given simultaneous inputs, reactions of a reactor are invoked in the order that they are declared. This distinction proves important when analyzing the distributed replicated databases considered earlier.

Using actions, a reactor can declare a logical delay:

```
1 reactor IndirectFlowThrough {
2 input in:int;
3 output out:int;
4 logical action a:int(10 ms); // minimum
delay of 10 ms.
5 reaction(a) -> out {=
6 ...
7 =}
8 reaction(in) -> a {=
9 ...
10 =}
11 }
```

In this example, the logical action has a minimum delay property (set to 10 ms). The pair of reactions, taken together, reveal that the input with tag g can counterfactually cause an output with tag g', where $\mathcal{T}(g')$ is larger than $\mathcal{T}(g)$ by at least 10 ms. This introduces a logical delay on the path from in to out.

Consider:

```
1 reactor Composition {
2 a = new IndirectFlowThrough();
3 b = new DirectFlowThrough();
4 a.out -> b.in;
5 }
```

Because of the connection, we can infer that output a.out can counterfactually cause input b.in with no logical delay. Moreover, because of the minimum delay property, we can infer that input a.in can counterfactually cause input b.in with logical delay of at least 10 ms.

The connection may also have a logical delay (written with the **after** keyword), as in:

```
4 a.out -> b.in after 20 ms;
```

Now, the program reveals that input a.in can counterfactually cause input b.in with logical delay of at least 30 ms.

Safe-to-assume absent

Similar to the STA offset (which is, essentially, found in the studies by Lamport, PTIDES, and Spanner), we extended LF to allow specification of a safe-to-assume-absent (STAA) offset associated with a network input port. The STAA offset is used to constrain when a reaction that depends on an input port can be invoked. Specifically, it asserts that the invocation of any reaction at tag g that depends, directly or indirectly, on a network input port p_i is delayed until either an input is received on port p_i with tag g or the physical clock T_i at i satisfies

$$T_i \geq \mathcal{T}(g) + \mathrm{STA}_i + \mathrm{STAA}_{p_i}. \tag{19}$$

At this physical time, federate i assumes it has seen all inputs at port p_i with tags less than or equal to g (vs. the STA offset alone, when it can assume it has seen all inputs with tags less than g). If no message has arrived with tag g, the federate assumes that there is no message with tag g. It would be an error, to be handled as a fault condition, to later receive a message with tag g.

A positive STAA offset causes the federate to block execution of reactions in the relevant reactor until either physical time advances sufficiently or a message arrives. In this circumstance, a null message could be used to reduce the amount of blocking, but, unlike with centralized coordination, no null message is required to make progress. It is sufficient for physical time to

```
target C {
    coordination: decentralized
}

import UserInput from "UserInput2.lf";
import ReplicaN from "ReplicatedDataStore.lf";

federated reactor (
    N:int(4)
) {
    u = new[N] UserInput();
    r = new[N] ReplicaN(N = N);
    u.update -> r.query;
    r.response -> u.current_value;
    (u.update)+ -> r.updates;
}
```

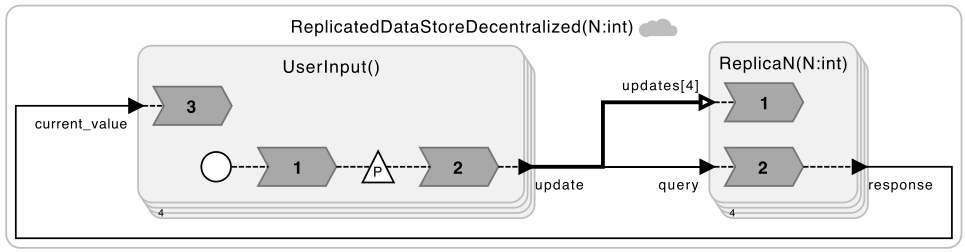

Fig. 9. Version of the replicated data store of Fig. 7 suitable for decentralized coordination, where UserInput and ReplicaN have been split into separate federates so that they can independently advance their current tags.

advance. Using a null message would just be an optimization that may allow progress sooner.

As we will see below, the STA and STAA offsets together ensure that any event that causally influences another is processed first. To determine the STA, we need to consider causal influence, but to determine STAA, we only need to consider counterfactual causality.

Decentralized coordination for the replicated data store

Consider now the replicated data store in Fig. 7. The pessimistic analysis of the "Pessimistic evaluation of processing offsets" section, for this program, yields an infinite processing offset for all federates. We will now show that, by leveraging the semantics of LF and our extensions to its runtime, the program can be executed correctly with finite STA and STAA offsets. We show how to determine these offsets for each of the federates and their input ports.

First, it will shortly become obvious that we need to separate ReplicaN and UserInput into distinct federates, even if they run on the same host, so that they can independently advance their current tags.

In the current implementation of LF, an entire federate, with all its component reactors, advances the current tag together. In principle, some future implementation of LF could allow component reactors to independently advance their current tags. This could be accomplished using mechanisms similar to LF's federated execution. However, for now, the only available mechanism to permit independent advancement of tags is to separate the reactors into distinct federates. Sometimes, however, it is not possible to create such a separation. The LF code generator assumes that any two reactions of the same reactor share state. It does not analyze the target code to check whether this is the case. As a consequence, if the reactions of UserInput and ReplicaN in Fig. 7 were instead reactions of the same reactor, then it would not be possible to separate them into distinct federates nor to independently advance their tags.

The refactored program is shown in Fig. 9. Given such a separation, note that each of the four instances of ReplicaN receive inputs from each of the four instances of UserInput, including the one running on the same host.

To further simplify the explanation, we have reduced the program to that shown in Fig. 10, which is a minimal version that avoids the compact bank and multiport syntax of LF and renames all the input ports so that they have unique names. There are a total of four federates and six input ports now, so we need to determine four STA offsets and six STAA offsets.

A key property of our execution policy for LF programs is that a federate advances its current tag to g only after it has completed handling all events with lesser tags, and then it completes handling of all events with tag g before advancing to another larger tag. That is, even if the federate is executing reactions concurrently (e.g., on a multicore machine), it performs a barrier synchronization with each tag advance. There is no such barrier synchronization across federates, but we need the barrier synchronization within a federate, as will become obvious.

Consider first federate f_1, the UserInput at the upper left of Fig. 10. Suppose that the physical action (depicted as a triangle with a "P") triggers and is assigned tag g_1 using the local physical clock. The question now is, when can the federate advance its tag to g_1? It has to ensure that it has seen all inputs with lesser tags, including events that may have been sent to port p_5. For the first triggering of the physical action, it is evident from the program structure that there is no event at p_5 with a lesser tag because all events at p_5 are ultimately counterfactually caused by this same physical action. Therefore, the federate can safely advance to tag g_1 and invoke its reaction 2 with no delay. Thus, it seems that $STA_1 = 0$ could work for federate f_1, at least for this first event.

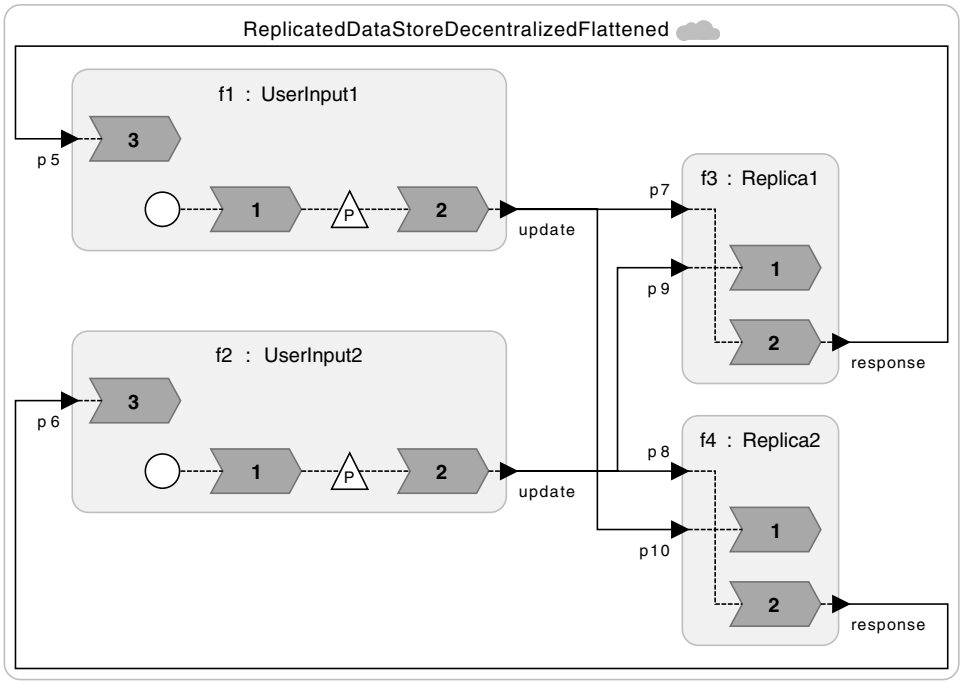

Fig. 10. Minimal version of the replicated data store of Fig. 9 with input ports renamed to all be unique.

Once federate f_1 has advanced to g_1, it will block any further advances until physical time advances past $\mathcal{T}(g_1) + \text{STA}_1 + \text{STAA}_{p_5}$. Assuming that it does this correctly, then for the next triggering of the physical action with tag $g_2 > g_1$, the federate will not even face this question of whether to advance its tag to g_2 until it has completed processing events with tag g_1. More generally, each time the physical action triggers with tag g_n, $n > 1$, by the time the federate is considering advancing its tag to g_n, it will have completed processing all events with tag g_{n-1}. Therefore, it does not need to wait for physical time to further advance. Hence, by induction, $\text{STA}_1 = 0$ for all tag advances. The same argument applies for federate f_2, yielding

$$\text{STA}_1 = 0 \tag{20}$$

$$\text{STA}_2 = 0. \tag{21}$$

To determine STAA_{p5}, we can follow all physical time lags that might occur on the path from the original source. We now make a critical assumption that is required to ensure finite offsets:

Assumption 1 *Reaction 2 of f_1 is invoked exactly at $\mathcal{T}(g_1) + STA_1$ or negligibly thereafter.*

We will see in the "Unavailability in the replicated data store" section how this assumption can be enforced by the LF program, but first, we determine the consequences of this assumption. With it, any STAA_{p5} that satisfies the following will suffice:

$$\text{STAA}_{p_5} \geq \text{STA}_3 + \max\left(\text{STAA}_{p_7}, \text{STAA}_{p_9}\right) + X_{31} + X_{32} + L_{13} + E_{13},$$

where X_{ij} is an execution time bound on reaction j of federate f_i, L_{ij} is a communication latency bound on messages from f_j to f_i, and E_{ij} is a bound on the clock synchronization error from f_j to f_i. The maximization and the presence of X_{31} is a consequence of the LF semantics that requires that if reactions 1 and 2 of the same reactor are both enabled at any tag g, then reaction 1 must run to completion before reaction 2 is invoked.

Let us make a simplifying assumption to manage the complexity of this (this assumption, unlike Assumption 1, is not necessary but drastically simplifies our example). Specifically, let us assume that execution time bounds are negligible compared to communication latencies. With this assumption, we get

$$\text{STAA}_{p_5} \geq \text{STA}_3 + \max\left(\text{STAA}_{p_7}, \text{STAA}_{p_9}\right) + L_{13} + E_{13}. \tag{22}$$

We can write a similar inequality for STAA_{p_6},

$$\text{STAA}_{p_6} \geq \text{STA}_4 + \max\left(\text{STAA}_{p_8}, \text{STAA}_{p_{10}}\right) + L_{24} + E_{24}. \tag{23}$$

So far, we have $\text{STA}_1 = \text{STA}_2 = 0$ and these two inequalities. Let us now look at STA_3.

The question is, given an event with tag g_3 that federate f_3 wishes to process, how much physical time should it wait before advancing to tag g_3? First, this federate has no local sources of events (actions or timers), so the event must be an input on either port p_7 or p_9. In either case, in order to advance to g_3, the federate needs to be assured that it has seen all inputs earlier than g_3 on the other port in order to ensure causality. (LF assumes that messages on each channel are delivered in tag order.)

If the input has arrived on p_7, then it requires

$$\text{STA}_3 \geq \text{STA}_2 + X_{22} + L_{32} + E_{32},$$

which is obtained by following the counterfactual causality chain upstream from p_9. Using $\text{STA}_2 = 0$ and the negligible execution time assumption,

$$\mathrm{STA}_3 \geq L_{32} + E_{32}.$$

If the input with tag g_3 has arrived instead on p_9, then we require

$$\mathrm{STA}_3 \geq \mathrm{STA}_1 + X_{12} + L_{31} + E_{31}. \tag{24}$$

Combining these and ignoring execution times, we get

$$\mathrm{STA}_3 \geq \max\left(L_{32} + E_{32}, L_{31} + E_{31}\right).$$

There are no further constraints on STA_3, so we can simply set

$$\mathrm{STA}_3 = \max\left(L_{32} + E_{32}, L_{31} + E_{31}\right). \tag{25}$$

Similarly,

$$\mathrm{STA}_4 = \max\left(L_{41} + E_{41}, L_{42} + E_{42}\right). \tag{26}$$

Similar reasoning leads to

$$\begin{aligned}
\mathrm{STAA}_{p_7} &= L_{32} + E_{32} - \mathrm{STA}_3 = \min\left(0, L_{32} + E_{32} - L_{31} - E_{31}\right) \\
\mathrm{STAA}_{p_9} &= L_{31} + E_{31} - \mathrm{STA}_3 = \min\left(0, L_{31} + E_{31} - L_{32} - E_{32}\right) \\
\mathrm{STAA}_{p_8} &= L_{41} + E_{41} - \mathrm{STA}_4 = \min\left(0, L_{41} + E_{41} - L_{42} - E_{42}\right) \\
\mathrm{STAA}_{p_{10}} &= L_{42} + E_{42} - \mathrm{STA}_4 = \min\left(0, L_{42} + E_{42} - L_{41} - E_{41}\right).
\end{aligned}$$

There is no point in having a negative STAA offset, so

$$\mathrm{STAA}_{p_7} = \mathrm{STAA}_{p_9} = \mathrm{STAA}_{p_8} = \mathrm{STAA}_{p_{10}} = 0. \tag{27}$$

Finally, from Eqs. 22 and 23, we can set

$$\begin{aligned}
\mathrm{STAA}_{p_5} &= \mathrm{STA}_3 + \max\left(\mathrm{STAA}_{p_7}, \mathrm{STAA}_{p_9}\right) + L_{13} + E_{13} \\
&= \max\left(L_{32} + E_{32}, L_{31} + E_{31}\right) + L_{13} + E_{13}
\end{aligned} \tag{28}$$

$$\begin{aligned}
\mathrm{STAA}_{p_6} &= \mathrm{STA}_4 + \max\left(\mathrm{STAA}_{p_8}, \mathrm{STAA}_{p_{10}}\right) + L_{24} + E_{24} \\
&= \max\left(L_{41} + E_{41}, L_{42} + E_{42}\right) + L_{24} + E_{24}.
\end{aligned} \tag{29}$$

As a sanity check, let us simplify further by assuming that f_1 and f_3 are mapped to the same host, so that $E_{31} = E_{13} = E_{42} = E_{24} = 0$, and L_{31}, L_{13}, L_{42} and L_{24} are all negligible. Under these assumptions, we get the following total results:

$$\begin{aligned}
\mathrm{STA}_1 &= 0 \\
\mathrm{STA}_2 &= 0 \\
\mathrm{STA}_3 &= \max\left(L_{32} + E_{32}, 0\right) \\
\mathrm{STA}_4 &= \max\left(L_{41} + E_{41}, 0\right) \\
\mathrm{STAA}_{p_5} &= \max\left(L_{32} + E_{32}, 0\right) \\
\mathrm{STAA}_{p_6} &= \max\left(L_{41} + E_{41}, 0\right) \\
\mathrm{STAA}_{p_7} &= 0 \\
\mathrm{STAA}_{p_9} &= 0 \\
\mathrm{STAA}_{p_8} &= 0 \\
\mathrm{STAA}_{p_{10}} &= 0.
\end{aligned}$$

Further, let us assume that clock synchronization error is negligible compared to network latencies. Then, we get:

$$\begin{aligned}
\mathrm{STA}_1 &= 0 \\
\mathrm{STA}_2 &= 0 \\
\mathrm{STA}_3 &= L_{32} \\
\mathrm{STA}_4 &= L_{41} \\
\mathrm{STAA}_{p_5} &= L_{32} \\
\mathrm{STAA}_{p_6} &= L_{41} \\
\mathrm{STAA}_{p_7} &= 0 \\
\mathrm{STAA}_{p_9} &= 0 \\
\mathrm{STAA}_{p_8} &= 0 \\
\mathrm{STAA}_{p_{10}} &= 0.
\end{aligned} \tag{30}$$

These results are intuitive. They show that, at the UserInput federates, when a physical action triggers with tag g, the federate can immediately advance its current tag to g, so reaction 2 can be immediately invoked, resulting in a network output. Whether to invoke reaction 4, the UserInput federates cannot be determined until physical time exceeds $\mathcal{T}(g)$ by a bound on the network latency from the other host, a satisfyingly intuitive result because that is where a remote update may occur.

At the Replica federates, when they receive an input with tag g, they can advance to tag g only when physical time exceeds $\mathcal{T}(g)$ by the bound on the network latency from the other host. This too is intuitive because only at that physical time can they be sure there is no forthcoming message from the other host with a lesser tag.

For this particular example, at the UserInput federates, as long as the assumptions on network latency and clock synchronization are satisfied, there will be a network input with tag g, and reaction 4 will be invoked. However, the LF infrastructure cannot be sure that this is the case without imposing further constraints on the target code in reaction 2 of UserInput and reaction 2 of Replica. Those reactions are free to choose to not produce an output.

As of this writing, in LF, the STA and STAA offsets must be derived by hand and provided as part of the specification of the program. We leave it to future work to derive these thresholds automatically given assumptions about apparent latency. This will require performing analysis of the structure of the program and rejecting programs that result in infinite values for these offsets. The analysis is simple for this program, but it could be quite challenging in general. For example, if UserInput had a second physical action and there were a logical delay D somewhere along the path from its update output back to its current_value input, then $\mathrm{STA}_i = 0$ is not necessarily any longer valid.

Why did we have to separate UserInput and Replica into distinct federates? Were they in the same federate, then $\mathrm{STA}_3 = \mathrm{STA}_1$. From Eq. 24, we have the constraint that $\mathrm{STA}_3 > \mathrm{STA}_2$. Correspondingly, $\mathrm{STA}_4 > \mathrm{STA}_1$ and $\mathrm{STA}_4 = \mathrm{STA}_2$. Combining these, we get $\mathrm{STA}_1 > \mathrm{STA}_2 > \mathrm{STA}_1$, a constraint that is not satisfiable.

Unavailability in the replicated data store

Using the most simplified result, given by Eq. 30, we can see the consequences of the CAL theorem for the replicated data store example. This program is strongly consistent. There are no logical delays, so each replica will agree on the value of the shared variable at every tag. A user who issues a query gets a reply when reaction 4 of UserInput is invoked. From Eq. 30, we see that $\text{STAA}_{p5} = L_{32}$ and $\text{STAA}_{p6} = L_{41}$, which means that the time it takes to respond to a user query is at most the network latency between nodes.

This is intuitive and not surprising. However, there is more subtle consequence. Recall Assumption 1, that reaction 2 of f_1 is invoked exactly at $\mathcal{T}(g_1) + \text{STA}_1$ or negligibly thereafter. Because of the barrier synchronization for advancement of the current tag in a federate, this assumption may not be met if the physical action triggers too closely after its previous trigger, specifically within L_{32}. If the physical action triggers while UserInput1 is waiting on port p_5, then there will be a delay in the invocation of reaction 2 and the derived STA and STAA offsets are no longer assured to be valid. A fault condition may occur.

Fortunately, LF provides mechanisms to prevent such eventualities. First, a physical action can have a minimum spacing parameter, a minimum logical time interval between tags assigned to events. When the environment tries to violate this constraint by issuing requests too quickly, the programmer can specify one of three policies: drop, replace, or defer. The drop policy simply ignores the event. The replace policy replaces any previously unhandled event or, if the event has already been handled, defers. The defer policy assigns at tag g to the event with timestamp $\mathcal{T}(g)$ that is larger than the previous event by the specified minimum spacing. This feature of the language can be used to help protect a system against denial of service attacks that might otherwise trigger fault conditions.

While the minimum spacing parameter ensures that tags are sufficiently spaced, it does not, by itself, ensure that the scheduler will prioritize execution of reaction 2 so as to satisfy Assumption 1. LF provides a mechanism to ensure this, a deadline that can be associated with a reaction. The syntax for this is as follows:

```
38 physical action r;
39 reaction(r) {=
40 ... normal case
41 =} deadline(1 ms) {=
42 ... exception case
43 =}
```

The semantics of an LF deadline is that if the reaction to an event with tag g is invoked at a physical time $T > \mathcal{T}(g) + d$, where $d \in \mathbb{T}$ is a time interval specified by the deadline, then instead of invoking the "normal case" reaction, the "exception case" reaction will be invoked. This provides a mechanism to handle overload conditions, but, more importantly, the deadline provides a hint to the scheduler to prioritize invocation of this reaction. Indeed, LF uses an earliest-deadline-first scheduling policy, thereby ensuring, for sufficiently simple programs, that Assumption 1 will be met.

Adaptation

Whether we use centralized or decentralized coordination, our assumptions about network latency may be violated in the field. For example, the network could fail altogether. We treat such violations of assumptions as faults. For most applications, it is imperative to provide hooks for the system to adapt when these occur. How to adapt is very much application dependent, but the CAL theorem tells us that any adaptation will require giving up some measure of either consistency or availability or both. In a tiered, heterogeneous network, it is possible for some links in a system to fail while others continue working. For example, a factory system may lose connectivity to the cloud while still preserving the local area network. Fortunately, LF provides hooks with fine enough granularity to be able to accommodate such heterogeneous networks.

Centralized coordination bounds inconsistency at the expense of availability. When a network connection fails, components in the system may be unable to advance logical time and therefore become unable to respond to user requests (losing availability). In LF, dependencies between federates are used to regulate advancement of time, so, in some cases, a careful design may be sufficient to keep safety-critical subsystems responsive. For other cases, the system may need to adapt to the new conditions.

The primary mechanism provided in LF to detect availability violations is the deadline construct. Suppose we replace lines 38 to 40 in the UserInput reactor (Fig. 3) with this:

```
38 reaction(balance) {=
39 printf("Balance: %d\n", balance->value);
40 =} deadline(100 ms) {=
41 printf("Apologies for the delay! Your
balance is %d\n", balance->value);
42 =}
```

The use of deadline here means that if the reaction to an event with tag g is invoked at a physical time T that exceeds the logical time $\mathcal{T}(g)$ by more than 100 ms, then the second body of code will be invoked instead of the first. The deadline handler can respond in an application-specific way, for example by switching to a different mode of operation or by safely shutting down a system.

The deadline of 100 ms can be interpreted as a specified bound on unavailability and used for driving system design decisions. For example, let us assume that the UserInput reactor has an upstream reactor (denoted below by subscript 1) whose output port connects to the input port of UserInput (denoted by subscript 2) with a logical delay of 10 ms. Assuming further that the processing offsets are zero, which can be derived from Eq. 14 using Theorem 3.17 of Baccelli et al. [22], the CAL theorem tells us that the unavailability at UserInput is

$$\overline{A}_2 = \max\left(0, \max\left(X_{21} + L_{21} + E_{21} - 10\text{ms}, X_{22}\right)\right)$$

The deadline of 100 ms can then be interpreted as an explicit requirement on the latencies.

$$\max\left(X_{21} + L_{21} + E_{21} - 10\text{ms},\ X_{22}\right) < 100\text{ms}$$

If X_{22} is negligible, after simplifying the inequality, we have

$$X_{21} + L_{21} + E_{21} < 110\text{ms}.$$

Thus, for the deadline to be met, the execution time of the upstream reactor, the time it takes to communicate the outputs to UserInput, and the clock synchronization error between the two reactors must be less than 110 ms. Given these requirements, deploying the upstream reactor on the Cloud might be risky because of potentially high latency, whereas deploying the

reactor onto an edge device could help reduce the likelihood of deadline violations.

One drawback of the current implementation of the LF deadline is that a violation is only detected when the balance input finally appears. In the presence of a network partition, that will not occur until the network is repaired. We leave it as further work to find an extension to the language that can detect earlier a loss of availability exceeding a specified threshold.

When we want to bound unavailability instead of inconsistency, we should use decentralized coordination. Decentralized coordination may also be more efficient, because it does not require null messages to handle physical actions, but it requires clock synchronization, which also increases network traffic.

In LF, we can replace lines 11 through 18 in Fig. 2 with this:

```
11 reaction(local_update, remote_update) {=
12 if (local_update->is_present) {
13 self->balance += local_update->value;
14 }
15 if (remote_update->is_present) {
16 self->balance += remote_update->value;
17 }
18 =} staa(100 ms) {=
19 ... handle fault condition ...
20 =}
```

The addition of the **staa** clause has two effects. First, it specifies that it is safe to assume that the triggering inputs are absent at a logical time $\mathcal{T}(g)$ if they have not arrived by physical time $\mathcal{T}(g) + 100$ ms. That is, this specifies the STAA offset for those input ports. This ensures availability because the reaction can be executed as long as the local physical clock keeps running. Second, the **staa** clause gives a body of code to execute if and when an input with timestamp $\mathcal{T}(g)$ (or less) arrives later than $\mathcal{T}(g) + 100$ms. Again, how to handle this fault condition is application specific. A database application, such as Google Spanner, may, for example, overlay a transaction schema on top of the mechanisms provided by decentralized coordination and reject a transaction when such a fault occurs.

Conclusion

Our generalization of Brewer's CAP theorem, the CAL theorem, quantifies the relationship between inconsistency, unavailability, and apparent latency in distributed systems. Apparent latency includes network latency, execution time overhead, and clock synchronization error. The relationship is a given as a linear system of equations in a max-plus algebra. We show how this relationship enables deliberate choices about availability and consistency and how fault handlers can adapt the system when these choices cannot be respected because of system failures. Moreover, because the CAL theorem gives the numerical relationships between consistency, availability, and network latency, it can serve to guide placement of software components in end devices, in edge computers, or in the cloud. The consequences of such choices can be derived rather than measured or intuited.

We have shown how the LF coordination language enables arbitrary trade-offs between consistency and availability as apparent latency varies. We have extended the implementation of LF with two forms of coordination for distributed programs. With centralized coordination, inconsistency remains bounded by a chosen numerical value at the cost that unavailability becomes unbounded under network partitioning. With decentralized coordination, unavailability remains bounded by a chosen numerical quantity at the cost that inconsistency becomes unbounded under network partitioning. In both cases, LF semantics provides predictable and repeatable behaviors in the absence of faults. In the case of decentralized coordination, a simple fault handling mechanism enables an application to react in controlled ways to loss of consistency while preserving availability. For centralized coordination, a deadline violation handler serves as a fault handler for loss of availability while preserving consistency.

Both coordination mechanisms given here are significant extensions over prior art. Our centralized coordination extends previous methods that have been used for distributed simulation to support asynchronous injection of user-input events and cycles in the communication topology. Our decentralized coordination extends previous methods used for distributed databases to enable better support for cyclic communication structures and asynchronously injected user events.

Acknowledgments

The authors thank E. Brewer, I. Incer, and anonymous reviewers for helpful suggestions on an earlier version. **Funding**: The work in this paper was supported in part by the National Science Foundation award #CNS-1836601 (Reconciling Safety with the Internet); the iCyPhy (Industrial Cyber-Physical Systems) research center, which is supported by Denso, Siemens, and Toyota; and the German Federal Ministry of Education and Research (Software Campus, 01IS12051). **Competing interests**: The authors declare that there are no competing interests.

Data Availability

The LF programming framework is an open source with a Berkeley Software Distribution (BSD) license and can be downloaded from https://lf-lang.org. The programs given in this paper can be downloaded from https://cal.lf-lang.org/.

References

1. Brewer E. Towards robust distributed system. Paper presented at: Symposium on Principles of Distributed Computing (PODC), Keynote talk; 2000 Jul 19; Portland, OR.
2. Brewer E. CAP twelve years later: How the "rules" have changed. *IEEE Computer*. 45(2):23–29.
3. Brewer E. Spanner, TrueTime & the CAP theorem. Google, Report. 14 Feb 2017. [accessed 20 Sep 2022] https://storage.googleapis.com/pub-tools-public-publication-data/pdf/45855.pdf
4. Lee EA, Bateni S, Lin S, Lohstroh M, Menard C. Quantifying and generalizing the CAP theorem. arXiv. 2021. https://arxiv.org/abs/2109.07771.
5. Lo Bello L, Steiner W. A perspective on IEEE time-sensitive networking for industrial communication and automation systems. *Proc IEEE*. 2019;107(6):1094–1120.
6. Zhang B, Jin X, Ratnasamy S, Wawrzynek J, Lee EA. AWStream: Adaptive wide-area streaming analytics. Paper presented at: Proceedings of SIGCOMM, ACM; 2018 Aug 20; Budapest, Hungary.
7. Lohstroh M, Menard C, Bateni S, Lee EA. Toward a lingua franca for deterministic concurrent systems. *ACM Trans Embed Comput Syst*. 2021;20(4):Article 36.

8. Kuhn R. *Reactive design patterns*. Shelter Island (NY): Manning Publications Co.; 2017.
9. Gilbert S, Lynch N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*. 2002;33(2):51–59.
10. Lynch NA. *Distributed algorithms*. San Francisco (CA):Morgan Kaufmann; 1996.
11. Abadi D. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer*. 2012;45(2):37–42.
12. Yu H, Vahdat A. The costs and limits of availability for replicated services. *ACM Trans Comput Syst*. 2006;2(24):70–113.
13. Maler O, Manna Z, Pnueli A. From timed to hybrid systems. In: de Bakker JW, Huizing C, de Roever WP, Rozenberg G, editors. *Real-time: Theory and practice, REX Workshop*. Berlin, Heidelberg (Germany): Springer-Verlag; 1992. p. 447–484.
14. Cataldo A, Lee EA, Liu X, Matsikoudis E, Zheng H. A constructive fixed-point theorem and the feedback semantics of timed systems. Paper presented at: Workshop on Discrete Event Systems (WODES); 2006 July 10–12; Ann Arbor, Michigan.
15. Cremona F, Lohstroh M, Broman D, Lee EA, Masin M, Tripakis S. Hybrid co-simulation: It's about time. *Softw Syst Model*. 2017;18:1655–1679.
16. Lee EA, Sangiovanni-Vincentelli A. A framework for comparing models of computation. *IEEE Trans Comput Aided Des Circuits Syst*. 1998;17(12):1217–1229.
17. Schwarz R, Mattern F. Detecting causal relationships in distributed computations: In search of the holy grail. *Distrib Comput*. 1994;7:149–174.
18. Kleppmann M. A critique of the CAP theorem. arXiv. 2015. https://arxiv.org/abs/1509.05393.
19. Lee EA. *The coevolution: The entwined futures of humans and machines*. Cambridge (MA): MIT Press; 2020.
20. Lamport L, Shostak R, Pease M. Time, clocks, and the ordering of events in a distributed system. *Commun ACM*. 1978;21(7):558–565.
21. Mattern F. Virtual time and global states of distributed systems. In: Cosnard M, Quinton P, Raynal M, Robert Y, editors. *Parallel and distributed algorithms*. Amsterdam (Netherlands): North-Holland; 1988. p. 215–226.
22. Baccelli F, Cohen G, Olster GJ, Quadrat JP. *Synchronization and linearity, an algebra for discrete event systems*. New York (NY): Wiley; 1992.
23. Mills DL. A brief history of NTP time: Memoirs of an internet timekeeper. *ACM Comput Commun Rev*. 2003;33(2):9–21.
24. Mills DL, Mills DL. *Computer network time synchronization — The network time protocol*. Boca Raton (FL): CRC Press; 2006.
25. Eidson JC. *Measurement, control, and communication using IEEE 1588*. New York (NY): Springer; 2006.
26. Eidson JC, Stanton KB. Timing in cyber-physical systems: The last inch problem. Paper presented at: IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS), IEEE; 2015 Oct 11–16; Beijing, China. p. 19–24.
27. Lohstroh M, Romeo ÍÍ, Goens A, Derler P, Castrillon J, Lee EA, Sangiovanni-Vincentelli A, *8th international workshop on model-based design of cyber physical systems (CyPhy'19)*, Springer-Verlag; 2019. Reactors: A deterministic model for composable reactive systems; in press:vol. LNCS 11971.
28. Lohstroh M. Reactors: A deterministic model of concurrent computation for reactive systems [dissertation]. [Berkeley (CA)]: University of California, Berkeley; 2020.
29. Schneider C, Spönemann M, von Hanxleden R. Just model! – Putting automatic synthesis of node-link-diagrams into practice. Paper presented at: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13); 2013 Sep 15–19; San Jose, CA. p. 75–82.
30. Bateni S, Lohstroh M, Wong HS, Tabish R, Kim H, Lin S, Menard C, Liu C, Lee EA. Xronos: Predictable coordination for safety-critical distributed embedded systems. arXiv. 2022. https://arxiv.org/abs/2207.09555.
31. Geng Y, Liu S, Yin Z, Naik A, Prabhakar B, Rosenblum M, Vahdat A. Exploiting a natural network effect for scalable, fine-grained clock synchronization. Paper presented at: USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2018 Apr 9–11; Renton, WA.
32. Alvaro P, Conway N, Hellerstein JM, Marczak WR. Consistency analysis in Bloom: A CALM and collected approach. *5th Biennial Conference on Innovative Data Systems Research (CIDR)*; Jan 2011.
33. Bailis P, Ghodsi A. Eventual consistency today: Limitations, extensions, and beyond. *Commun ACM*. 2013;56(5):55–63.
34. Helland P, Campbell D. Building on quicksand. *Conference on innovative data systems research (CIDR)*. ACM; Jan. 2009. [Online]. Available: https://arxiv.org/abs/0909.1788.
35. Kuhl F, Weatherly R, Dahmann J. *Creating computer simulation systems: An introduction to the high level architecture*. Upper Saddle River (NJ): Prentice Hall PTR; 1999.
36. Fujimoto R. *Parallel and distributed simulation systems*. Hoboken (NJ): John Wiley and Sons; 2000.
37. Zeigler BP, Praehofer H, Kim TG. *Theory of modeling and simulation*. 2nd. Academic Press; 2000, Discrete event systems (DEVS).
38. Liskov BH, Ladin R. Highly available distributed services and fault-tolerant distributed garbage collection. Paper presented at: Symposium on Principles of distributed computing (PODC), ACM; 1986 Nov. p. 29–39.
39. Chandy KM, Misra J. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans on Softw Eng*. 1979;5(5):440–452.
40. Lamport L. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans Program Lang Syst*. 1984;6(2):254–280.
41. Zhao Y, Lee EA, Liu J. A programming model for time-synchronized distributed real-time systems. Paper presented at: Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE; 2007 Apr 3–6; Bellevue, WA.
42. Corbett JC, Dean J, Epstein M, Fikes A, Frost C, Furman JJ, Ghemawat S, Gubarev A, Heiser C, Hochschild P, et al. Spanner: Google's globally distributed database. *ACM Trans Comput Syst*. 2012;31(3):1–22.
43. Pearl J. *Causality*. Cambridge University Press; 2009.