Consistency vs. Availability in Distributed Cyber-Physical Systems

EDWARD A. LEE, University of California, Berkeley, USA
RAVI AKELLA, DENSO International America, Inc., USA
SOROUSH BATENI, SHAOKAI LIN, and MARTEN LOHSTROH, University of California, Berkeley, USA
CHRISTIAN MENARD, Technische Universität Dresden, Germany

In distributed applications, Brewer's CAP theorem tells us that when networks become partitioned (P), one must give up either consistency (C) or availability (A). Consistency is agreement on the values of shared variables; availability is the ability to respond to reads and writes accessing those shared variables. Availability is a real-time property whereas consistency is a logical property. We extend consistency and availability to refer to cyber-physical properties such as the state of the physical system and delays in actuation. We have further extended the CAP theorem to relate quantitative measures of these two properties to quantitative measures of communication and computation latency (L), obtaining a relation called the CAL theorem that is linear in a max-plus algebra. This paper shows how to use the CAL theorem in various ways to help design cyber-physical systems. We develop a methodology for systematically trading off availability and consistency in application-specific ways and to guide the system designer when putting functionality in end devices, in edge computers, or in the cloud. We build on the Lingua Franca coordination language to provide system designers with concrete analysis and design tools to make the required tradeoffs in deployable embedded software.

CCS Concepts: • **Computing methodologies** → **Distributed programming languages**; • **Computer systems organization** → **Embedded and cyber-physical systems;**

Additional Key Words and Phrases: Coordination, concurrency, consistency, availability

ACM Reference format:
Edward A. Lee, Ravi Akella, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard. 2023. Consistency vs. Availability in Distributed Cyber-Physical Systems. *ACM Trans. Embedd. Comput. Syst.* 22, 5s, Article 138 (September 2023), 24 pages.
https://doi.org/10.1145/3609119

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Embedded Software (EMSOFT), 2023.
The work in this paper was supported in part by the National Science Foundation (NSF), awards #CNS-1836601 (Reconciling Safety with the Internet) and #CNS-2233769 (Consistency vs. Availability in Cyber-Physical Systems) and the iCyPhy Research Center (Industrial Cyber-Physical Systems), supported by Denso, Siemens, and Toyota. This work was also supported in part by the German Federal Ministry of Education and Research in the Software Campus program (01IS12051) and the "Souverän. Digital. Vernetzt" program in the joint project 6G-life (16KISK001K)..
Authors' addresses: E. A. Lee, S. Bateni, S. Lin, and M. Lohstroh, University of California, Berkeley, 545Q Cory Hall, Berkeley, CA, 94720-1770, USA; email: eal@berkeley.edu, soroosh129@gmail.com, shaokai@berkeley.edu, marten@berkeley.edu; R. Akella, DENSO International America, Inc., 101 Metro Dr, STE 760, San Jose, CA, 95110, USA; email: ravi.akella@na.denso.com; C. Menard, Technische Universität Dresden, Chair for Compiler Construction, cfaed – Center for Advancing Electronics Dresden, Dresden, 01062, Germany; email: christian.menard@tu-dresden.de.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2023 Copyright held by the owner/author(s).
1539-9087/2023/09-ART138 $15.00
https://doi.org/10.1145/3609119

1 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) [6, 7]. 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. Gilbert and Lynch [14] prove two variants of this theorem, one strong result for asynchronous networks [33, chapter 8] and one weaker result for partially synchronous networks. The CAP theorem has helped the research and development of distributed database systems by clarifying a fundamental limit and suggesting application-dependent tradeoffs. For some database applications, availability is more important than consistency, while for others it is the other way around. The purpose of this paper is to apply and adapt the CAP theorem to distributed cyber-physical systems (CPSs) to derive similar benefits.

In distributed CPSs, the state information shared between software components is often information about the physical world in which the software is operating. In autonomous vehicles, for example, the state of an intersection is shared among all the vehicles contending for access to that intersection. Even within a single vehicle, where software components may be distributed across an onboard network, many of these software components will share state information about the vehicle and its environment derived from sensor input. We therefore generalize the notion of consistency to include physical state rather than just variables in software.

In cyber-physical systems, the time it takes for software to respond through an actuator to stimulus from a sensor is a critical property. We therefore generalize the notion of availability to include this time, not just the system's response time to human users. A software subsystem where sensor-to-actuator response time is large is less available than one for which it is small.

Brewer's CAP theorem, then, immediately applies in an obvious way. When the network becomes partitioned, one of availability or consistency must be sacrificed. However, network partitioning is just the limiting case of network latency, as pointed out by Brewer [7] and Abadi [1]. Moreover, network latency is not the only latency that can force this compromise. Long execution times can be just as damaging as long network latencies, and so can large clock synchronization discrepancies, as we will show. In our formulation, network latencies, execution times, and clock synchronization errors get lumped together into a single measurable quantity that we call, simply, "latency" (or "apparent latency" to emphasize that the quantity we work with is measurable).

We have recently discovered that consistency, availability, and latency can all be quantified, and that they have a simple algebraic relationship between them [24]. We call this relationship the 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. To make this paper self contained, we re-derive the CAL theorem here (with a slightly improved notation). Our main contribution here is to adapt and apply these concepts to CPS, rather than to distributed databases as done in the previous work.

How to trade off consistency, availability, and latency against one another is application specific. Consider for example a four-way intersection, access to which is regulated by a distributed algorithm running in software on autonomous vehicles that contend for the intersection. For this application, and specifically for the state of the intersection, consistency is paramount. All vehicles *must* agree on the state of the intersection (strong consistency) before any one vehicle can enter the intersection. Hence, for this application, when latencies get large for any reason, we choose to sacrifice availability (vehicles do not enter the intersection) rather than consistency (vehicles crash).

Consider, however, a complementary application. Suppose a vehicle has a computer-vision-based automatic braking system as part of an ADAS (advanced driver assistance system) as well as an ordinary brake pedal. Suppose the vision-based system has significant latency (it may even be

computed in the cloud). Should the system delay responses to pushes of the brake pedal until the vision system has reported the state of the world at the time of the brake pedal push? The answer is most certainly "no." The system should respond immediately to brake pedal pushes, thereby maintaining high availability, even at the cost of consistency.

The CAL theorem also easily accommodates tiered, heterogeneous networks, where end devices may connect to edge computers over wired or wireless links, and edge computers 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) [28] on a factory floor, for example, may yield reliable latencies on the scale of microseconds between edge computers, whereas wide-area networks (WAN) may yield highly variable latencies that can extend up to tens of seconds [49]. Moreover, any of these networks can fail, yielding unbounded latencies, and systems need to be designed to handle such failures.

The CAL theorem will allow us to model a heterogeneous network topology interconnecting a wide variety of nodes. In particular, the matrix form of the equations enables compact modeling of heterogeneous networks, where the latencies between pairs of nodes can vary considerably.

We use the Lingua Franca (LF) coordination language [32] to specify programs that explicitly define availability and consistency requirements for a distributed CPS 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.

System designers can use the CAL theorem in at least two complementary ways. They can derive networking requirements from availability and/or consistency requirements, or they can derive availability and/or consistency properties from assumptions about the network behavior.

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. Moreover, by quantifying consistency and availability, the CAL theorem makes the concept applicable to real-time systems. In this paper, we show how to carry out such rigorous design using LF, which supports explicit representations of availability and consistency requirements. Moreover, we demonstrate 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 or has excessive latency. We describe how LF can provide exception handlers that enable the designer to explicitly choose how to handle such fault conditions, for example, by reducing accuracy [49] or by switching to failsafe modes of operation.

The contributions of this paper are as follows (Lingua Franca is *not* a contribution):

- We reformulate the CAL theorem using the notation of Lohstroh et al. [31] and extend it to integrate cyber-physical interactions.
- We show how availability, a real-time property of a system, and consistency, a logical property, relate numerically to clock synchronization and latencies introduced by networks and computation.
- We show how deadlines used to specify real-time requirements in cyber-physical systems are availability requirements and therefore are subject to this relation. Specifically, as latencies increase, it becomes impossible to meet deadlines without sacrificing consistency.

- We propose a methodology that allows a system designer to explicitly define availability and consistency requirements using the Lingua Franca coordination language.
- We show how a system designer can choose how to handle runtime violations of these requirements by explicitly choosing whether to further relax consistency or availability and how to provide fault handlers to be invoked when violations are detected.
- We give practical real-time system examples that show that the choice of whether to sacrifice availability or consistency when faults occur is application dependent.

The paper is organized as follows. Section 2 gives an overview of related work. Section 3 explains the underlying model of time. Section 4 formally defines the terms and derives the CAL theorem. Section 5 introduces the Lingua Franca coordination language, shows how it can explicitly specify availability and consistency requirement, and shows how to use the CAL theorem to analyze a program. Section 6 gives two practical distributed real-time system examples and shows how one needs to prioritize availability while the other needs to prioritize consistency in the presence of faults. Section 8 draws some conclusions.

2 RELATED WORK

We are aware of two prior attempts to quantify the CAP theorem. The first quantifies availability and consistency and shows a tradeoff between them [47]. The quantifications, however, are in terms of fractions of satisfied accesses (availability) and fractions of out-of-order writes (inconsistency), and Yu and Vahdat 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. In the second, Rahman et al. [40] derive a result similar to our CAL theorem for the special case of two nodes communicating and having perfectly synchronized clocks (their theorem 2.6). Our work generalizes this result with arbitrary heterogeneous network topologies with imperfect clocks, a formulation as a linear equation in a max-plus algebra, and its application to CPSs.

Our formulation requires separating the notion of logical time from that of physical time. Separation of these two has appeared many times before in software technologies focused on embedded systems, including the synchronous languages [5], the notion of a logical execution time (LET) [16, 17], and various programing languages and frameworks [13, 32, 39]. Even with this history, the separation remains a difficult concept. Natural language does not easily accommodate sentences that refer to two distinct timelines. The intuitive notion of Newtonian time, which is baked in to natural language, proves to be an inaccurate model of physical time [38, 42]. When dealing with relatively slow Lipschitz continuous phenomena, the Newtonian model often proves adequate. But when dealing with discrete events, where the order in which events occur can have enormous effects on the outcome, the Newtonian model breaks down and proves inadequate.

There is a great deal of related work on distributed real-time systems and real-time programming languages—too much to cite here. To our knowledge, however, none of these give a clear and quantified tradeoff between availability and consistency as a function of network latency using separated notions of logical and physical time.

3 LOGICAL AND PHYSICAL TIME

Central to our ability to quantify both consistency and availability is the distinction between **logical time** and **physical time**. Our notation here follows Lohstroh et al. [31].

Let $\mathbb{T}$ be the set of **physical time values** that any clock in the system of interest may return. For example, elements of $\mathbb{T}$ may represent Coordinated Universal Time (UTC). We assume all clocks are imperfect, and that, because we are interested in distributed systems, there is typically more

than one clock. We assume that $\mathbb{T}$ is a totally ordered set and that $\mathbb{T}$ is an abelian group, and hence we can add or subtract its elements from one another to obtain another element in $\mathbb{T}$.

In Lingua Franca (LF), which we will use to specify our CPSs, $\mathbb{T}$ is a set of 64-bit integers that follow the POSIX standard, so that $T \in \mathbb{T}$ is the number of nanoseconds that have elapsed since midnight, January 1, 1970, Greenwich mean time. As a practical matter, these numbers will overflow in systems running near the year 2270.

Let $\mathbb{G}$ be another totally ordered set of **tags** [26] that label events in the system. From the perspective of any component of a distributed system, the order in which events occur is defined by the order of their tags. Two distinct events with the same tag are **logically simultaneous**.

In physical time, there is no well-founded notion of simultaneity [38], and ambiguity about the order of two spatially-separated events can lead to significant disagreements. Consider, for example, a bank account where a deposit occurs in London nearly simultaneously with a withdrawal in Singapore. Did the account go into overdraft? One way to resolve this question that eliminates ambiguity is to timestamp the events using a local clock at each location, and to *define* the order of the transactions to be the numerical order of their timestamps. This takes an imperfect measure of physical time, itself an ambiguous concept, and elevates it to a logical property on which there can be no disagreement. Once the events have timestamps, all observers who can see both events will agree on the order of the *timestamps* (now a logical time), even if they cannot agree on the order in which the transactions actually occurred.

Our tags provide a general way to resolve such ambiguities. The order of tagged events will be *defined* to be the numerical order of their tags. Agreement about the value of a shared quantity (such as a bank account balance) becomes agreement about the value *at a tag*, not at a *time*. The latter is impossible to achieve, while the former, though not easy, is possible. *Consistency*, therefore, becomes easy to define, resolving the considerable confusion and disagreement about meaning of this term [15]. Consistency is agreement about the value of shared quantity *at a tag*.

To *quantify* consistency and availability, we need to establish a relationship between the tags of events and time values. We assume a monotonically nondecreasing function $\mathcal{T} : \mathbb{G} \to \mathbb{T}$ that gives a physical time interpretation to any tag. For any tag g, we call $\mathcal{T}(g)$ its **timestamp**. A timestamp is a **logical time** value drawn from the same set $\mathbb{T}$ as the **physical time** values reported by clocks. Since the set $\mathbb{T}$ is totally ordered, we can compare logical and physical times.

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 [8, 11, 34]. The order relation in $\mathbb{G}$ is the dictionary order, where $(t_1, m_1) < (t_2, m_2)$ if $t_1 < t_2$ or $t_1 = t_2$ and $m_1 < m_2$. And $\mathcal{T}$ is defined such that for any tag $g = (t, m) \in \mathbb{G}$, $\mathcal{T}(g) = t$. Hence, to get a timestamp from a tag, you just ignore the microstep.

An external input, such as a user input or sensor reading, 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 software system. In LF, this tag is normally given microstep 0, $g = (T, 0)$. Once the tagged event enters the system, its relationship to physical time becomes incidental. The only semantic requirement is that software components process events in tag order, irrespective of physical time, although we will see that the relationship between a timestamp (a logical time) of an event and the physical time at which that event is processed represents availability.

We assume sets $\mathbb{G}$ and $\mathbb{T}$ each have largest element that we will designate $\infty_{\mathbb{G}}$ and $\infty_{\mathbb{T}}$ and a smallest element $-\infty_{\mathbb{G}}$ and $-\infty_{\mathbb{T}}$. We require that $\mathcal{T}(\infty_{\mathbb{G}}) = \infty_{\mathbb{T}}$.

In summary, the tag of an event determines when it will be seen by software components relative to other events, the timestamp represents the logical time of that event, and physical times are time values read from a clock. We will consistently denote tags with a lower case $g \in \mathbb{G}$ or a lower-case tuple $(t, m) \in \mathbb{G}$ and measurements of physical time $T \in \mathbb{T}$ (clock values) with upper case.

We will use the tags g to specify consistency requirements. Relaxing consistency requirements to improve availability is accomplished with **after** delays in Lingua Franca, which add a constant d to the tag of an event. The source of the event and the destination disagree on values carried by the event during a logical interval specified by d. Availability requirements will be specified using the Lingua Franca **deadline** construct. Deadlines in LF are a bit subtle because they span two timelines. A deadline $D \in \mathbb{T}$ is a declaration that the event with tag g must be processed before physical time T exceeds $\mathcal{T}(g) + D$. We will use these deadlines to specify availability requirements.

4 THE CAL THEOREM

Following Schwartz and Mattern [44], 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; a program is formally a family of traces.

In Schwartz and Mattern's model, the k-th event of process i is associated with a tag $g_{i,k}$ (which has a logical time $t_{i,k} = \mathcal{T}(g_{i,k})$) and a physical time $T_{i,k}$. The physical time $T_{i,k}$ is the reading on a local clock at the time when the event starts being processed. We require that events in a process have nondecreasing tags and increasing physical times. That is, if $g_{i,k}$ is the tag and $T_{i,k}$ is the physical time of the k-th event on process i, then $g_{i,k} \leq g_{i,k+1}$ and $T_{i,k} < T_{i,k+1}$. For notational simplicity, when we don't care about the index k of a tag on process i, we will denote the tag by a generic g_i, using only a single subscript. i.e., g_i is the tag of *some* (unspecified) event on process i. If we also don't care about which process the tag is associated with, we omit the subscript altogether, writing just g.

Schwartz and Mattern focus on distributed database systems, not CPSs. Their model has four types of events, read, write, accept, and send. We assume each event has a tag, and we use a superscript to identify the type of event, g^r, g^w, g^a, or g^s respectively. In their model, within each process, a **read event** with tag g^r 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 **write event** in the same process with tag $g^w \leq g^r$ or an **accept event** in the same process with tag $g^a \leq g^r$. An accept event receives an updated value of the variable from the network. A read event with tag g^r will yield the value assigned by the write or accept event with the largest tag g' where $g' < g^r$. If $g' = g^r$, we require that $T' < T^r$, where T' is the physical time of the write or accept event and T^r is the physical time of the read event.

A **send event** is where a process launches into the network an update to a shared variable x. (Here, the "network" is whatever medium is being used for communication between processes.) If a send event with tag g_j^s on process j results in a corresponding accept event with tag g_i^a on process i, we require that $g_i^a \geq g_j^s$. This ensures that timestamps cannot decrease during network transport. 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.

We require that write and accept events in a process i with the same tag as a read or send event in i precede the read or send event in the process. That is, if $g_{i,k}^w = g_{i,k'}^r$, $g_{i,k}^a = g_{i,k'}^r$, $g_{i,k}^w = g_{i,k'}^s$, or $g_{i,k}^a = g_{i,k'}^s$, then $k < k'$. This ensures that a read (or send) event uses a value that was written at an earlier physical time, and that if a read (or send) and write (or accept) are logically simultaneous, the write (or accept) occurs before the read (or send). In addition, to ensure consistency, if two write (or accept) events in a process are logically simultaneous, then they must have a well-defined order in physical time. This property is enforced at the language level in Lingua Franca, but some other framework could simply disallow logically simultaneous writes and accepts for the same shared variable.

In a cyber-physical system, suppose that process j reads a sensor value at physical time T_j (according to its local clock). It writes the sensor value to variable x via a write event with tag $g^w_{j,k}$ such that $\mathcal{T}(g^w_{j,k}) = T_j$. The tag of the event now has a timestamp equal to the clock reading at the time the sensor is read. Suppose process j then generates a send event with tag $g^s_{j,k'} \geq g^w_{j,k}$ for some $k' > k$. The timestamp that is sent to process i along with the sensor reading is no less than the physical time of the sensor reading. At process i, there will be a corresponding accept event with tag $g^a_{i,h} \geq g^s_{j,k'}$ for some h. If $g^a_{i,h} = g^w_{j,k}$, then we say that x is *strongly consistent* across processes i and j. The sensor data have the same timestamp (and tag) on both processes, and that timestamp represents the physical time (at process j) at which the sensor was read. If $g^a_{i,h} > g^w_{j,k}$, then we have a measured inconsistency. We will see that allowing such inconsistency can improve availability.

4.1 Consistency

Definition 1. For each write event to a shared variable x on process j with tag g^w_j, if there is a send event on j that sends this updated value to process i, let g^a_i be the tag of a corresponding accept event on process i. If there is no corresponding accept event, let $g^a_i = \infty_{\mathbb{G}}$. The **inconsistency** $\bar{C}_{ij} \in \mathbb{T}$ from j to i w.r.t. x is defined to be

$$\bar{C}_{ij} = \max\left(\mathcal{T}\left(g^a_i\right) - \mathcal{T}\left(g^w_j\right)\right), \tag{1}$$

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

Because of the constraints on tags of events transported over the network, it is clear that $\bar{C}_{ij} \geq 0$. If $\bar{C}_{ij} = 0$, we have **strong consistency** between processes i and j for variable x. We will see that this strong consistency comes at a price in availability, and that network failures can result in unbounded unavailability. If $\bar{C}_{ij}$ is finite, we have a quantified **eventual consistency**. $\bar{C}_{ij}$ is infinite if updates to x at i are not sent to (or received by) j.

Notice that inconsistency measures the difference between two *timestamps* (logical times). We will show how Lingua Franca enables manipulation of the tags of events to relax consistency requirements in order to gain availability. It does so without sacrificing determinacy.

Discussion. The notion of consistency has many confusingly interrelated conceptualizations and ambiguous definitions in the literature [15]. By defining (in)consistency in terms of logical time rather than physical time, we offer an unambiguous definition that we believe captures the intent. A naive way to use physical time to define consistency might go something like this: "All parts of a distributed system agree on the value of a shared variable at all instants t in (physical) time." There are many problems with this definition. First, there is no shared t [38, 42]. Theoretically, relativity tells us that the order in which events occur in a distributed system depends on the frame of reference. For relatively slowly varying continuous quantities, small discrepancies in the order of events may not matter, but for systems with discrete behaviors, even tiny discrepancies in order may matter a lot. Consider for example the door of a commercial aircraft. It matters a great deal whether the door is disarmed first or opened first. Even a tiny disagreement about the order of these events could result in the emergency escape slides being inadvertently deployed.

Relativity aside, from a practical perspective, clocks cannot be perfectly synchronized, so any distributed system that uses more than one clock to measure physical time cannot know any *true* order of events. It can only compare clock readings. Our formulation recognizes that these clock readings become logical properties, not some physical ground truth. Consistency becomes a *semantic* property of programs rather than a vague and approximate notion of agreement.

4.2 Availability

In database systems, unavailability, $\bar{A}$, is a measure of the time it takes for a system to respond to user requests [19]. We generalize this notion to any reading of a shared variable x and particularly focus on situations where a read of a shared variable is required to drive an actuator at the interface between the cyber and the physical in a CPS. In this situation, unavailability translates into a delay in actuation, which can be particularly problematic in feedback control systems. A read event for shared variable x with tag g^r may be delayed because of network latencies and/or to ensure that all write and accept events with earlier or equal tags have been processed. We call this delay "unavailability."

To measure unavailability, we compare the physical time on process i at which an update (write or accept) event is processed against the timestamp of the event. This measure is particularly useful for CPS because that timestamp typically represents the physical time at which a sensor measurement is taken, and the physical time at which the update occurs gives the earliest time at which that measurement can be read. This motivates the following definition:

Definition 2. For each update (write or accept) event of variable x on process i, let g_i^u be its tag and T_i^u be the physical time at which it is processed. The **unavailability** $\bar{A}_i \in \mathbb{T}$ of x at process i w.r.t. x is defined to be

$$\bar{A}_i = \max\left(T_i^u - \mathcal{T}\left(g_i^u\right)\right), \tag{2}$$

where the maximization is over all update events of x on process i. If there are no such update events on process i, then we define $\bar{A}_i = -\infty_{\mathbb{T}}$.

Note that there is no fundamental constraint that $\bar{A}_i \geq 0$. Even in a CPS, it may be possible to process an update event with timestamp $\mathcal{T}(g_i^u)$ at an earlier physical time T_i^u according to the local clock, i.e., $T_i^u < \mathcal{T}(g_i^u)$. This could occur, for example, if a sensor reading occurs on a remote node j whose physical clock is poorly synchronized with the local node i.

From this definition, we see that unavailability of x at process i may be dominated either by local write events on i or by accept events that are receiving updates over a network.

4.3 Processing Offsets

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 [22, 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 and e_2 is the acceptance of the value in another process. 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 [21], 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 [35]).

To maintain causal consistency, the requirement that a process have nondecreasing tags means that, in a trace, a write event to a shared variable x may have to be processed at a physical time T^w that is significantly larger than its tag's timestamp $\mathcal{T}(g^w)$. In a CPS system, the timestamp $\mathcal{T}(g^w)$ for such an event typically represents the physical time at which the sensor is read, i.e., the reading of a physical clock at the time the external input appears. But an event that writes that sensor value to x may have to be processed at a physical time later than this timestamp to ensure that all events with earlier tags have been processed. This motivates the following definition:

Definition 3. For process i, the **processing offset** $O_i \in \mathbb{T}$ for a shared variable x is

$$O_i = \max\left(T_i^w - \mathcal{T}\left(g_i^w\right)\right) \tag{3}$$

where T_i^w and g_i^w are the physical time and tag, respectively, of a write event to x on process i. The maximization is over all such write events in process i. If there are no such write events, then $O_i = -\infty_{\mathbb{T}}$.

When the variable x stores a sensor reading, and the timestamp $\mathcal{T}(g_i^w)$ of the write event that updates x is drawn from the same local clock that provides T_i^w (the physical time of the write to x), then $T_i^w \geq \mathcal{T}(g_i^w)$ and hence the processing offset is greater than or equal to zero, in this case. But, in general, it can be negative.

4.4 Apparent Latency

When a write to a shared variable x occurs in process j and this update is sent to process i, some physical time will elapse before a corresponding accept event on process i is processed. This motivates the following definition:

Definition 4. Let g_j^w be the tag of a write event to x in process j that is (eventually) sent to and accepted by process $i \neq j$. Let T_i^a be the physical time of the corresponding accept event in process $i \neq j$ (or $\infty_{\mathbb{T}}$ if there is no such event). The **apparent latency** or just **latency** $\mathcal{L}_{ij} \in \mathbb{T}$ for communication from j to i is

$$\mathcal{L}_{ij} = \max\left(T_i^a - \mathcal{T}\left(g_j^w\right)\right), \tag{4}$$

where maximization is over all such write events in process j. If there are no such write events, then $\mathcal{L}_{ij} = -\infty_{\mathbb{T}}$. When $i = j$, we define $\mathcal{L}_{ij} = O_j$, the processing offset at j.

Note that T_i^a and $\mathcal{T}(g_j^w)$ are times that may be derived from *two different clocks*, 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 always appear summed together in our formalism, so there is need to individually measure them. In practice, they can quite difficult to measure individually. Together, however, the apparent latency is easy to measure.

The clock synchronization error E_{ij} can be positive or negative, whereas the processing offset O_j is typically non-negative and 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.

4.5 The CAL Theorem

The above definitions lead immediately to the following theorem:

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

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

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

Proof. The definition of unavailability will be dominated by either a write event, in which case we get the first term of the maximization, or by an accept event, in which case we get the second. □

This can be put in an elegant form using max-plus algebra [3]. Let N be the number of processes, and define an $N \times N$ matrix Γ such that its elements are given by

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

That is, from (5), the i, j-th entry in the matrix is a bound on $X_{ij} + L_{ij} + E_{ij}$ (execution time, network latency, and clock synchronization error), adjusted downwards by the inconsistency $\bar{C}_{ij}$.

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

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

where the matrix multiplication is in the max-plus algebra. This can be rewritten as

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

where $\boldsymbol{I}$ is the identity matrix in max-plus, which has zeros along the diagonal and $-\infty_{\mathbb{T}}$ 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.

4.6 Evaluating 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. These time delays are a consequence of the requirement that events be processed in tag order. Specifically, node i can begin handling a user input (a write event) with tag g_i^w at physical time $T_i^w = \mathcal{T}(g_i^w) + O_i$, which is the earliest time at which the runtime system is sure that all reads and writes of the same variable with earlier tags have already been processed. In the absence of any further information about a program, we can use Γ to calculate these offsets. However, the result is conservative because it does not use dependency information that may be present in a program (and is present in the Lingua Franca programs we give in the next section). A less conservative technique is explained below in Section 6.2.

First, in the current implementation of LF, by default, logical time "chases" physical time, meaning that logical time never gets ahead of physical time. To model this, define a zero column vector $\boldsymbol{Z}$ where every element is zero. With this, we require at least that $\boldsymbol{O} \geq \boldsymbol{Z}$. 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, $O_i \geq \max_j(\mathcal{L}_{ij} - \bar{C}_{ij})$. The smallest processing offsets that satisfy these two constraints satisfy

$$\boldsymbol{O} = \boldsymbol{Z} \oplus \Gamma \boldsymbol{O}. \tag{10}$$

This is a system of equations in the max-plus algebra. From Baccelli et al. [3] (Theorem 3.17), if every cycle of the matrix Γ has weight less than zero, then the unique solution of this equation is

$$\boldsymbol{O} = \Gamma^* \boldsymbol{Z}, \tag{11}$$

where the **Kleene star** is $\Gamma^* = \boldsymbol{I} \oplus \Gamma \oplus \Gamma^2 \oplus \cdots$. Baccelli et al. (Theorem 3.20 [3]) show that this reduces to

$$\Gamma^* = \mathbf{I} \oplus \Gamma \oplus \cdots \oplus \Gamma^{N-1}, \tag{12}$$

where N is the number of processes.

The requirement that the cycle weights be less than zero is intuitive, but overly restrictive. We will show that it means that along any communication path from a node i back to itself, we have to tolerate a non-zero inconsistency somewhere on the cycle.

In practice, programs may have zero or positive cycle means. Theorem 3.17 of Baccelli et al. [3] shows that if all cycle weights are non-positive, then there is a solution, but the solution may not be unique. If there are cycles with positive cycle weights, there is no finite solution for $\boldsymbol{O}$ in (10). In this case, the only solution to (10) sets all the processing offsets to $\infty_{\mathbb{T}}$. Every node must wait forever before handling any user input. This is, of course, the ultimate price in availability.

Equation (11) is conservative because, absent 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. For particular applications, it is possible to use the detailed structure of the Lingua Franca program to derive processing offsets that are not conservative, as we do below in Section 6.2.

5 AVAILABILITY AND CONSISTENCY IN LF

In this section, we briefly introduce Lingua Franca and show how it expresses consistency and availability requirements. We then discuss how processing offsets can be determined without the conservative approximation of Section 4.6.

5.1 Brief Introduction to Lingua Franca

Lingua Franca (or LF, for short) [32] 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). In LF, applications are defined as concurrent compositions of components called **reactors** [29, 30]. LF borrows the best semantic features of established models of computation, such as actors [2], logical execution time (LET) [18], synchronous reactive languages [5], and discrete event systems [25] including DEVS [48] and SystemC [27]. LF programs are concurrent and deterministic [23] (except when fault handlers are invoked). Given any set of tagged input events, there is exactly one correct behavior.

Figure 1 gives a simple example that we use to explain the structure of an LF program and how it specifies availability and consistency requirements. This program defines a simple pipeline consisting of a data source, a data processor, and a data sink. The data source could, for example, poll a sensor and filter its readings. The data processor could use the sensor data to calculate a command to send to an actuator. The data sink could drive the actuator.

The diagram in Figure 2 is automatically generated by the LF tools given the source code.[1] In later examples, we will show the diagram only and not the source code because the diagram contains sufficient information. The chevrons in the diagram represent **reactions**, which process events, and their dependencies on inputs and their ability to produce outputs is shown using dashed lines.

Line 1 in the source code defines the target language, which is the language in which reactions are written, and the language into which the entire LF program is translated. This example uses the C target, which means that the bodies of reactions are written in C.

Line 2 declares a **reactor class** named Sense, which has an **output port** (line 3), a **timer** (line 4), a **state variable** (line 5), and a **reaction** (line 6). The output port has name out and type int. The timer has name t, offset 0 (it should start when the program starts), and period 10ms. The state variable has a name, type, and initial value. Each of these properties of the reactor is represented in the diagram in Figure 2.

```
1  target C;
2  reactor Sense {
3    output out:int;
4    timer t(0, 10ms);
5    state my_state:int(0);
6    reaction(t) -> out {=
7      // code in C: produce out
8    =}
9  }
10 reactor Actuate {
11   input in:int;
12   reaction(in) {=
13     // code in C: read in
14   =} deadline(10ms) {=
15     // code in C: handle violations
16   =}
17 }
18 reactor Compute {
19   input in:int;
20   output out:int;
21   reaction(in) -> out {=
22     // code in C: read in, write out
23   =}
24 }
25 main reactor {
26   i1 = new Sense();
27   i2 = new Compute();
28   i3 = new Actuate();
29   i1.out -> i2.in after 10ms;
30   i2.out -> i3.in after 10ms;
31 }
```

Fig. 1. Structure of an LF program for a simple pipeline.

A **reaction**, like that on line 6, is defined with a syntax of the form

reaction(L_1) L_2 -> L_3 {=
code body
=}

where L_1 is a list of **triggers**, which are inputs, timers, and actions (we will discuss actions later); L_2 is an optional list of **observables**, which are inputs and actions that do not trigger the reaction but may be read by the reaction; and L_3 is an optional list of **effects**, which are outputs, actions, and modes (which are not used in this paper). A reaction is logically instantaneous in that effects have the same tag as the triggers and observables that produce them.

The particular reaction on line 6 is triggered by the timer every 10ms. When it is triggered for the n-th time, its logical time will be $t = s + n \times 10$ ms, where s is the logical start time, typically set using the local physical clock when the program starts. The runtime system attempts to align logical time with physical time, so this reaction will be invoked roughly every 10ms, but this cannot be done perfectly. By default, logical time "chases" physical time in a program execution, so that reactions with a logical time t are invoked close to (but never before) physical time $T = t$.

A reaction may optionally have a **deadline**, as shown in the Actuate reactor class on line 14. This gives a time value and a code body

[1]The diagram synthesis feature was created by Alexander Schulz-Rosengarten of Kiel University using the graphical layout tools from the KIELER Lightweight Diagrams framework [43] (see https://rtsys.informatik.uni-kiel.de/kieler).

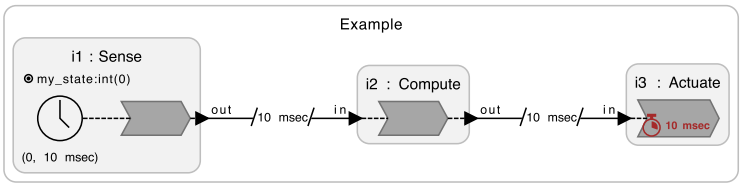

Fig. 2. Diagram for the LF program in Figure 1.

to execute instead of the reaction if the deadline is violated. The time value may be a parameter of the reactor class but here is shown as the constant 10ms. A deadline with time value $d = 10$ms is violated for an event with tag g if the reaction is invoked at a physical time T where $T > \mathcal{T}(g) + d$. Such a deadline explicitly specifies an availability requirement. The deadline violation handler (line 15) is a fault handler. The LF runtime system uses an EDF scheduler to attempt to avoid violating this deadline, as usual in real-time systems.

The top-level (main) reactor is defined on line 25. Within it, reactor instances are created as on line 26 using the **new** keyword. If the **main** keyword is replaced with **federated**, then a separate C program is generated for each reactor instantiated within the federated reactor. Otherwise, a single multi-threaded C program is generated for the entire program. For the federated case, each instance is called a **federate**, and tagged inputs will arrive from the network at the input ports and be handled in tag order.

The routing of messages is specified by **connections**, as shown on line 29, which connects the output of `i1` to the input of `i2`. Such a connection may optionally alter the timestamp of the message using the **after** keyword. The connection on line 29 specifies that the timestamp of the input event at `i2` should be 10ms greater than the timestamp at the output of `i1`. Such a **logical delay** explicitly relaxes the consistency requirements because it explicitly states that `i2` can use information that is 10ms out of date relative to `i1`.

We can see immediately that use of logical delays improves availability for this example, as expected from the CAL theorem. Suppose that this program is federated and that the three instances are mapped to distinct processors on a network. Were it not for the logical delays, intuitively, the `Actuate` reactor `i3` would be unable to react until the message from the `Sense` reactor `i1` had flowed through the network to `i2`, `i2` had completed its reaction, and the result from `i2` had flowed over the network to `i3`. If these physical delays add up to more than 10ms, the `i3`'s deadline will be missed. With the logical delays, however, as long as the physical delays add up to less than 30 ms, the deadline will not be missed. If the physical delays add up to less than 20 ms, then `i3` can react to its input with timestamp t as soon as physical time T matches or exceeds t. The specified tolerance for inconsistency improves availability. This intuition can be made rigorous using the CAL theorem.

5.2 Evaluating Unavailability

For the program in Figure 1, the Γ matrix is given by

$$\Gamma = \begin{bmatrix} 0 & -\infty & -\infty \\ \Gamma_{21} & 0 & -\infty \\ -\infty & \Gamma_{32} & 0 \end{bmatrix} \tag{13}$$

where

- $\Gamma_{21} = X_{21} + L_{21} + E_{21} - 10\text{ms}$,
- $\Gamma_{32} = X_{32} + L_{32} + E_{32} - 10\text{ms}$,

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

The $-\infty$ entries (short for $-\infty_{\mathbb{T}}$) in the matrix are a consequence of a lack of communication.

The communication path from `i1` to `i2` has a logical delay of 10ms, which is an explicit declaration of an inconsistency $\bar{C}_{21} = 10$ ms. The `Compute` reactor's view of the data from the `Sense` reactor is 10ms behind. We can now determine that this allowance of 10ms of inconsistency improves availability compared to what we would get without it. First, we can use the analysis of Section 4.6 to evaluate the processing offsets. There, $N = 3$, so (12) 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 `Actuate` reactor indirectly depends on the `Sense` reactor, something not directly represented in the Γ matrix.

We can now evaluate (11) to get

$$\boldsymbol{O} = \Gamma^* \boldsymbol{Z} = \begin{bmatrix} 0 \\ \max(\Gamma_{21}, 0) \\ \max(\Gamma_{21} + \Gamma_{32}, \Gamma_{32}, 0) \end{bmatrix}. \tag{14}$$

Next, we evaluate (9) to get the unavailability at each node,

$$\boldsymbol{A} = (\boldsymbol{I} \oplus \Gamma)\boldsymbol{O} = \begin{bmatrix} 0 \\ \max(\Gamma_{21}, 0) \\ \max(\Gamma_{21} + \Gamma_{32}, \Gamma_{32}, 0) \end{bmatrix}. \tag{15}$$

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

These unavailability numbers are intuitive. First, note that the `Sense` can react to external stimulus immediately. It has no network inputs to worry about, so $\boldsymbol{A}_0 = 0$. The `Compute` reactor, however, can react to an input stimulus with timestamp t immediately when physical time $T = t$ only if $X_{21} + L_{21} + E_{21} \leq 10$ms. Otherwise, in order to ensure that it processes events in timestamp order, it may need to wait until $T = t + X_{21} + L_{21} + E_{21} - 10$ms. Similarly, the `Actuate` reactor can respond immediately if $\Gamma_{32} \leq 0$ and $\Gamma_{21} + \Gamma_{32} \leq 0$. These conditions occur if the 10ms logical delay is larger than the apparent latencies in communication.

If we change line 29 to this subtly different version:

```
29   i1.out ~>  i2.in;
```

then there is no upper bound on the inconsistency between these two instances. The subtle change is to replace the **logical connection** `->` with a **physical connection** `~>`. In Lingua Franca, this is a directive to assign a new tag g_i^a at the accepting end i based on a local measurement of physical time T_i^a when the message is received such that $\mathcal{T}(g_i^a) = T_i^a$. The original tag is discarded. Such connections, therefore, have no effect on availability, but they completely abandon consistency.

5.3 Processing Offsets in Lingua Franca

Lingua Franca offers two coordination strategies for federated execution, *centralized* and *decentralized* [4]. The centralized coordinator is an extension of the High-Level Architecture (HLA) [12], a distributed discrete-event simulation standard. The decentralized coordinator is an extension of PTIDES [51], a real-time technique also implemented in Google Spanner [10], a globally distributed database. This coordinator is also influenced by Lamport [20] and Chandy and Misra [9, 37].

For the purposes of this paper, we only need to know how these coordination mechanisms relate to processing offsets and availability. The *centralized* coordinator is the easiest to understand. It does whatever is necessary to ensure that events are processed in tag order. In particular, execution in a federate will be delayed when such a delay is needed to ensure tag order semantics. As a consequence, the processing offsets and unavailability bounds emerge from an execution of the program. As network latencies vary, the offsets and unavailability vary. The CAL theorem tells what to expect these numbers to be, given network latencies. The programmer, therefore, can use the CAL theorem to determine whether deadlines will be met, given specific latencies.

The processing offsets and unavailability bounds play a bigger role when using the *decentralized* coordinator. In particular, with this coordinator, the programmer is required to specify a **safe-to-advance** (**STA**) offset for each federate. The choice of STA at federate i can be guided by processing offset $\boldsymbol{O}_i$ for federate i, with the caveat that $\boldsymbol{O}_i$ is a property of a *trace*, whereas STA is a property of a *program* (a family of traces). The STA specified for a federate i **enforces** O_i during execution for all the traces produced by federate i. For particular reactions, the programmer can also give an additional **safe-to-assume-absent** (**STAA**) offset. STAA gives an additional time beyond the STA to wait before assuming that the absence of a message on a particular input means that there will be no message on that port with the current tag or less. The availability bound given by the CAL theorem can similarly guide the choice of STAA. Such guidance is much better than guesswork.

Both coordinators will deterministically execute the LF program identically, yielding the same behavior as an unfederated execution, under certain assumptions. For the centralized coordinator, the assumptions are that the network latencies are sufficiently low that no deadlines are missed. For the decentralized coordinator, the assumptions are that the network latencies are sufficiently low that events are seen by each reactor in tag order. In both cases, violations of the assumptions are detectable and can be handled by fault-handling code provided by the programmer.

The key difference between the two coordinators, therefore, is in their fault handling. When network latencies get large (or the network gets partitioned), the centralized coordinator sacrifices availability, whereas the decentralized coordinator sacrifices consistency. Which of these is the right choice is application dependent.

Note also that for both coordinators, the CAL theorem can be used to derive the requirements on network latencies, and therefore provides a principled guide for choosing networking technology and can guide designers to move computations between embedded, edge, and cloud computing.

5.4 Fault Handling

Any assumptions about network latency may be violated in the field. In centralized coordination, such violations will manifest as deadline violations, whereas in decentralized coordination, they will manifest as consistency violations. In both cases, LF allows the programmer to specify exception handlers to be invoked when such violations occur. Hence, for safety-critical CPS applications, the proposed framework promises some attractive properties. First, a programmer can explicitly decide when and how much to give up consistency and when and how much to give up availability to accommodate execution times, network latency, and clock synchronization error. Second, the

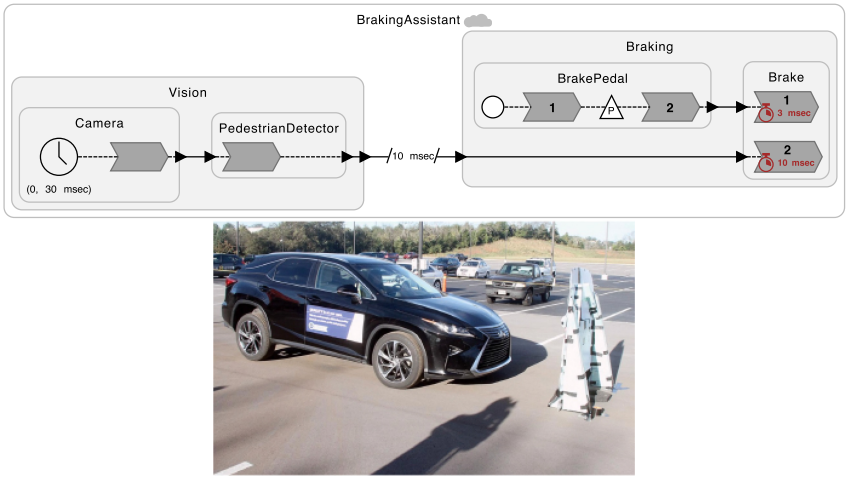

Fig. 3. ADAS system schematic and photo from 2018 demo by Denso, reported in *The Daily Times*.

programmer's specification will imply explicit constraints on the technology (networking, processing, and clock synchronization) that can be used to guide selection of parts to use and mapping of software components onto resources (embedded, edge, or cloud). Third, to allow for (inevitable) possible failures in the field, where specifications are not met due to unforeseen circumstances such as hardware failures, the programmer can explicitly give code to execute when the fault occurs. We show now how these principles can be applied through two complementary practical examples.

6 TRADEOFFS IN PRACTICAL SYSTEMS

In this section, we consider two safety-critical real-time systems with complementary properties. The first demands availability, where timely responses take priority over consistency in the presence of faults. This example also requires a measured relaxation of consistency to meet tight deadlines. The second example demands that consistency be prioritized over availability in the presence of faults and also illustrates how cycles are handled by the CAL theorem.

6.1 ADAS

Consider an Advanced Driver Assist Systems (ADAS), as shown in Figure 3. It has a camera with a computer vision system that analyzes images for pedestrians and applies braking when a pedestrian is detected, as shown in the photo. The figure shows the diagram synthesized from an LF program that shows the structure of this system. In this structure, there are two federated reactors, a `Vision` subsystem and a `Braking` subsystem. The structure is a pipeline similar to that of Figure 1, except with a twist. Inside the `Braking` subsystem is a second sensor, which senses when a driver presses on the brake pedal. Both the camera and the pedal can affect the same actuator, which is driven by the `Brake` reactor.

The `Vision` federate has a time-triggered periodically invoked reaction that captures and analyzes an image. It then sends the results over the network to the `Braking` federate. The `Braking` federate has a local interface to a sensor on the brake pedal, represented in the diagram by the triangle with a "P" (which represents a **physical action** in LF). When the brake pedal is pushed, an event is generated that is assigned a time stamp from the local clock and triggers invocation of the reaction labeled "2" within the `BrakePedal` reactor.

Let the `Vision` federate be process 1 and the `Braking` federate be process 2. Then the Γ matrix is similar to (13):

$$\Gamma = \begin{bmatrix} 0 & -\infty \\ \Gamma_{21} & 0 \end{bmatrix} \tag{16}$$

where

- $\Gamma_{21} = X_{21} + L_{21} + E_{21} - 10\text{ms}$,
- X_{21} is the execution time in `Vision` to prepare the data to send to `Braking`,
- L_{21} is the network latency from `Vision` to `Braking`, and
- E_{21} is the clock synchronization error.

The logical delay of 10ms on the communication path from 1 to 2 is an explicit declaration of an inconsistency $\bar{C}_{21} = 10\text{ms}$. The Braking system's view of the sensor data from the Vision system is 10ms behind. Using the same methods, the processing offsets and unavailability are similar to (14) and (15):

$$\boldsymbol{O} = \begin{bmatrix} 0 \\ \max(\Gamma_{21}, 0) \end{bmatrix} \quad \boldsymbol{A} = \begin{bmatrix} 0 \\ \max(\Gamma_{21}, 0) \end{bmatrix}$$

The allowance of 10ms of inconsistency improves availability compared to what we would get without it. In particular, if $X_{21} + L_{21} + E_{21} \leq 10\text{ms}$, then the processing offsets and unavailability are all zero.

In Figure 3, notice that the first reaction of `Brake` has a deadline of 3 ms. This deadline is as an explicit requirement for availability, stating, effectively, that we require

$$X_{21} + L_{21} + E_{21} - 10\text{ms} + X_{22} \leq 3\text{ms},$$

where X_{22} is the execution time of reaction 2 in `BrakePedal`.[2] The requirement becomes

$$X_{21} + L_{21} + E_{21} < 13\text{ms} - X_{22}. \tag{17}$$

This requirement almost certainly means that the vision processing cannot be done in the cloud. If it is, then the deadline is likely to be violated. In principle, this analysis can be automated, so that a system designer simply enters the requirements (by specifying deadlines, communication paths, and consistency requirements), and the system provides feedback on the realizability of the requirements.

If the system designer really wants to do the vision processing in the cloud, then these results can be used to negotiate a service-level agreement, for example, with the 5G network vendor and the cloud service provider. Alternatively, the 10ms tolerance for inconsistency could be increased, although this would require an evaluation of whether the ADAS system continues to be able to do its job safely. Once the requirements and assumptions are specified, then the next key decision is what to do when those assumptions are violated. For this application, missing the deadline could be disastrous, so we should emphasize availability over consistency. To accomplish that in Lingua Franca, we just have to specify to use decentralized coordination. With this coordination mechanism, if messages fail to arrive on time from the network, each local runtime system assumes there are no messages and continues accordingly. This will ensure that the brake pedal event gets handled as long as the `Braking` federate's host computer is still working.

6.2 Four-Way Intersection

Consider (semi)autonomous vehicles that leverage communication with a roadside unit (RSU) to mediate access to a four way intersection. There are many projects working on such automation

[2]Note that LF implements an EDF scheduling policy, and that deadlines are inherited upstream, so reaction 2 of `BrakePedal` will have high priority.

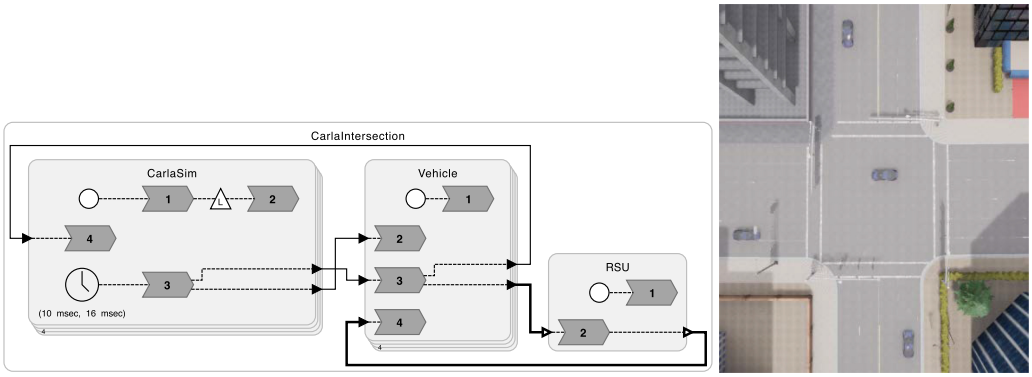

Fig. 4. Four-way intersection example.

to improve traffic flow [41, 45]. A prototype is depicted in Figure 4. This prototype uses a popular open-source vehicle simulator called Carla, which generates the animated image in the figure that gets updated as the program runs. The prototype is implemented using the Python target in LF, which enables easy integration of large legacy subsystems, such as Carla, that have Python APIs.

The LF program depicted in Figure 4 consists of nine top-level components: four vehicle simulators, four vehicle controllers, and one roadside unit. The program uses a compact LF notation for banks of reactors and a multiplicity of communication channels. In this program, as a vehicle approaches the intersection, it communicates with the RSU, sending its kinematic state (position and velocity). The RSU handles competing requests for access to the shared resource, the intersection, by granting time windows to particular vehicles during which they may use the intersection. This application represents a common pattern that occurs whenever distinct agents contend for access to a shared resource.

A key property of this application is that, very much unlike the ADAS example in Section 6.1, consistency is far more important. All vehicles and the RSU *must* have a consistent view of the state of the intersection before *any* vehicle can enter the intersection. In other words, we prefer that vehicles stop (making the intersection *unavailable*) over having them enter the intersection without a consistent view on the state of the system, which could lead to a collision.

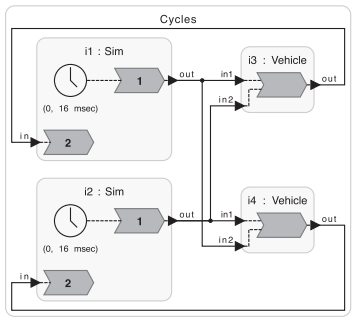

Fig. 5. Simplified program with cycles.

In LF, if we choose centralized coordination for the federated execution, the system will emphasize consistency over availability in the event of faults (violations of the assumptions and requirements). If messages from the network do not arrive on time, each federate stops progressing, which will prevent a vehicle from entering the intersection.

The contrast between the requirements of the intersection and ADAS examples demonstrates that tradeoffs between availability and consistency are application dependent. System designers should be able to make such tradeoffs, and software needs to be designed so it responds to faults in coherence with the stated requirements. For the ADAS example, we need to sacrifice consistency, whereas for the intersection example, we need to sacrifice availability.

A second difference in the intersection vs. ADAS example is that the program has communication cycles without logical delays. This changes how we do the analysis because we will no longer

be able to use (11) to determine the processing offsets. Instead, we have to derive the processing offsets using more detailed information about the program structure. We now show how to do that.

The simpler LF program depicted in Figure 5 has the essential structure of the intersection example reduced to the minimum that illustrates the issues. The Γ matrix is

$$\Gamma = \begin{bmatrix} 0 & -\infty & \Gamma_{13} & -\infty \\ -\infty & 0 & -\infty & \Gamma_{24} \\ \Gamma_{31} & \Gamma_{32} & 0 & -\infty \\ \Gamma_{41} & \Gamma_{42} & -\infty & 0 \end{bmatrix}$$

The finite non-zero entries are defined as before,

$$\Gamma_{ij} = X_{ij} + L_{ij} + E_{ij} - \bar{C}_{ij},$$

where, in this case, $\bar{C}_{ij} = 0$ because none of the connections has a logical delay. To find the processing offsets, we use information that is evident in the LF program but not in the matrix. Specifically, note that any inputs that arrive at the inputs of the `Sim` reactors will have the same tag as an output produced by one of the two `Sim` reactors. Moreover, the two `Sim` reactors' outputs are driven by timers with the same offset and period (zero and 16 ms), so these outputs are logically simultaneous. We now make a key assumption:

Assumption 1. *The period of the timers is greater than any unavailability.*

With this assumption, each `Sim` reactor will have completed processing all events with timestamp t before it needs to advance to logical time $t + p$, where p is the timer period. Recall that we are assuming that logical time chases physical time, so physical time has to advance to $t + p$ before the federate will even attempt to advance its logical time. At this point, it can safely advance its logical time immediately. Hence, the processing offset for both `Sim` reactors is zero if our Assumption 1 is true.

The processing offset for the two `Vehicle` instances is easier to derive. It simply depends on the communication latencies, clock synchronization errors, and execution time bounds. The resulting processing offset vector is

$$\boldsymbol{O} = \begin{bmatrix} 0 \\ 0 \\ \max(\Gamma_{31}, \Gamma_{32}) \\ \max(\Gamma_{41}, \Gamma_{42}) \end{bmatrix} \tag{18}$$

We can now use (9) to calculate the unavailability:

$$\boldsymbol{A} = (\boldsymbol{I} \oplus \Gamma)\boldsymbol{O} = \begin{bmatrix} \Gamma_{13} + \max(\Gamma_{31}, \Gamma_{32}) \\ \Gamma_{24} + \max(\Gamma_{41}, \Gamma_{42}) \\ \max(\Gamma_{31}, \Gamma_{32}) \\ \max(\Gamma_{41}, \Gamma_{42}) \end{bmatrix} \tag{19}$$

We can now see that if the clock period is less than that top two entries in this vector, then Assumption 1 will be violated, so this becomes a requirement.

These results are intuitive and correspond with observation when we run the federated LF program. Assumption 1 asserts that the period of the clocks is large enough that each period begins fresh without an accumulated backlog of unprocessed events. The execution will begin each period by advancing the logical time of each `Sim` federate to the next period as soon as physical time matches that logical time. The zeros in the first two entries of (18) tell us this is done without delay. The logical time of the two `Vehicle` federates, however, cannot be advanced until enough physical

time has elapsed to allow for propagation of events from *both* `Sim` federates. This delay appears as the last two entries in (18).

As shown in (19), the unavailability of the two `Vehicle` federates matches their processing offset. This is not surprising because they each have only one reaction and that reaction reacts to both network inputs. However, the unavailability at the `Sim` reactors is larger than their processing offset. This reflects the fact that reaction 2 in each of the `Sim` reactors has to wait for the upstream `Vehicle` reactors to execute and for their results to propagate over the network. In other words, strong consistency—which lets actuation be logically simultaneous with the acquisition of sensor inputs—comes at a cost in availability. The actuation is delayed in physical time, and, more fundamentally, the period with which sensing and actuation can be done has a lower bound that depends on the network delays.

Under centralized coordination, the actual values of all the Γ_{ij} latencies are determined automatically at runtime as apparent latencies. If the program fails to keep up for any reason (e.g., network failures), then the centralized coordinator will preserve consistency; unavailability will rise and deadlines (if any are specified) will be missed. Fault handlers provided by the programmer can adapt the system accordingly. Moreover, in this case, Assumption 1 becomes invalid, so the derived unavailability bound becomes invalid. Unavailability will exceed our calculated bound for such a trace and, in the event of total network failure, will grow without bound.

Under decentralized coordination, the programmer chooses numbers to the Γ_{ij} latencies based on assumptions about network behavior and derives processing offsets (18) and unavailability (19). These then guide the choices of STA and STAA numbers specified in the program. At runtime, each federate proceeds on the assumption that the network latencies will be respected. If these assumptions are violated, then a reactor may see events out of timestamp order, in which case a fault handler will be invoked. If the network fails altogether, however, no reactor will see events out of timestamp order, and no fault handler will be invoked. Instead, each vehicle will act based on inconsistent information. Hence, with decentralized coordination, availability is prioritized over consistency when a fault occurs, which is the wrong choice for this application.

7 DISCUSSION

7.1 Scientific Principle

The relationship between consistency, availability, and latency is relatively straightforward and the the tradeoffs are already part of the folklore for designers of distributed systems. The CAL theorem, with its rigorous definitions and quantification of the tradeoffs, elevates folklore to a scientific principle that can be used systematically in system design. To our knowledge, this is the first time the notions of "availability" and "consistency" from distributed computing have been applied to CPS, where availability translates into actuation delays and consistency to alignment, and hence the first time that that the fundamental tradeoff between such actuation delays and consistency has been quantified. In other words, we show that meeting tight deadlines in a distributed CPS may *require* relaxing the required agreement about the state of the physical world. Components may have to be allowed to disagree (by a measured amount) in order to meet deadlines, and the deadlines and measure of disagreement can be numerically related to observed latencies due to communication, computation, and clock synchronization errors.

7.2 Usage Patterns

In CPS design, the CAL theorem can form part of the fundamental toolkit available to engineers. One pattern of usage is to specify the availability and consistency requirements of an application, and then use the CAL theorem to derive the networking requirements. This can guide the selection

of networking technology (wireless? fieldbus? 5G? CAN Bus? Ethernet? TSN?) and can help designers determine which components can be put in the cloud and which must reside more locally at the edge. For example, can the image analysis of our ADAS example be done in the cloud? Another pattern of usage is to take the networking technology as a given, specify the availability requirements, and then derive the extent to which consistency must be relaxed given these constraints. A third pattern is to take all three, consistency, availability, and latency, as requirements and use them to assess the likelihood of faults. Faults will appear as either deadline violations (violations of the availability requirements) or as consistency violations (out-of-order events). Fault handlers can then be added to systems so that they fail gracefully.

7.3 Scalability

Although Lingua Franca is not a contribution of this paper, the reader may be concerned about overhead introduced through its use. Menard et al. [36] perform a detailed analysis of the performance of LF and conclude that it is competitive with the state-of-the-art actor frameworks Akka and CAF. Their evaluation, however, is limited to multicore execution; they did not consider distributed systems. Whether Lingua Franca remains competitive for distributed systems remains to be seen. There is no doubt, however, that its ability to directly work with the tradeoffs between availability and consistency is a unique feature.

Using the CAL theorem in practice will involve performing calculations involving primarily maximization and addition. In this paper, we performed the calculations by hand, but such an approach will probably not scale to larger systems. It is not hard to imagine semi-automated tools that would function as part of an integrated development environment (IDE) for Lingua Franca. For example, given a deadline specification in LF, the IDE could highlight the computations and communications that affect the ability to meet the availability requirement implied by the deadline. Even in a very large system with thousands of components, in practice, the subsystem involved for each particular deadline is likely to be much smaller (otherwise, meeting the deadline reliably is probably unrealistic anyway). So even without leveraging sparse matrix techniques, naive direct manipulation of the max-plus formulas seems tractable. For example, directly calculating Γ^* using (12), with no optimization for sparsity, involves performing $N - 1$ matrix multiplications in max-plus (i.e., using only addition and maximization). A brute-force algorithm for matrix multiplication has complexity $O(N^3)$, making the overall complexity of the calculation $O(N^4)$. For $N = 100$ components (an unrealistically large number, in our opinion), this means on the order of $100^4 = 100{,}000{,}000$ max-plus operations. Any modern laptop can deliver such a result in modest time. Recognizing that the N components are unlikely to be fully connected, the vast majority of these operations could be optimized away.

Some usage patterns for the CAL theorem in practice will require estimation of network latencies and computation times for particular system implementations. Rigorous determination of these times can be extremely challenging [46, 50]. Fortunately, the "latency" in the CAL theorem is "apparent latency," a quantity that is trivially easy to measure (it is the difference between a physical time as reported by a local clock and a logical timestamp carried by a message). Hence, a measurement-based approach to obtaining these numbers is easy to deploy.

8 CONCLUSIONS

The CAL theorem, which generalizes Brewer's CAP theorem, quantifies the relationship between inconsistency, unavailability, and apparent latency in distributed systems, where apparent latency includes network latency, execution time overhead, and clock synchronization error. The relationship is a linear system of equations in a max-plus algebra. We have applied this theorem to distributed cyber-physical systems, showing how consistency affects the ability to bound the

time it takes to react to an external stimulus, such as a sensor input, and produce a response, an actuator output. These bounds (which we call unavailability) depend on apparent latency and can be reduced by explicitly relaxing consistency requirements. Moreover, because the CAL theorem defines the effect of network latency on the responsiveness of a system, 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 Lingua Franca coordination language enables arbitrary tradeoffs between consistency and availability as apparent latency varies. We have also shown how LF programs can define fault handlers, sections of code that are executed when specified consistency and availability requirements cannot be met because apparent latency has exceeded the assumed bounds. Because of its deterministic semantics, LF provides predictable and repeatable behaviors in the absence of faults. And when faults occur, LF provides mechanisms for the system to adapt.

An intriguing extension, suggested by Rahman et al. [40] (their theorem 2.7), is to develop a probabilistic model where a fraction of the events meet consistency and availability requirements and the rest fail. The CAL theorem, as given here, looks only at worst case behaviors. We conjecture, however, a similar probabilistic model could benefit from the CAL theorem formulation to generalize to arbitrary networks and imperfectly synchronized clocks. This is left for further work.

ACKNOWLEDGMENTS

The authors thank Enrico Bini and anonymous reviewers for very helpful suggestions on earlier versions of the paper.

REFERENCES

[1] Daniel Abadi. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* 45, 2 (February 2012), 37–42. https://doi.org/10.1109/MC.2012.33

[2] Gul A. Agha. 1997. Abstracting interaction patterns: A programming paradigm for open distributed systems. In *Formal Methods for Open Object-based Distributed Systems, IFIP Transactions*, E. Najm Stefani and J.-B. (Eds.). Chapman and Hall.

[3] F. Baccelli, G. Cohen, G. J. Olster, and J. P. Quadrat. 1992. *Synchronization and Linearity, An Algebra for Discrete Event Systems.* Wiley, New York.

[4] Soroush Bateni, Marten Lohstroh, Hou Seng Wong, Rohan Tabish, Hokeun Kim, Shaokai Lin, Christian Menard, Cong Liu, and Edward A. Lee. 2022. Xronos: Predictable coordination for safety-critical distributed embedded systems. *arXiv:2207.09555 [cs.DC]* (July 2022). https://arxiv.org/abs/2207.09555

[5] Albert Benveniste and Gérard Berry. 1991. The synchronous approach to reactive and real-time systems. *Proc. IEEE* 79, 9 (1991), 1270–1282.

[6] Eric Brewer. 2000. Towards robust distributed system. In *Symposium on Principles of Distributed Computing (PODC).* Keynote talk.

[7] Eric Brewer. 2012. CAP twelve years later: How the "rules" have changed. *IEEE Computer* 45, 2 (February 2012), 23–29. https://doi.org/10.1109/MC.2012.37

[8] Adam Cataldo, Edward A. Lee, Xiaojun Liu, Eleftherios Matsikoudis, and Haiyang Zheng. 2006. A constructive fixed-point theorem and the feedback semantics of timed systems. In *Workshop on Discrete Event Systems (WODES).*

[9] K. Mani Chandy and Jayadev Misra. 1979. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering* 5, 5 (1979), 440–452.

[10] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's globally-distributed database. In *OSDI.* https://doi.org/10.1145/2491245

[11] Fabio Cremona, Marten Lohstroh, David Broman, Edward A. Lee, Michael Masin, and Stavros Tripakis. 2017. Hybrid co-simulation: It's about time. *Software and Systems Modeling* 18 (November 2017), 1655–1679. https://doi.org/10.1007/s10270-017-0633-6

[12] Judith S. Dahmann, Richard M. Fujimoto, and Richard M. Weatherly. 1997. The department of defense high level architecture. In *Proceedings of the 29th Conference on Winter Simulation*. 142–149.
[13] Stephen A. Edwards and John Hui. 2020. The sparse synchronous model. In *Forum on Specification and Design Languages (FDL)*.
[14] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News* (June 2002), 33(2). https://doi.org/10.1145/564585.564601
[15] Pat Helland. 2021. Don't get stuck in the "con" game: Consistency, convergence and confluence are not the same! *Queue* 19, 330 (June 2021), 16–35. https://doi.org/10.1145/3475965.3480470
[16] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. 2003. Giotto: A time-triggered language for embedded programming. *Proceedings of IEEE* 91, 1 (January 2003), 84–99. https://doi.org/10.1109/JPROC.2002.805825
[17] Christoph M. Kirsch and Ana Sokolova. 2012. The logical execution time paradigm. In *Advances in Real-Time Systems*, S. Chakraborty and J. Eberspächer (Eds.). Springer-Verlag, Berlin Heidelberg, 103–120. https://doi.org/10.1007/978-3-642-24349-35
[18] Christoph M. Kirsch and Ana Sokolova. 2012. The logical execution time paradigm. In *Advances in Real-Time Systems*. Springer, 103–120.
[19] Martin Kleppmann. 2015. A Critique of the CAP Theorem. (2015). https://arxiv.org/abs/1509.05393 arXiv:1509.05393 [cs.DC].
[20] Leslie Lamport. 1984. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems* 6, 2 (1984), 254–280.
[21] Leslie Lamport, Robert Shostak, and Marshall Pease. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.
[22] Edward Ashford Lee. 2020. *The Coevolution: The Entwined Futures of Humans and Machines*. MIT Press, Cambridge, MA.
[23] Edward A. Lee. 2021. Determinism. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 5 (July 2021), 1–34. https://doi.org/10.1145/3453652
[24] Edward A. Lee, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard. 2023. Trading off consistency and availability in tiered heterogeneous distributed systems. *Intelligent Computing* 2, Article 0013 (February 15 2023), 1–23. https://doi.org/10.34133/icomputing.0013
[25] Edward A. Lee, Jie Liu, Lukito Muliadi, and Haiyang Zheng. 2014. Discrete-event models. In *System Design, Modeling, and Simulation using Ptolemy II*, Claudius Ptolemaeus (Ed.). Ptolemy.org.
[26] Edward A. Lee and Alberto Sangiovanni-Vincentelli. 1998. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Circuits and Systems* 17, 12 (1998), 1217–1229.
[27] Stan Liao, Steve Tjiang, and Rajesh Gupta. 1997. An efficient implementation of reactivity for modeling hardware in the scenic design environment. In *Design Automation Conference*. ACM, Inc.
[28] Lucia Lo Bello and Wilfried Steiner. 2019. A perspective on IEEE time-sensitive networking for industrial communication and automation systems. *Proc. IEEE* 107, 6 (2019), 1094–1120. https://doi.org/10.1109/JPROC.2019.2905334
[29] Marten Lohstroh. 2020. *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-235.html
[30] Marten Lohstroh, Íñigo Íncer Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. 2019. Reactors: A deterministic model for composable reactive systems. In *8th International Workshop on Model-Based Design of Cyber Physical Systems (CyPhy'19)*, Vol. LNCS 11971. Springer-Verlag. in press.
[31] Marten Lohstroh, Edward A. Lee, Stephen A. Edwards, and David Broman. 2023. Logical time for reactive software. In *Workshop on Time-Centric Reactive Software (TCRS)*. ACM.
[32] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. 2021. Toward a lingua franca for deterministic concurrent systems. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 4 (May 2021), Article 36. https://doi.org/10.1145/3448128
[33] N. A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann.
[34] Oded Maler, Zohar Manna, and Amir Pnueli. 1992. From timed to hybrid systems. In *Real-Time: Theory and Practice, REX Workshop*. Springer-Verlag, 447–484.
[35] Friedemann Mattern. 1988. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, Michel Cosnard, Patrice Quinton, Michel Raynal, and Yves Robert (Eds.). North-Holland, 215–226.
[36] Christian Menard, Marten Lohstroh, Soroush Bateni, Matthew Chorlian, Arthur Deng, Peter Donovan, Clément Fournier, Shaokai Lin, Felix Suchert, Tassilo Tanneberger, Hokeun Kim, Jeronimo Castrillon, and Edward A. Lee. 2023. High-performance deterministic concurrency using lingua franca. *arXiv:2301.02444 [cs.PL]* (01/09/2023 2023). https://arxiv.org/abs/2301.02444
[37] Jayadev Misra. 1986. Distributed discrete event simulation. *Comput. Surveys* 18, 1 (1986), 39–65.

[38] Richard A. Muller. 2016. *Now — The Physics of Time*. W. W. Norton and Company.
[39] Saranya Natarajan and David Broman. 2018. Timed C: An extension to the C programming language for real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 227–239. https://doi.org/10.1109/RTAS.2018.00031
[40] Muntasir Raihan Rahman, Lewis Tseng, Son Nguyen, Indranil Gupta, and Nitin Vaidya. 2017. Characterizing and adapting the consistency-latency tradeoff in distributed key-value stores. *ACM Transactions on Autonomous and Adaptive Systems* 11, 4 (January 2017), Article 20. https://doi.org/10.1145/2997654
[41] B.S.Y. Rao and P. Varaiya. 1994. Roadside intelligence for flow control in an intelligent vehicle and highway system. *Transportation Research Part C: Emerging Technologies* 2, 1 (1994), 49–72. https://doi.org/10.1016/0968-090X(94)90019-1
[42] Carlo Rovelli. 2018. *The Order of Time*. Riverhead Books, New York.
[43] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. 2013. Just model! – putting automatic synthesis of node-link-diagrams into practice. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'13)*. San Jose, CA, USA, 75–82.
[44] Reinhard Schwarz and Friedemann Mattern. 1994. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing* 7 (1994), 149–174. https://doi.org/10.1007/BF02277859
[45] Sanaz Shaker Sepasgozar and Samuel Pierre. 2022. Network traffic prediction model considering road traffic parameters using artificial intelligence methods in VANET. *IEEE Access* 10 (2022), 8227–8242. https://doi.org/10.1109/ACCESS.2022.3144112
[46] Reinhard Wilhelm et al. 2008. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 3 (2008), 1–53.
[47] Haifeng Yu and Amin Vahdat. 2006. The costs and limits of availability for replicated services. *ACM Transactions on Computer Systems* 2, 24 (February 2006), 70–113. https://doi.org/10.1145/1124153.1124156
[48] Bernard P Zeigler, Yoonkeon Moon, Doohwan Kim, and George Ball. 1997. The DEVS environment for high-performance modeling and simulation. *IEEE Computational Science and Engineering* 4, 3 (1997), 61–71.
[49] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A. Lee. 2018. AWStream: Adaptive wide-area streaming analytics. In *Proceedings of SIGCOMM*. ACM, Budapest, Hungary. https://doi.org/10.1145/3230543.3230554
[50] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A. Lee. 2018. AWStream: Adaptive wide-area streaming analytics. In *Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. 236–252. https://doi.org/10.1145/3230543.3230554
[51] Yang Zhao, Edward A. Lee, and Jie Liu. 2007. A programming model for time-synchronized distributed real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 259–268.

Received 23 March 2023; revised 2 June 2023; accepted 13 July 2023