

Logical Synchrony and the Bittide Mechanism

Sanjay Lall , *Fellow, IEEE*, Călin Cașcaval , *Fellow, IEEE*, Martin Izzard, *Member, IEEE*, and Tammo Spalink

Abstract—We introduce logical synchrony, a framework that allows distributed computing to be coordinated as tightly as in synchronous systems without the distribution of a global clock or any reference to universal time. We develop a model of events called a logical synchrony network, in which nodes correspond to processors and every node has an associated local clock which generates the events. We construct a measure of logical latency and develop its properties. A further model, called a multiclock network, is then analyzed and shown to be a refinement of the logical synchrony network. We present the bittide mechanism as an instantiation of multiclock networks, and discuss the clock control mechanism that ensures that buffers do not overflow or underflow. Finally we give conditions under which a logical synchrony network has an equivalent synchronous realization.

Index Terms—Computer networks, distributed computing.

I. INTRODUCTION

DISTRIBUTED computation requires processes on networked machines to coordinate, presenting challenges in maintaining a consistent notion of time across nodes. Local clocks require continual realignment to prevent divergence, while distributing a global clock is fragile and expensive at scale. When coordination is focused on correctness, instead of tracking time an option is to track only causality. This takes the form of event sequence information, such as vector clocks, which avoid needing to synchronize clocks but remain expensive at scale.

In this paper, we introduce *logical synchrony*, a novel approach providing a shared notion of time sufficient for reasoning about causality without requiring a shared system-wide clock. Logical synchrony scheduling relies solely on knowledge of graph topology and logical latencies. We present the bittide mechanism, which facilitates efficient implementation of logical synchrony on modern networks, establishing *synchrony* alongside wall-clock time as a primary abstraction. By ensuring that clocks advance in lockstep with data frames sent between nodes, bittide creates a clock mechanism with reduced state-keeping compared to vector clocks, enhancing scalability.

Modern networks, including recent versions of Ethernet, continually transmit frames regardless of nodes sending *actual data* or not, in order to maintain synchronization of SerDes [1] and clock recovery [2] circuits. Leveraging this, the bittide mechanism achieves logical synchrony by directly tying the

clock advancement to the continuous frame transmission of such networks. It is this continual transmission that enables bittide synchronization to occur without the overhead of sending additional information, a benefit over explicit synchronization protocols such as PTP [3], [4]. Applications on networks with clocks synchronized to wall-clock time must utilize clock error-bounds for correctness reasoning [5]. The bittide system enables cycle-accurate coordination without additional clock error-bounds or any associated barriers. This is achieved by defining the clock ordering at neighboring nodes using a graph of frame transmission events.

Logical synchrony is particularly useful for applications with predictable behavior and resource requirements, including financial exchanges [6], databases [4], [5], [7], robotics [8], and large-scale numerical computations such as machine-learning training and inference [9]. Such predictability allows for ahead-of-time scheduling across both communications and computation, which in turn allows for high efficiency and bounded response times. An example use case is ensuring concurrency control correctness in lock-free database transactions by ensuring that all distributed system nodes observe the same order of events.

Ahead-of-time scheduling, inherent to logical synchrony, is naturally limited to applications with predictable communication, memory, and compute cycles. Traditional dynamic communication stacks and scheduling infrastructure can be implemented above bittide transparently, which allows running applications which do not have the requisite predictability, but applications running on these stacks lose the benefits of ahead-of-time scheduling. Further research may extend the utility of logical synchrony to more dynamic and data-dependent situations, for example to support probabilistic ahead-of-time scheduling of such applications where behavior is evolving slowly enough for a scheduler to adapt and reconfigure.

Logical synchrony and bittide have nodes track logical time, which potentially diverges from wall-clock time. This poses a limitation for applications requiring wall-clock time, such as real-time embedded systems or control systems. Addressing this limitation and failure handling requires augmentation of the basic bittide mechanism presented here, and thus are beyond the scope of this paper. A consequence of ahead-of-time scheduling is that failure handling naturally happens independently of scheduling because there is no runtime dynamic scheduler. Node or link failures may necessitate rescheduling execution or communication, which in turn may require application participation.

The bittide mechanism enables processes on distributed network cores to behave as if perfectly synchronized despite individual cores being only imperfectly synchronized. A logical

Manuscript received 20 July 2023; revised 3 July 2024; accepted 9 August 2024. Date of publication 16 August 2024; date of current version 9 September 2024. Recommended for acceptance by O. Ozkasap. (*Corresponding author: Sanjay Lall.*)

The authors are with Google, Mountain View, CA 94043 USA (e-mail: lall@stanford.edu).

Digital Object Identifier 10.1109/TPDS.2024.3444739

synchrony network, abstracting the bittide mechanism, characterizes causality relationships between events. Logical latencies specify these relationships exactly, a striking property that allows precise coordination and reasoning about both system performance and event ordering. Using communication events between processes for logical coordination originates with the work of Lamport [10] and allows precise reasoning about correctness. Logical synchrony ties these events to the repetitive frame transmission events of the network, and thus allows precise coordination and reasoning about the performance of the system as well as the ordering of events, bringing the guarantees available in synchronous execution to distributed systems without the need for a global time reference. Our work extends Lamport’s framework into the efficiency domain, enabling reasoning about both correctness and scheduling.

Synchronous execution models have been used successfully in realtime systems [11], [12], [13] to reason about correctness, in particular meeting deadlines. Often, synchronous abstractions are decoupled from implementation and are used to validate system functional behavior. When mapping synchronous abstractions to asynchronous non-deterministic hardware, work has been done to automate code generation that matches the functional semantics, hiding the non-deterministic behavior of the hardware with explicit synchronization, for example [14]. Logical Execution Time (LET) was introduced by Henzinger and Kirsch [15] to support the design of reactive, cyber-physical systems. More recently, Lingua Franca [16], [17] supports concurrent and distributed programming using time-stamped messages. Lingua Franca exposes to programmers the notion of *reactors* that are triggered in logical time, allowing deterministic reasoning about four common design patterns in distributed systems: alignment, precedence, simultaneity, and consistency. We argue that the causality reasoning in the logical synchrony framework subsumes such design patterns – they are all effectively enabling reasoning about ordering of events in a system that exchanges messages, and as we will show in the paper, this is exactly the class of applications for which logical synchrony determines precisely the causality relationships.

Alternatively, synchronous execution can be implemented using a single global clock. For small real-time systems, cyber-physical systems, and control systems, a global clock can be distributed from a single oscillator. Scaling such systems is difficult because large clock distribution networks introduce delays which must be corrected.

Preceding works such as Sundial [4] have also showcased the difficulty in managing fault tolerance for synchronized real-time clocks. For the majority of systems using wall-clock time as their global clock, synchronization implies exchanging timestamps [3], [18]. Techniques such as TrueTime [5] and White Rabbit [19] attempt to reduce the latency uncertainty, and thus the time-uncertainty bounds, from milliseconds in TrueTime to sub-nanosecond in White Rabbit.

To achieve desired levels of performance using existing network protocols requires expensive time references such as dedicated atomic clocks and networking hardware enhancements to reduce protocol overhead. Time uncertainty is exposed to programmers through an uncertainty interval which guarantees

that current time is within interval bounds for all nodes in the system, such that every node is guaranteed to have passed current time when the bound elapses.

Logical synchrony, formalized in Section II, abstracts the notion of shared time and allows us to avoid a global reference clock or wall-clock. Time is defined only by local clocks decoupled from physical time. The idea is that events at the same node are ordered by local time, and events at different nodes are ordered by causality. As we will show, logical synchrony requires no system-wide global clock and no explicit synchronization (timestamp exchanges or similar), which thereby allows for potentially infinitely scalable systems. Reasoning about ordering of events in logically synchronous systems follows the partial order semantics of Lamport [10] and thus provides equivalence with any synchronous execution that generates identical event graphs.

To establish how logical synchrony can be realized in practice, we first define what logical synchrony means within an abstract model of distributed systems with multiple clocks, defining local clocks in a multiclock network. We show how to combine the FIFO occupancies with the offsets between neighboring clocks, and how this combination is enough to determine the causality relationships.

We then explain how bittide [20], [21], [22] is a mechanism to efficiently implement logical synchrony with real hardware and thereby bring desirable synchronous execution properties to distributed applications efficiently at scale.

A. Mathematical Preliminaries and Notation

An *undirected graph* \mathcal{G} is pair $(\mathcal{V}, \mathcal{E})$ where \mathcal{V} is a set and \mathcal{E} is a subset of the set of 2-element subsets of \mathcal{V} . A *directed graph* \mathcal{G} is pair $(\mathcal{V}, \mathcal{E})$ where $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$ and $(v, v) \notin \mathcal{E}$ for all $v \in \mathcal{V}$. An edge $e \in \mathcal{E}$ in a directed graph may be denoted (u, v) or $u \rightarrow v$. A directed graph may contain a 2-cycle, that is a pair of edges $u \rightarrow v$ and $v \rightarrow u$. An *oriented graph* is a directed graph in which there are no 2-cycles.

Suppose $G = (\mathcal{V}, \mathcal{E})$ is a directed graph, and number the vertices and edges so that $\mathcal{V} = \{1, \dots, n\}$ and $\mathcal{E} = \{1, \dots, m\}$. Then the *incidence matrix* $B \in \mathbb{R}^{n \times m}$ is

$$B_{ij} = \begin{cases} 1 & \text{if edge } j \text{ starts at node } i \\ -1 & \text{if edge } j \text{ ends at node } i \\ 0 & \text{otherwise} \end{cases}$$

for $i = 1, \dots, n$ and $j = 1, \dots, m$.

A *walk* in a directed graph G is a non-empty alternating sequence $v_0, s_0, v_1, s_1, \dots, s_{k-1}, v_k$ in which $v_i \in \mathcal{V}$, $s_i \in \mathcal{E}$, and either $s_i = v_i \rightarrow v_{i+1}$ or $s_i = v_{i+1} \rightarrow v_i$. In the former case we say s_i has *forward* or $+1$ orientation, otherwise we say it has *backward* or -1 orientation. A *path* is a walk in which all vertices are distinct. A *cycle* is a walk in which vertices v_0, \dots, v_{k-1} are distinct, all edges are distinct, and $v_0 = v_k$. Walks, paths, and cycles are called *directed* if all edges are in the forward orientation.

In a directed graph G , given a walk

$$W = (v_0, s_0, v_1, s_1, \dots, s_{k-1}, v_k)$$

the corresponding *incidence vector* $x \in \mathbb{R}^m$ is such that $x_i = 1$ if there exists j such that $i = s_j$ and s_j has forward orientation, and $x_i = -1$ if there exists j such that $i = s_j$ and s_j has reverse orientation, and $x_i = 0$ otherwise. For a directed graph with 2-cycles, there is an edge $u \rightarrow v$ and $v \rightarrow u$, and we assign one of these directions as primary and the other as secondary. This is simply a choice of sign convention. From a directed graph we construct an associated oriented graph by discarding all secondary edges. From an oriented graph we construct an associated undirected graph by discarding all orientations. The concepts of spanning tree and connectedness when applied to a directed graph always refer to the associated undirected graph. The following two results are well-known.

Theorem 1: Suppose $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a directed graph with incidence matrix B , and suppose edges $1, \dots, n-1$ form a spanning tree. Partition B according to

$$B = \begin{bmatrix} B_{11} & B_{12} \\ -\mathbf{1}^\top B_{11} & -\mathbf{1}^\top B_{12} \end{bmatrix}$$

then B_{11} is unimodular. Further

$$B = \begin{bmatrix} B_{11} & 0 \\ -\mathbf{1}^\top B_{11} & \mathbf{1} \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} I & N \\ 0 & I \end{bmatrix}$$

where $N = B_{11}^{-1} B_{12}$.

Proof: See for example Theorem 2.10 of [23]. ■

For convenience, denote by Z the $m \times (m-n+1)$ matrix

$$Z = \begin{bmatrix} -N \\ I \end{bmatrix}$$

Then we have the following important property.

Theorem 2: Every column of Z is the incidence vector of a cycle in \mathcal{G} .

Proof: See, for example, Chapter 5 of [23]. ■

Theorem 1 implies that the columns of Z are a basis for the null space of B , since $BZ = 0$ and $\text{null}(Z) = \{0\}$. The columns of Z are called the *fundamental cycles* of the graph. Note that each of the fundamental cycles is associated with exactly one of the non-tree edges of the graph.

II. LOGICAL SYNCHRONY NETWORKS

The goal of this section is to develop an abstraction which contains two key things; first, a notion of ordering of events such as that of Lamport [10]; and second, a notion of network latency. It turns out that these two ideas may be combined into a simple unified abstraction, which we call the *logical synchrony network*, and this allows analysis of both causality and system performance. We build an event model, in which events may be thought of as ticks of a local clock at each node, corresponding to process execution. The events at neighboring nodes are linked by data transmission. There is no notion of global time, and yet within this framework there is still a notion of latency and duration. We show that ordering of events can be defined in a meaningful way when round-trip latencies are positive.

We start with a formal definition of a logical synchrony network as a directed graph with edge weights, as follows.

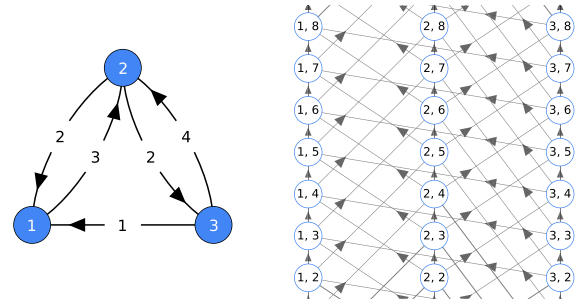


Fig. 1. A logical synchrony network (edges labeled with λ) and corresponding extended graph.

Definition 1: A *logical synchrony network* is a directed graph $(\mathcal{V}, \mathcal{E})$ together with a set of edge weights $\lambda : \mathcal{E} \rightarrow \mathbb{R}$.

In this model, each node corresponds to a processor, and an edge between nodes $i \rightarrow j$ indicates that node i can send data along a physical link to node j . Sent data is divided into tokens which we refer to as *frames*.

Local clocks: Every node has an infinite sequence of *events* associated with it, which can be thought of as compute steps. The events at node i are denoted (i, τ) , where τ is referred to as a *localtick* and thereby implicitly defines a local clock. We define the set of all events

$$\mathcal{V}_{\text{ext}} = \{(i, \tau) \mid i \in \mathcal{V}, \tau \in \mathbb{Z}\}$$

Events at one node are aligned to events at other nodes by the transmission of frames. At localtick τ and node i , a frame is sent from node i to node j , and it arrives at node j at localtick $\tau + \lambda_{i \rightarrow j}$. The constant $\lambda_{i \rightarrow j}$ is called the *logical latency*. We define the following binary relation.

Definition 2: Event (i, τ) is said to *directly send to* the event (j, ρ) if $(i, j) \in \mathcal{E}$ and $\rho = \tau + \lambda_{i \rightarrow j}$, or $i = j$ and $\rho = \tau + 1$. We use the notation

$$(i, \tau) \rightarrow (j, \rho)$$

to mean (i, τ) directly sends to (j, ρ) , and define the set

$$\mathcal{E}_{\text{ext}} = \{((i, \tau), (j, \rho)) \mid (i, \tau) \rightarrow (j, \rho)\}$$

The graph $\mathcal{G}_{\text{ext}} = (\mathcal{V}_{\text{ext}}, \mathcal{E}_{\text{ext}})$ is called the *extended graph* of the logical synchrony network.

This relation may be viewed as an infinite directed graph with vertex set \mathcal{V}_{ext} and directed edges $(i, \tau) \rightarrow (j, \rho)$. In this graph, those edges $(i, \tau) \rightarrow (j, \rho)$ for which $i = j$ are called *computational edges*. An edge that is not a computational edge is called a *communication edge*. Fig. 1 illustrates a logical synchrony network and its corresponding extended graph. Definition 2 adds two types of edges to the extended graph. Computational edges are vertical in the figure, and they connect (i, τ) to $(i, \tau + 1)$. These express the relationship between sequential events at node i . Communication edges are non-vertical, and connect (i, τ) to $(j, \tau + \lambda_{i \rightarrow j})$. These express the relationship between the sending of a frame from node i at time τ and its reception at node j at time $\tau + \lambda_{i \rightarrow j}$.

The localticks define a separate and ideal notion of local duration at each node by counting events (i.e., frame transmissions or

receptions.) We can speak of the event (i, τ) as occurring at time τ localticks on node i . We say that event $(i, \tau + a)$ happens a localticks after event (i, τ) , for any $a \in \mathbb{Z}$. We cannot in general compare clock values at two different nodes.

Execution: This model captures the local evolution of time at each node $i \in \mathcal{V}$, and the transmission of frames between them. Although we do not investigate execution models in this paper, it is possible to define many different execution semantics. One simple choice is the functional model, where frames carry data, and associated with each event $(i, \tau) \in \mathcal{V}_{\text{ext}}$ in the extended graph we have a function, which maps data from incoming edges to data on outgoing edges. Another possibility is to have a more procedural model, where events in \mathcal{V}_{ext} correspond to the clock ticks of a processor in the corresponding \mathcal{V} . For the purposes of this paper it is not necessary to specify how many bits each frame contains but we assume all frames on a given link are equally sized.

The abstract models considered in this paper consist of sequences of events which extend infinitely far into both the future and the past. It is possible to extend this model to include system startup, for example by introducing a minimum node within the extended graph, or by modifying the execution model. We do not address startup within this paper.

Frames and logical latency: If A denotes a particular frame sent $i \rightarrow j$, then we will make use of the notation $\text{receive}(A)$ to refer to the localtick at node j when A arrives at j . Similarly $\text{send}(A)$ refers to the localtick at node i when A was sent. This notation leaves implicit the source and destination of frame A , in that i, j are not included as arguments of the send and receive functions. We do not as yet assume any particular mechanism for transmission of frames, but we assume that frames are received in the order that they are sent, without any loss. Note that the logical latency has no connection to *physical latency*. If we were to measure the send and receive times with respect to a global notion of time, we would know that, for example, the receive time must be greater than the send time. In the framework presented here, that is not the case; the localticks are strictly local, and as a result there is no such requirement on their numerical value; the logical latency $\lambda_{i \rightarrow j}$ may be negative. This is, of course, a statement about the clocks, not about causality.

In words, the logical latency is the time of arrival *in the receiver's clock* minus the time of departure *in the sender's clock*. There are several observations worth making about logical latency.

- *Logical latency is constant:* For any two nodes i, j , every frame sent $i \rightarrow j$ has the same logical latency. It is a property of the edge $i \rightarrow j$ in \mathcal{E} .
- *Despite the name, logical latency is not a measure of length of time or duration:* It is not the case that if $\lambda_{i \rightarrow j}$ is greater than $\lambda_{p \rightarrow q}$ then it takes longer for frames to move from i to j than it does for frames to move from p to q . (In fact, we do not have a way within this framework to compare two such quantities.)
- *The logical latency can be negative:*

Logical latencies and paths: Logical latencies add along a path. Suppose node i sends a frame B along edge $i \rightarrow j$ to node

j , and then node j forwards it $j \rightarrow k$. Then we have

$$\text{receive}(B) = \text{send}(B) + \lambda_{i \rightarrow j} + \lambda_{j \rightarrow k}$$

This means that we can speak of the logical latency of the path $i \rightarrow j \rightarrow k$ as being $\lambda_{i \rightarrow j} + \lambda_{j \rightarrow k}$, and more generally we can define the logical latency of a directed path $\mathcal{P} = v_0, s_0, v_1, s_1, \dots, s_{k-1}, v_k$ from node v_0 to node v_k in \mathcal{G} . The logical latency is path dependent; two paths with the same endpoints may have different logical latencies. We have

$$\lambda_{\mathcal{P}} = \sum_{i=0}^{k-1} \lambda_{s_i}$$

This makes sense, which is potentially surprising because we are measuring arrival and departure times with different clocks. Since frames are being relayed, there may be additional delay at intermediate nodes (i.e., additional compute steps) which would need to be included when determining the destination event. Logical latencies are defined such that they do not include this additional delay.

A. Ordering of Events

A fundamental question regarding causality arises in the study of distributed systems. Given two events, we would like to determine which happened first. In a nonrelativistic physical setting, such a question is well-defined. In a relativistic setting, there are events which are separated in space for which the relative order is undetermined — the order depends on the observer. Something similar happens in distributed systems, as was pointed out by Lamport [10]. Given two events, instead of asking which event happened first, a more useful question is to ask which event, if any, *must have* happened first. The framework for distributed clocks developed by Lamport [10] established that there is a partial ordering on events determined by one event's ability to influence another by the sending of messages. In that paper the author defines a global notion of time consistent with said partial order. Subsequent work [24], [25] defines *vector clocks* which assign a vector-valued time to events for which the partial ordering is equivalent to that defined by message-passing. We would like to construct the corresponding notion of causality in a logical synchrony network.

We define below the \sqsubseteq relation, which can be used to define a partial order on \mathcal{G}_{ext} provided we can ensure that it is acyclic. To do this, we consider round-trip times.

Round trip times: Logical latencies are not physical latencies, despite the additive property. However, there is one special case where logical latency is readily interpreted in such physical terms, specifically the time for a frame A to traverse a cycle in the graph, the cycle round-trip time. Suppose $\mathcal{C} = v_0, s_0, v_1, s_1, \dots, s_{k-1}, v_k$ is a directed cycle, then

$$\lambda_{\mathcal{C}} = \text{receive}(A) - \text{send}(A)$$

is the round-trip time measured in localticks. Two different cycles from a single node i may have different round-trip times, and these are comparable durations since they are both measured

in localticks at that node. We have

$$\lambda_C = \sum_{i=0}^{k-1} \lambda_{s_i}$$

We make the following definition.

Definition 3: A logical synchrony network is said to have *positive round-trip times* if, for every directed cycle C in the graph \mathcal{G} we have $\lambda_C > 0$.

We then have the following result, which says that if the round-trip times around every directed cycle in the logical synchrony network are positive, then the extended graph is acyclic.

Theorem 3: If a logical synchrony network has positive round-trip times then its extended graph is acyclic.

Proof: Suppose for the purpose of a contradiction that the extended graph is cyclic. Then there exists a directed cycle $\mathcal{C}_1 = v_0, s_0, v_1, s_1, \dots, s_{k-1}, v_k$ where each $v_j \in \mathcal{V}_{\text{ext}}$ is a pair $v_j = (i_j, \tau_j)$. Since the start and end node is the same, we have

$$\begin{aligned} 0 &= \sum_{j=1}^{k-1} (\tau_{j+1} - \tau_j) \\ &= \sum_{j \in C_{\text{comp}}} (\tau_{j+1} - \tau_j) + \sum_{j \notin C_{\text{comp}}} (\tau_{j+1} - \tau_j) \end{aligned} \quad (1)$$

where C_{comp} is the set of indices j such that (v_j, v_{j+1}) is a computational edge. Each of the computational edges has $\tau_{j+1} - \tau_j = 1$. If all of the edges in the graph are computational then the right-hand side is positive. If there are some communication edges, then the second of the two terms on the right-hand side is positive due to the assumption that the logical synchrony graph has positive round-trip times, and again the right-hand-side is positive. This contradicts the claim that the sum is zero. ■

This acyclic property is necessary for an execution model based on function composition to be well-defined. It also allows us to define a temporal partial ordering between events in \mathcal{G}_{ext} . Since a logical synchrony network with positive round-trip times has an extended graph which is acyclic, the reachability relation on the extended graph defines a partial order. Specifically, we write

$$(i, \tau) \sqsubset (j, \rho)$$

if there is a directed path from (i, τ) to (j, ρ) in the extended graph. Here, the notation is meant to be similar to $<$, indicating *comes before*. Under these conditions, a logical synchrony network is a distributed system in the sense of Lamport [10], with logical latencies providing strict inter-event timings at any node $i \in \mathcal{V}$. The partial ordering on the induced logical synchrony network has exactly the property that, if $u \sqsubset v$, then u must have happened before v .

III. EQUIVALENCE OF LSNS

The goal of this section is to establish an invariant, which we will use in the subsequent sections to analyze system correctness. We introduce the idea of clock relabeling, which modifies logical latencies while preserving the interconnection of events and the

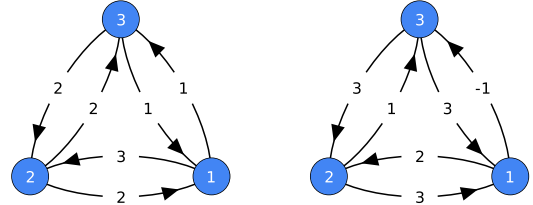


Fig. 2. Two equivalent logical synchrony graphs (edges labeled with λ). Relabeling the clocks using $c = (1, 2, 3)$ maps the left-hand graph to the right-hand one.

underlying physical system. We show that the round trip times, being physically measurable properties, cannot change. We use this invariant to characterize when two networks are physically the same, even though their clock labels may be different.

Two logical synchrony networks may have different logical latencies, but be nonetheless equivalent for the purpose of executing processes. An example is given by the graphs in Fig. 2.

This arises because we can relabel the events. Specifically, given a logical synchrony network with events \mathcal{V}_{ext} , we define a new logical synchrony network. Given $c_1, \dots, c_n \in \mathbb{Z}$, we relabel event (i, τ) as $(i, \tau + c_i)$. This is a relabeling of the vertices of the graph \mathcal{G}_{ext} . In \mathcal{G}_{ext} we have edges

$$(i, \tau) \rightarrow (j, \tau + \lambda_{i \rightarrow j})$$

for every $i \neq j \in \mathcal{V}$ and $\tau \in \mathbb{Z}$. Under the relabeling, these are mapped to

$$(i, \tau + c_i) \rightarrow (j, \tau + \lambda_{i \rightarrow j} + c_j)$$

and since there is such an edge for all $\tau \in \mathbb{Z}$ the edge set of the relabeled extended graph is

$$\hat{\mathcal{E}}_{\text{ext}} = \{((i, \tau), (j, \tau + \lambda_{i \rightarrow j} + c_j - c_i)) \mid i, j \in \mathcal{V}, \tau \in \mathbb{Z}\}$$

This is the extended graph for a logical synchrony network with logical latencies

$$\hat{\lambda}_{i \rightarrow j} = \lambda_{i \rightarrow j} + c_j - c_i$$

This leads us to the following definition of equivalence.

Definition 4: Suppose we have two logical synchrony networks on a directed graph $(\mathcal{V}, \mathcal{E})$, with edge weights λ and $\hat{\lambda}$. We say these LSNS are *equivalent* if there exists $c_1, \dots, c_n \in \mathbb{Z}$ such that, for all $i, j \in \mathcal{V}$,

$$\hat{\lambda}_{i \rightarrow j} = \lambda_{i \rightarrow j} + c_j - c_i \quad (2)$$

We can write this equation as

$$\lambda - \hat{\lambda} = B^T c$$

where B is the incidence matrix of \mathcal{G} . Relabeling the clocks results in a relabeling of the corresponding extended graph. Since this only changes the labels of the nodes, not how the nodes are interconnected, any code which is executable on one graph may also be executed on the other (but any references to particular localticks will need to be changed.) Physically measurable properties such as round-trip times cannot change under such a simple relabeling. We have

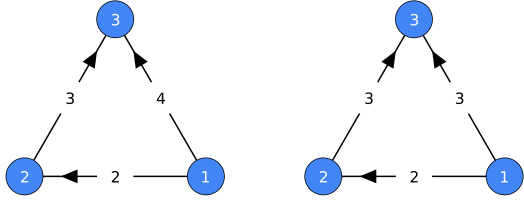


Fig. 3. Two non-equivalent logical synchrony graphs with no directed cycles (edges labeled with λ).

Proposition 1: If two LSNs are equivalent, they will have the same round trip times on every directed cycle.

Proof: The round-trip times for a directed cycle $\mathcal{C} = v_0, s_0, v_1, s_1, \dots, s_{k-1}, v_k$ in \mathcal{G} satisfy

$$\sum_{j=0}^{k-1} \lambda_{s_j} = \sum_{j=0}^{k-1} \hat{\lambda}_{s_j}$$

which follows from (2). \blacksquare

The converse is not generally true, as the following example shows.

Example 1: Consider the logical synchrony networks shown in Fig. 3. Both networks have the same underlying graph, which has no directed cycles, and so the round trip times on every directed cycle are trivially equal on both networks. If we order the edges $((1 \rightarrow 2), (2 \rightarrow 3), (1 \rightarrow 3))$ then we have incidence matrix

$$B = \begin{bmatrix} 1 & 0 & 1 \\ -1 & 1 & 0 \\ 0 & -1 & -1 \end{bmatrix}$$

which has $\text{rank}(B) = 2$. In the left-hand network of Fig. 3 the logical latencies are $\lambda_1 = 2, \lambda_2 = 3$ and $\lambda_3 = 4$, and in the right-hand network they are $\hat{\lambda}_1 = 2, \hat{\lambda}_2 = 3$ and $\hat{\lambda}_3 = 3$. Therefore

$$\lambda - \hat{\lambda} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad (3)$$

and there is no vector c such that $\lambda - \hat{\lambda} = B^T c$.

If the round trip times are equal around every cycle, accounting for signs and orientations, then the two logical synchrony networks are equivalent. To show this, we need a preliminary result.

Lemma 1: Let the graph be connected. Suppose $y \in \mathbb{Z}^m$, and for every cycle \mathcal{C} we have $y^T x = 0$ for the corresponding incidence vector x . Then $y = B^T c$ for some $c \in \mathbb{Z}^n$.

Proof: Pick a spanning tree, and partition B according to the spanning tree. Let $N = B_{11}^{-1} B_{12}$. Partition y according to

$$y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

where $y_1 \in \mathbb{Z}^{n-1}$. We choose

$$c = \begin{bmatrix} B_{11}^{-T} y_1 \\ 0 \end{bmatrix}$$

and note that since B_{11} is unimodular c must be integral. Using Theorem 1 we have

$$\begin{aligned} B^T c &= \begin{bmatrix} I & 0 \\ N^T & I \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} B_{11}^T & -B_{11}^T \mathbf{1} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} B_{11}^{-T} y_1 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} I & 0 \\ N^T & I \end{bmatrix} \begin{bmatrix} y_1 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \end{aligned}$$

as desired, where in the last line we use Theorem 2 to show that

$$y^T \begin{bmatrix} -N \\ I \end{bmatrix} = 0$$

since y is orthogonal to the incidence vectors of the fundamental cycles. \blacksquare

We now state and prove a variant of Proposition 1 which is both necessary and sufficient.

Theorem 4: Suppose we have two logical synchrony networks on a connected directed graph $(\mathcal{V}, \mathcal{E})$, with edge weights λ and $\hat{\lambda}$. These networks are equivalent if and only if they have the same signed round trip times on every cycle in \mathcal{G} . That is, for every cycle $\mathcal{C} = v_0, s_0, v_1, s_1, \dots, s_{k-1}, v_k$ we have

$$\sum_{j=0}^{k-1} \lambda_{s_j} o_j = \sum_{j=0}^{k-1} \hat{\lambda}_{s_j} o_j \quad (4)$$

where o_j is the orientation of edge s_j on the cycle \mathcal{C} .

Proof: Equation (4) means that for every cycle \mathcal{C} with incidence vector x we have

$$(\lambda - \hat{\lambda})^T x = 0$$

Then Lemma 1 implies that $\lambda - \hat{\lambda} = B^T c$ for some integer vector c , and hence λ and $\hat{\lambda}$ are equivalent. \blacksquare

What this means, in particular, is that in Example 1 the graph does not have a directed cycle but it does have a cycle, where edges $1 \rightarrow 2$ and $2 \rightarrow 3$ are oriented in the forward direction, and edge $1 \rightarrow 3$ is oriented in the backward direction. Then λ and $\hat{\lambda}$ are equivalent if and only if

$$\lambda_1 + \lambda_2 - \lambda_3 = \hat{\lambda}_1 + \hat{\lambda}_2 - \hat{\lambda}_3$$

Since this does not hold for λ and $\hat{\lambda}$ in that example, those two networks are not equivalent.

One cannot verify equivalence by checking pairs of nodes. That is, it is not sufficient to simply check the length-2 round trip times, as the following example shows.

Example 2: Suppose \mathcal{G} is the complete graph with 3 nodes. For the two logical synchrony networks, shown in Fig. 4, the length-2 round trip times are

$$\lambda_{1 \rightarrow 2 \rightarrow 1} = 5$$

$$\lambda_{2 \rightarrow 3 \rightarrow 2} = 4$$

$$\lambda_{1 \rightarrow 3 \rightarrow 1} = 2$$

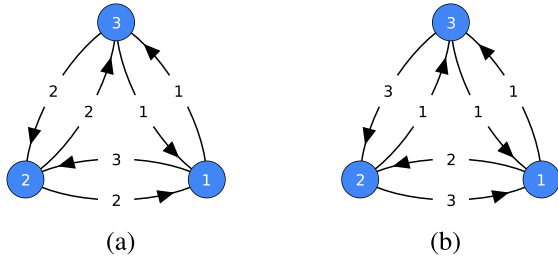


Fig. 4. Logical synchrony networks for Example 2.

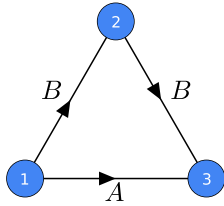


Fig. 5. Triangle invariant.

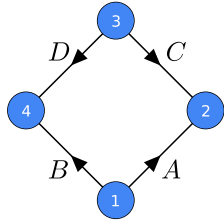


Fig. 6. Diamond invariant.

and they are the same for $\hat{\lambda}$. However, these networks are not equivalent. There is no way to relabel so that the logical latencies are the same. This is because the length-3 round trip times are $\lambda_{1 \rightarrow 2 \rightarrow 3 \rightarrow 1} = 6$ and $\hat{\lambda}_{1 \rightarrow 2 \rightarrow 3 \rightarrow 1} = 4$.

Invariants: As shown by the above results, round-trip times around directed cycles are invariant under relabeling. Cycles which are not directed also result in invariants which may be physically measured and interpreted. We give some examples below.

Example 3: Fig. 5 shows a triangle graph in which node 1 sends frame A to node 3, and simultaneously sends frame B to node 3 via node 2. Then $\text{receive}(B) - \text{receive}(A)$ is measured in local ticks at node 3, and it is invariant under relabeling.

Example 4: Fig. 6 shows a square graph. Here node 1 sends frame A to node 2 and simultaneously sends frame B to node 4. Node 3 sends frame C to node 2 and simultaneously sends frame D to node 4. Note that the transmissions of node 1 and node 3 are not synchronized with each other. Then the quantity

$$(\text{receive}(A) - \text{receive}(C)) - (\text{receive}(B) - \text{receive}(D))$$

is invariant under clock relabelings.

Equivalent networks can have different logical latencies, but must have the same round-trip times. The question of how

much freedom this leaves is interesting, and has an important consequence which we discuss below. We first show that one can set the logical latencies arbitrarily on any spanning tree.

Theorem 5: Suppose \mathcal{G}, λ is a logical synchrony network, where $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Suppose $\mathcal{T} \subset \mathcal{E}$ is a spanning tree. Then for any $\gamma : \mathcal{T} \rightarrow \mathbb{Z}$ there exists $c \in \mathbb{Z}^n$ such that

$$\gamma_{i \rightarrow j} = \lambda_{i \rightarrow j} + c_j - c_i \text{ for all } i \rightarrow j \in \mathcal{T}$$

Proof: We would like to show that there exists $c \in \mathbb{Z}^n$ such that

$$[I \ 0] (\lambda - \gamma) = [I \ 0] B^T c$$

Let y_1 be the left-hand side, then using Theorem 1, this is equivalent to

$$y_1 = [B_{11}^T \ -B_{11}\mathbf{1}] c$$

and hence we may choose

$$c = \begin{bmatrix} B_{11}^{-T} y_1 \\ 0 \end{bmatrix}$$

which is integral since B_{11} is unimodular. \blacksquare

We can use this result in the following way. There is no requirement within this framework that logical latencies be nonnegative. However, it turns out that any logical synchrony network which has nonnegative round-trip times is equivalent to one with nonnegative logical latencies. We state and prove this result below. This result will be useful when we discuss multiclock networks in the subsequent section.

Theorem 6: Suppose \mathcal{G}, λ is a logical synchrony network with \mathcal{G} strongly connected, and for every directed cycle \mathcal{C} the round-trip logical latency $\lambda_{\mathcal{C}}$ is nonnegative. Then there exists an equivalent LSN with edge weights $\hat{\lambda}$ which are nonnegative.

Proof: Pick a node r . Since the graph has no negative cycles, there exists a spanning tree \mathcal{T} , rooted at r , with edges directed away from the root, each of whose paths is a shortest path [26]. Use Theorem 5 to construct c such that

$$\lambda_{i \rightarrow j} + c_j - c_i = 0 \text{ for all } i \rightarrow j \in \mathcal{T}$$

As a result, we have $\lambda_{i \rightarrow j} = c_i - c_j$ for all edges $i \rightarrow j$ in the tree \mathcal{T} . Denote by $t_{i \rightarrow k}$ the length of the path from i to k in the tree. Then we have $t_{i \rightarrow k} = c_i - c_k$.

Since this is a shortest path tree, we have for any edge $i \rightarrow j$

$$t_{r \rightarrow i} + \lambda_{i \rightarrow j} \geq t_{r \rightarrow j}$$

because the path in the tree from r to j must be no longer than the path via node i . Therefore

$$c_r - c_i + \lambda_{i \rightarrow j} \geq c_r - c_j$$

Setting $\hat{\lambda}_{i \rightarrow j} = \lambda_{i \rightarrow j} + c_j - c_i$ for all edges we find $\hat{\lambda}_{i \rightarrow j} \geq 0$ as desired. \blacksquare

This result says that, if we have a shortest path tree, we can relabel the clocks so that the logical latency is zero on all edges of that tree, and with that new labeling the logical latency will be nonnegative on every tree edge. An example is given in Fig. 7.

Note also that an edge having zero logical latency does not imply that communication between the endpoints is instantaneous; only that the numerical value of the time at which the

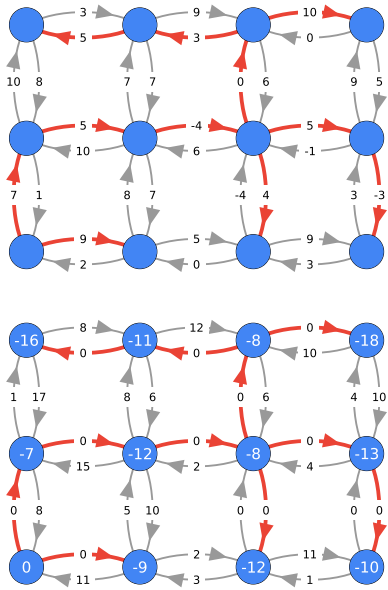


Fig. 7. Relabeling so that logical latencies are nonnegative. The upper graph shows edges labeled with λ . The root node is in the lower left, and the shortest-path spanning tree is shown in red. The lower graph shows an equivalent LSN, with nodes i labeled with c_i , and the corresponding logical latencies $\hat{\lambda}_{i \rightarrow j} = \lambda_{i \rightarrow j} + c_j - c_i$. All logical latencies in this graph are nonnegative.

frame is received is equal to the numerical value of the time at which it was sent.

IV. MULTICLOCK NETWORKS

The objective of this section is to build a model of a physical system, and relate its correctness to the invariants of the previous section. We introduce a model in which there are physical clocks at each node, and the nodes pass data to each other, according to specific timed sequential communications which occur through FIFOs. We call such a system a *multiclock network*. We show that the latencies that arise satisfy exactly the semantics of the abstract latencies of logical synchrony networks. We further show that the natural requirements that the FIFO occupancies be bounded leads to the physical requirement that round trip times are nonnegative. In other words, building a correct multiclock network will result in a correct logical synchrony network.

We formulate the relationship between events on a network in terms of physical clocks, leading to a mathematical definition called the multiclock network. We show that multiclock networks are special types of logical synchrony networks.

We will use t to denote an idealized notion of time, called *wall-clock time*, or *ideal time* [27]. Time on the network is *multiform* [11], in the sense that the nodes on the network each maintain their own sense of time. At each node, there is a real-valued clock, denoted by θ_i . Its units are the *localticks*. We refer to the value θ_i as the *local time* or *phase* at node i . Local time has no quantitative relationship to physical or wall-clock time. In particular, we do not view θ_i as an approximation to wall-clock time and consequently clocks at two distinct nodes are inherently unrelated.

At a node i , a processor can read the value θ_i , its own clock, but cannot access the value θ_j at any other node $j \neq i$. We mathematically model θ_i as a function of physical time t , so that $\theta_i : \mathbb{R} \rightarrow \mathbb{R}$, without implying anything about its construction; it simply means that if at physical time t a hypothetical outside observer were to read clock i , it would read value $\theta_i(t)$. What is required is that θ_i is continuous and increasing, so that $\theta_i(s) < \theta_i(t)$ if $s < t$. We emphasize again that this does not imply that any processes running on the system can access wall-clock time t . The quantity θ_i is not related to physical time.

At times t where θ_i is differentiable, we define the frequency ω_i of the clock θ_i by

$$\omega_i(t) = \frac{d\theta_i(t)}{dt}$$

At a node i , a clock generates an infinite sequence of events, also referred to as *localticks*, which happen whenever θ_i is an integer. Clocks are not required to be periodic, and this definition of frequency is applicable in the general aperiodic case. Clocks at different nodes may have very different frequencies. If the frequency at node i is large, then events at that node occur more often.

We model the process of frame transmission from node i to node j as a FIFO, but real-world implementations are likely to consist of uninterrupted physical communication streams feeding into memory buffers. Every node can access the output (or head) of the FIFO corresponding to each of its incoming links, and the input (or tail) of the FIFO corresponding to each of its outbound links. We will discuss below the requirement that FIFOs neither overflow nor underflow.

Logical synchrony in multiclock networks: With every localtick, node i inserts a frame at the tail of each of its outgoing link FIFOs and removes a frame from the head of each of its incoming link FIFOs. This lock-step alignment of input and output is the fundamental synchronization mechanism that imposes logical synchrony upon the network. At each node, with every localtick, one frame is removed from each incoming FIFO and one frame is sent on each outgoing FIFO.

Formal definition of multiclock network: We now turn to a mathematical model that will enable us to analyze the behavior of this system.

Definition 5: A *multiclock network* is a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ together with continuous increasing functions $\theta_i : \mathbb{R} \rightarrow \mathbb{R}$ for each $i \in \mathcal{V}$, and edge weights $\lambda : \mathcal{E} \rightarrow \mathbb{Z}$.

This definition contains the entire evolution of the clock phases θ_i , and the link properties $\lambda_{i \rightarrow j}$. We will discuss the physical meaning of $\lambda_{i \rightarrow j}$ below. Unlike the logical synchrony network, where events are abstract and have no physical time associated with them, in a multiclock network the global timing of all events is defined by the clocks θ . We will show that a multiclock network is a special case of a logical synchrony network, and the constants λ are the associated logical latencies. To do this, we model the behavior of the FIFOs connecting the nodes.

FIFO model: If $i \rightarrow j$ in the graph \mathcal{G} , then there is a FIFO connecting node i to node j . With every localtick at node i , a frame is added to this FIFO, and with every localtick at node j , a

frame is removed from the FIFO. We number the frames in each FIFO by $k \in \mathbb{Z}$, according to the localtick at the sender, and the frames in the FIFO are those with k satisfying

$$\alpha_{i \rightarrow j}(t) \leq k \leq \beta_{i \rightarrow j}(t)$$

where α and β specify which frames are currently in the FIFO at time t . The FIFO model is as follows.

$$\beta_{i \rightarrow j}(t) = \lfloor \theta_i(t) \rfloor \quad (5)$$

$$\alpha_{i \rightarrow j}(t) = \lfloor \theta_j(t) \rfloor - \lambda_{i \rightarrow j} + 1 \quad (6)$$

Equation (5) means that frames are added with each localtick at the sender, and numbered according to the sender's clock. Equation (6) means that frames are removed with each localtick at the receiver. The constant λ is to account for the offset between the frame numbers in the FIFO and the clock labels at the receiver. (We add 1 for convenience.) This offset must be constant, since one frame is removed for each receiver localtick. This constant is specified by the multiclock network model in Definition 5.

This model precisely specifies the location of every frame on the network at all times t . In particular, this determines the FIFO occupancy at startup. For any time t_0 , the specification of λ is equivalent to specifying the occupancy of the FIFOs at time t_0 . This allows us to have a well-defined FIFO occupancy without requiring an explicit model of startup.

Logical latency: Logical latency is the fundamental quantity which characterizes the discrete behavior of a network, and allows us to ignore the details of the clocks θ_i . The idea is that we can understand the logical structure of the network, such as the events, the execution model, and causality, without needing to know specific wall-clock times at which these things occur.

We now show that the quantity $\lambda_{i \rightarrow j}$ corresponds to the logical latency. Suppose a frame is sent from node i at localtick $k \in \mathbb{Z}$, and wall-clock time t_{send}^k . Then $\theta_i(t_{\text{send}}^k) = k$. Let the time which it is received at node j be denoted by t_{rec}^k . Both t_{send}^k and t_{rec}^k are wall-clock times, and apart from the causality constraint that the frame must be received after it is sent, there is no constraint on the difference between these times; that is, the *physical latency* $t_{\text{rec}}^k - t_{\text{send}}^k$ may be large or small. In general, physical latency will be affected by both the number of frames in the FIFO $i \rightarrow j$ as well as the time required for a frame to be physically transmitted. We do not presuppose requirements on the physical latency.

Lemma 2: Suppose frame k is sent from node i to node j . Then t_{send}^k and t_{rec}^k satisfy

$$\theta_i(t_{\text{send}}^k) = k \quad (7)$$

$$\theta_j(t_{\text{rec}}^k) = k + \lambda_{i \rightarrow j} \quad (8)$$

and hence the logical latency is given by

$$\lambda_{i \rightarrow j} = \theta_j(t_{\text{rec}}^k) - \theta_i(t_{\text{send}}^k) \quad (9)$$

Proof: Since frames in the FIFO $i \rightarrow j$ are numbered according to the sender's clock, we have

$$t_{\text{send}}^k = \inf\{t \mid \beta_{i \rightarrow j}(t) = k\}$$

that is, t_{send}^k is the earliest time at which frame k is in the FIFO from i to j . Since the floor function is right continuous, this gives (7). Similarly, we have

$$t_{\text{rec}}^k = \inf\{t \mid \alpha_{i \rightarrow j}(t) = k + 1\}$$

which is the first time t at which the lowest-numbered frame in the FIFO is number $k + 1$, and therefore this is the time at which frame k has just left the FIFO, and hence has just arrived at the destination. This implies (8), and the logical latency follows. ■

Unlike the physical latency $t_{\text{rec}} - t_{\text{send}}$, the logical latency $\theta_j(t_{\text{rec}}^k) - \theta_i(t_{\text{send}}^k)$ does not change over time. Note also that the logical latency is an integer. Since the logical latency is constant, we can conclude that every multiclock network is a logical synchrony network; more precisely, the logical latencies defined by the multiclock network satisfy the same properties as those of a logical synchrony network.

A. Realizability

We now turn to an analysis of the occupancy of the FIFOs in more detail. A frame is considered *in-transit* from $i \rightarrow j$ at time t if it has been sent by node i but not yet received by node j ; that is, if it is in the FIFO from i to j . Define $\nu_{i \rightarrow j}(t)$ to be the number of frames in transit $i \rightarrow j$. Then we have

$$\begin{aligned} \nu_{i \rightarrow j}(t) &= \beta_{i \rightarrow j}(t) - \alpha_{i \rightarrow j}(t) + 1 \\ &= \lfloor \theta_i(t) \rfloor - \lfloor \theta_j(t) \rfloor + \lambda_{i \rightarrow j} \end{aligned} \quad (10)$$

and this holds for all t . Here we can see that the constant $\lambda_{i \rightarrow j}$ is a property of the link $i \rightarrow j$, which determines the relationship between the clock phases at each end of the link and the number of frames in transit.

So far in this model, there is nothing that prevents the FIFO occupancy on an edge $i \rightarrow j$ from becoming negative. If the clock at node θ_j has a higher frequency than the clock at θ_i , and if that frequency difference is maintained for long enough, then the FIFO $i \rightarrow j$ will be rapidly emptied. In this case, θ_j will become much larger than θ_i , and from (10) we have that $\nu_{i \rightarrow j}$ will become negative. Similarly, the FIFO will overflow if the frequencies become imbalanced in the other direction. In [22] a technique using a dynamically switching control algorithm is presented that allows prevention of such behaviors. We make the following definition.

Definition 6: A multiclock network is called *realizable* if there exists $\nu_{\text{max}} \in \mathbb{R}$ such that for all edges $i \rightarrow j$

$$0 \leq \nu_{i \rightarrow j}(t) \leq \nu_{\text{max}} \quad \text{for all } t \in \mathbb{R} \quad (11)$$

Note that this requirement must hold for all positive and negative time t . The terminology here is chosen to be suggestive, in that we would like a condition which implies that we can physically implement a multiclock network. A physically necessary condition is that the FIFO occupancies are bounded and cannot be negative.

Cycles and conservation of frames: Cycles within a multiclock network have several important properties. The first is *conservation of frames*, as follows.

Theorem 7: Suppose $\mathcal{C} = v_0, s_0, v_1, s_1, \dots, s_{k-1}, v_k$ is a directed cycle in a multiclock network. Then

$$\sum_{i=0}^{k-1} \nu_{s_i}(t) = \lambda_{\mathcal{C}}$$

In particular, the number of frames in transit around the cycle is constant, and is the sum of the logical latencies on the cycle.

Proof: The proof follows immediately from (10). ■

An immediate corollary of this is that, in a physical network, if every edge of \mathcal{G} is on a cycle, then the number of frames in the network is finite and the upper bound condition for realizability is satisfied. This is the case, for example, in a strongly connected graph. Note that this holds because, in a physical network, the FIFO occupancy cannot be negative. It is not the case that the FIFO model used here implies that ν is upper bounded, since in the model some FIFO lengths may become large and negative while others become large and positive.

This theorem is particularly evocative in the simple and common case where we have two nodes i, j connected by links in both directions. In this case, whenever i receives a frame, it removes it from its incoming FIFO from j , and adds a new frame to the outgoing FIFO to j . Thus the sum of the occupancies of the two FIFOs is constant.

The following result relates round trip times to realizability.

Theorem 8: Suppose \mathcal{C} is a cycle in a realizable multiclock network. Then $\lambda_{\mathcal{C}} \geq 0$.

Proof: This follows immediately from Theorem 7 and Definition 6. ■

That is, a realizable multiclock network has the important physical property that all round-trip times are nonnegative. The monotonic property of θ implies that this holds in both localticks and wall-clock time. No matter what path a frame takes around the network, it cannot arrive back at its starting point before it was sent. However, it is possible, within the class of realizable networks defined so far, for this sum to be equal to zero. In this case one would have a frame arrive at the time it is sent. This would require some pathological conditions on the clocks. This is an extreme case corresponding to the limit where frames spend zero time in the FIFOs, which in a physical network would require that the link have zero link latency. For example, in the case of a length 2 cycle between nodes i and j , we would need $\theta_j(t) = \theta_i(t) + \lambda_{i \rightarrow j}$ and $\theta_i(t) = \theta_j(t) + \lambda_{j \rightarrow i}$, which would give $\lambda_{i \rightarrow j} = \lambda_{j \rightarrow i}$. Since the clocks are related by integer constants, they tick at exactly the same times.

B. Equivalent Synchronous Systems

We now consider the class of perfectly synchronous systems, where all of the nodes of the graph share a single clock. The links between the nodes are FIFOs as before, and as a result of the synchronous assumption their occupancies are constant. This is a particular instance of the multiclock network where all clocks θ_i are equal.

Such a system has an extended graph, and it has logical latencies which do not change with time, and are equal to the occupancies of the FIFOs, according to (10). Because the system is synchronous, the FIFOs behave like a chain of delay buffers.

The corresponding execution model, defined by the extended graph, is identical to that of a logical synchrony network with the same logical latencies. Said another way, a logical synchrony network is equivalent to a perfectly synchronous network of processors connected by delay buffers with occupancies given by the logical latencies.

This suggests the following question; what happens if we have a logical synchrony network where one or more of the edges has a negative logical latency? Using Theorem 6, we know that if a network has nonnegative round-trip times, one can relabel the clocks so that all logical latencies are nonnegative. Hence any physically constructible multiclock network is equivalent to a perfectly synchronous network.

V. THE BITTIDE MECHANISM

We now turn to a specific form of multiclock network which can be implemented on modern networking hardware. In Section IV we have already discussed one of the key components of this, specifically that with each localtick, a node removes one frame from the head of every incoming FIFO, and sends one frame on every outgoing FIFO. However, this is not enough for implementation, since we must ensure that the occupancies of the FIFOs neither underflow nor overflow.

In the bittide model, the FIFO connecting node i to node j is composed of two parts, connected sequentially. The first part is a communication link, which has a latency $l_{i \rightarrow j}$, the number of wall-clock seconds it takes to send a frame across the link. The second part is called the *elastic buffer*. It is a FIFO which is located at the destination node j . Node i sends frames, via the communication link to node j , where they are inserted at the tail end of the elastic buffer. We assume that the communication link cannot reorder frames, and so together the communication link and the elastic buffer behave as a single FIFO.

Each node has an elastic buffer for each of its incoming links. With each clock localtick, it does two things; first, it removes a frame from the head of each of the elastic buffers and passes that frame to the processor core; second, the core sends one frame on each outgoing communication link.

The purpose of this structure is as follows. An implementation of bittide has nodes whose hardware oscillators are adjustable. The elastic buffer occupancies provide information regarding the relative clock frequencies of the node compared to its incoming neighbors. This allows the oscillators to be adjusted in real-time, by each node, based on measurements of the occupancy of the elastic buffers. Off-the-shelf modules are available which provide fine-grained control of the oscillator frequency. Specifically, if we have an edge $i \rightarrow j$, and node i has a lower clock frequency than node j , then the corresponding elastic buffer at node j will start to drain. Conversely, if node i has a higher clock frequency, the elastic buffer will start to fill. Node j can therefore use the occupancy of the elastic buffers to adjust its own clock frequency. If, on average, it's buffers are falling below half-full, the node can reduce its clock frequency, and conversely.

This mechanism was originally proposed in [28]. Further refinements to the implementation were developed in [20], [21], [22]. These papers show that, provided the frequency corrections

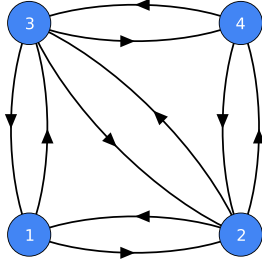


Fig. 8. Graph for bittide simulation.

are chosen appropriately, this mechanism will ensure that elastic buffers never underflow or overflow. A simple mechanism for doing this is to control the *correction*. Adjustable oscillators allow choosing a value for correction c , which causes the frequency ω to become

$$\omega = (1 + \alpha c)\omega^u$$

Here ω^u is the base frequency of the oscillator, which is only known approximately, and α is small, of the order of 10^{-6} . Let $\beta_{i \rightarrow j}$ be the occupancy of the elastic buffer at node j for the link from i to j . Each node j polls the hardware to observe these quantities, and sets the correction at node j to be

$$c = k_p \sum_{i|i \rightarrow j} (\beta_{i \rightarrow j} - \beta_0)$$

where k_p is a positive constant, and the sum is over all links which are incoming to j . The value β_0 is a fixed offset. For an appropriate choice of k_p , all of the the frequencies converge to the same steady-state value. See [20], [21], [22] for more details.

An example simulation of the clock dynamics is in Fig. 8. The time evolution of the clock frequency ω and the buffer occupancy β is shown in Fig. 9, with the buffer occupancy for edge $i \rightarrow j$ labeled i, j . For this simulation, the link latencies $l_{i \rightarrow j} = 1$ ns. Note that in this simulation the parameters are chosen so that the dynamics of the system are clearly visible. In particular, the nodes start at frequencies 1.1, 1.4, 1.8, 2.0 GHz, and in practical hardware systems typically the frequencies at startup would be separated by less than one part in 10^5 . Similarly, the control algorithm parameters are set so that convergence is slow and the equilibrium buffer occupancies are large, between 25 and 75 frames, whereas in practice (e.g., in the hardware of [29]) these parameters are chosen to keep the buffers much smaller. With more realistic parameters the dynamics follow the same general pattern, but are less visible on a plot.

Available implementations: There are three open-source efforts addressing bittide and logically synchronous systems. The first is the hardware description, written in Clash, available at [29]. This may be compiled onto standard FPGA boards, linked to controlled oscillator boards. Second, there is a simulator called *Callisto* [30], which is written in Julia, and simulates the dynamics of the oscillators and the occupancies of the elastic buffers. Finally, there is the *Aegir* simulator [31], written in Rust, which is a functional simulation of a logical synchrony network.

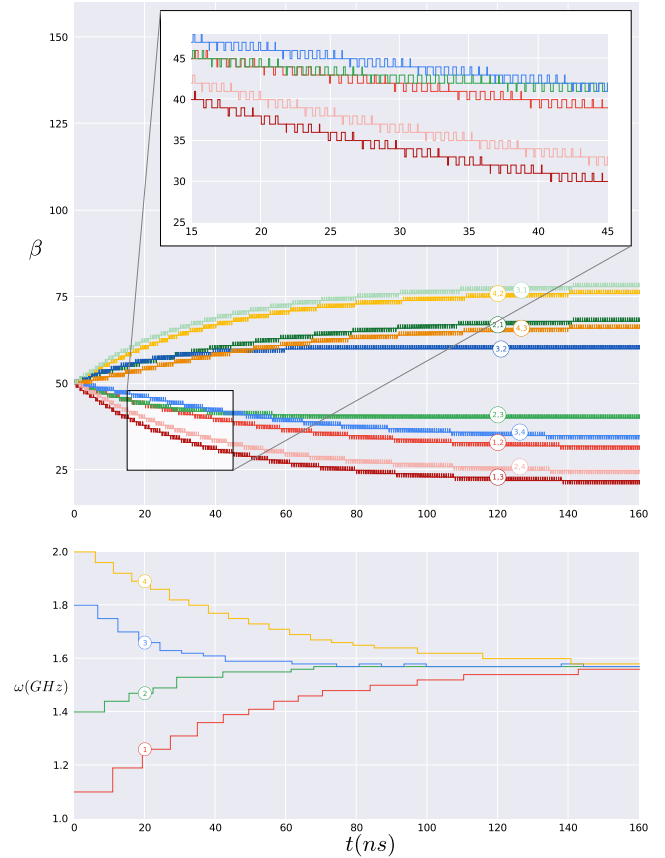


Fig. 9. Occupancy and frequency of the bittide system.

VI. RELATED WORK

The seminal work of Lamport [10] presents a formal framework for clocks in distributed systems, which in particular defined an ordering on a directed graph corresponding to temporal relationships between events, and a global scalar clock which was consistent with that ordering. Subsequent work [24], [25] developed the notion of vector clocks, where each node in a network maintains a vector notion of time which captures exactly the ordering defined by the graph. The synchronization mechanism of bittide was first proposed in [28]. Subsequent works include [20], which developed a mathematical model of the synchronization layer, and [21], which analyzed its performance properties.

Ever since the first distributed systems, synchronous execution has been a gold standard for formal reasoning, provable correctness properties, and ability to express efficient algorithms [32], [33], [34], [35]. As a consequence, the domain of synchronous execution has been studied extensively, in particular in the context of cyber-physical systems. Cyber-physical systems interact with physical processes, and Lee [36] argues that integrating the notion of time in system architecture, programming languages and software components leads to the development of predictable and repeatable systems.

Reasoning about distributed systems has led to the definition of both execution models and parallel programming models.

Kahn Process Networks [37] is one of the most general; while it does not involve time or synchronization explicitly, processes in a Kahn process network communicate through blocking FIFOs, and thus synchronize implicitly through the communication queues. An important distinction between bittide and the Kahn Process Networks is that the former does not make use of blocking.

Synchrony, and its most common representation as a global time reference, led to the definition of multiple models of computation. For example, Synchronous Dataflow [38] enables static scheduling of tasks to resources; Timed Concurrent Sequential Processes (Timed CSP) [39] develop a model of real-time execution in concurrent systems; Globally Asynchronous, Locally Synchronous (GALS) communication models [40] address the issue of mapping a synchronous specification to existing systems which are asynchronous.

Henzinger et al. [41] introduce the concept of *logical execution* and Kopetz et al. [42] introduce Time-Triggered Architectures (TTAs) as a system architecture where time is a first-order quantity and they take advantage of the global time reference to exploit some of the desirable properties of synchronous execution: precisely defined interfaces, simpler communication and agreement protocols, and timeliness guarantees.

Synchronous programming models led to synchronous programming languages, e.g., Esterel [43], Lustre [44], Signal [45], and the development of tools to formally analyze their execution correctness as well as compilers to generate correct synchronizing code for embedded [12] or multicore platforms [14]. This created a virtuous cycle – as researchers understood better properties and embedded them into languages and tools, they drove the adoption of synchronous execution and formal tools for a number of industrial control applications, avionics, and critical system components.

VII. CONCLUSION

This paper has presented the logical synchrony framework. We have shown how this may be used to enable processes on a network of distributed machines to coordinate as if they were synchronized, even if the the clocks on the individual cores are only imperfectly synchronized. We have discussed the bittide mechanism for implementing logical synchrony, how it is abstracted as a multiclock network, and how that corresponds to a further abstraction called the logical synchrony network (LSN). We have analyzed the invariant properties of these networks, and shown how these clocks provide predictable logical latencies on the network.

ACKNOWLEDGMENTS

The ideas for this paper came about through much collaboration. In particular, we would like to thank Nathan Allen, Pouya Dormiani, Chase Hensel, Logan Kenwright, Robert O’Callahan, Chris Pearce, Dumitru Potop-Butucaru, and Partha Roop for many stimulating discussions about this work. Robert had the idea for the proof of Theorem 6.

REFERENCES

- [1] D. Stauffer et al., *High Speed Serdes Devices and Applications*. Berlin, Germany: Springer, 2008.
- [2] F. Ling, *Synchronization in Digital Communication Systems*. Cambridge, U.K.: Cambridge Univ. Press, 2017.
- [3] Precision Clock Synchronization Protocol for Networked Measurement and Control Systems, IEEE Standard 61588–2021, 2021.
- [4] Y. Li et al., “Sundial: Fault-tolerant clock synchronization for datacenters,” in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, Nov. 2020, pp. 1171–1186.
- [5] J. C. Corbett et al., “Spanner: Google’s globally distributed database,” *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 1–22, 2013.
- [6] E. Gupta, P. Goyal, I. Marinos, C. Zhao, R. Mittal, and R. Chandra, “DBO: Fairness for cloud-hosted financial exchanges,” in *Proc. ACM SIGCOMM Conf.*, 2023, pp. 550–563.
- [7] R. Taft et al., “CockroachDB: The resilient geo-distributed SQL database,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 1493–1509.
- [8] S. Bateni et al., “Xronos: Predictable coordination for safety-critical distributed embedded systems,” 2022. [Online]. Available: <https://arxiv.org/abs/2207.09555>
- [9] M. Cowan, S. Maleki, M. Musuvathi, O. Saarikivi, and Y. Xiong, “MSC-CLang: Microsoft collective communication language,” in *Proc. ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2023, pp. 502–514.
- [10] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.
- [11] G. Berry and L. Cosserat, “The esterel synchronous programming language and its mathematical semantics,” in *Proc. Int. Conf. Concurrency*, 1984, pp. 389–448.
- [12] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, “The synchronous languages 12 years later,” *Proc. IEEE*, vol. 91, no. 1, pp. 64–83, Jan. 2003.
- [13] P. S. Roop, Z. Salcic, and M. W. S. Dayaratne, “Towards direct execution of esterel programs on reactive processors,” in *Proc. ACM Int. Conf. Embedded Softw.*, 2004, pp. 240–248.
- [14] K. Didier, A. Cohen, D. Potop-Butucaru, and A. Gauffriau, “Sheep in wolf’s clothing: Implementation models for dataflow multi-threaded software,” in *Proc. Int. Conf. Appl. Concurrency Syst. Des.*, Jun. 2019, pp. 43–52.
- [15] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, “Giotto: A time-triggered language for embedded programming,” *Proc. IEEE*, vol. 91, no. 1, pp. 84–99, Jan. 2003.
- [16] M. Lohstroh, C. Menard, S. Bateni, and E. A. Lee, “Toward a lingua franca for deterministic concurrent systems,” *ACM Trans. Embedded Comput. Syst.*, vol. 20, no. 4, pp. 1–27, May 2021.
- [17] E. A. Lee and M. Lohstroh, “Time for all programs, not just real-time programs,” in *Proc. 10th Int. Symp. Leveraging Appl. Formal Methods, Verification Validation*, 2021, pp. 213–232.
- [18] D. L. Mills, “Internet time synchronization: The network time protocol,” *IEEE Trans. Commun.*, vol. 39, no. 10, pp. 1482–1493, Oct. 1991.
- [19] M. Lipinski, T. Wlostowski, J. Serrano, and P. Alvarez, “White rabbit: A PTP application for robust sub-nanosecond synchronization,” in *Proc. IEEE Int. Symp. Precis. Clock Synchronization Meas., Control Commun.*, 2011, pp. 25–30.
- [20] S. Lall, C. Caçcaval, M. Izzard, and T. Spalink, “Modeling and control of bittide synchronization,” in *Proc. Amer. Control Conf.*, 2022, pp. 5185–5192. [Online]. Available: <https://arxiv.org/abs/2109.14111>
- [21] S. Lall, C. Caçcaval, M. Izzard, and T. Spalink, “Resistance distance and control performance for bittide synchronization,” in *Proc. Eur. Control Conf.*, 2022, pp. 1850–1857. [Online]. Available: <https://arxiv.org/abs/2111.05296>
- [22] S. Lall, C. Caçcaval, M. Izzard, and T. Spalink, “On buffer centering for bittide synchronization,” 2023. [Online]. Available: <https://arxiv.org/abs/2303.11467>
- [23] R. B. Bapat, *Graphs and Matrices*. Berlin, Germany: Springer, 2017.
- [24] C. J. Fidge, “Timestamps in message-passing systems that preserve the partial ordering,” *Australian Comput. Sci. Commun.*, vol. 100, no. 1, pp. 56–66, Feb. 1988.
- [25] F. Mattern, “Virtual time and global states of distributed systems,” in *Proc. Workshop Parallel Distrib. Algorithms*, 1989, pp. 215–226.
- [26] R. E. Tarjan, *Data Structures and Network Algorithms*, Philadelphia, PA, USA: SIAM, 1983.
- [27] C. André, F. Mallet, and R. de Simone, “Modeling time(s),” in *Proc. Int. Conf. Model Driven Eng. Lang. Syst.*, 2007, pp. 559–573.

- [28] T. Spalink, *Deterministic Sharing of Distributed Resources*. Princeton, NJ, USA: Princeton Univ., 2006.
- [29] "Hardware implementation of the bittide system," 2023. [Online]. Available: <https://github.com/bittide/bittide-hardware>
- [30] "Callisto: Simulator of bittide clock synchronization dynamics," 2023. [Online]. Available: <https://github.com/bittide/Callisto.jl>
- [31] "Aegir: Multi-level bittide functional simulator," 2023. [Online]. Available: <https://github.com/bittide/aegir>
- [32] L. Lamport, "The implementation of reliable distributed multiprocess systems," *Comput. Netw.*, vol. 2, no. 2, pp. 95–114, 1978.
- [33] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, 1988.
- [34] B. Liskov, "Practical uses of synchronized clocks in distributed systems," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 1991, pp. 1–9.
- [35] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann, "Polychrony for system design," *J. Circuits, Syst., Comput.*, vol. 12, no. 3, pp. 261–303, 2003.
- [36] E. A. Lee, "Computing needs time," *Commun. ACM*, vol. 52, no. 5, pp. 70–79, 2009.
- [37] G. Kahn, "The semantics of a simple language for parallel programming," *Inf. Process.*, vol. 74, pp. 471–475, 1974.
- [38] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, Jan. 1987.
- [39] G. M. Reed and A. W. Roscoe, "Metric spaces as models for real-time concurrency," in *Proc. Workshop Math. Found. Program. Lang. Semantics*, 1988, pp. 331–343.
- [40] D. Potop-Butucaru, B. Caillaud, and A. Benveniste, "Concurrency in synchronous systems," in *Proc. Int. Conf. Appl. Concurrency Syst. Des.*, 2004, pp. 67–76.
- [41] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Embedded control systems development with Giotto," in *Proc. ACM SIGPLAN Workshop Lang., Compilers Tools Embedded Syst.*, 2001, pp. 64–72.
- [42] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proc. IEEE*, vol. 91, no. 1, pp. 112–126, Jan. 2003.
- [43] G. Berry, "The foundations of Esterel," in *Proof, Lang., Interact.: Essays Honour Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press, 2000, Ch. 14, doi: [10.7551/mitpress/5641.003.0021](https://doi.org/10.7551/mitpress/5641.003.0021).
- [44] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proc. IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep. 1991.
- [45] A. Benveniste, P. Le Guernic, and C. Jacquemot, "Synchronous programming with events and relations: The SIGNAL language and its semantics," *Sci. Comput. Program.*, vol. 16, no. 2, pp. 103–149, 1991.



Sanjay Lall (Fellow, IEEE) received the BA degree in mathematics with first-class honors and the PhD degree in engineering from the University of Cambridge, England, in 1990 and 1995, respectively. He is professor of electrical engineering with the Information Systems Laboratory, Stanford University. His research group focuses on algorithms for control, optimization, and machine learning. From 2018 to 2019 he was director with the Autonomous Systems Group, Apple. Before joining Stanford he was a research fellow with the California Institute of Technology,

Department of Control and Dynamical Systems, and prior to that he was a NATO research fellow with the Massachusetts Institute of Technology, Laboratory for Information and Decision Systems. He was also a visiting scholar with the Lund Institute of Technology in the Department of Automatic Control. He has significant industrial experience applying advanced algorithms to problems including satellite systems, advanced audio systems, Formula 1 racing, the America's cup, cloud services monitoring, and integrated circuit diagnostic systems. He is currently a visiting researcher with Google.



Călin Cașcaval (Fellow, IEEE) received the MS degrees from the Technical University of Cluj, Romania and West Virginia University, USA, and the Ph. degree in computer science from the University of Illinois at Urbana-Champaign, in 2000. He is a director of engineering with Google Research, leading research in scalable distributed systems and compilers. He spent his career in industrial research, where identified industry trends, defined, built, and delivered first of a kind prototypes and products, including: the first programmable networking (P4)

production compiler and networking stack at Barefoot Networks; the first mobile heterogeneous computing runtime and parallel browser, mobile optimized math libraries and power optimization framework at Qualcomm Research; system software for the Blue Gene family of supercomputers and the first UPC compiler to scale to hundreds of thousands of processors at the IBM TJ Watson Research Center.



Martin Izzard (Member, IEEE) received the BS and MS degrees from Natal University in South Africa, and the PhD degree from Cambridge University, England (Trinity College). He joined Texas Instruments in 1991 where he held a variety of technical and general management roles in both the digital and analog divisions. He was instrumental in building the TI custom ASIC business in the Datacom and Telecom markets. He also ran a business building and selling TI Analog ICs (clocks and serdes) into the same networking space. Before leaving in 2013,

he was a TI vice president and responsible for the R&D Labs chartered with creating opportunities for TI in new markets. He was a founder of Barefoot Networks and was its Founding CEO through 2017. He joined Google in 2019 to work on a long-range research project, bittide, which seeks to change the way distributed systems are built. Martin is an alumnus of the Stanford Business School Executive Program (SEP) and was at one time a Stanford EECS visiting scholar.



Tammo Spalink received the BS degree from Carnegie Mellon University, the MS degree from the University of Arizona, and the PhD degree from Princeton University, all in computer science. He has spent his career to date with Alphabet where he has contributed to Android, ChromeOS, Loon, and numerous internal projects. He is currently an engineering director in Google Research responsible for a range of projects from silicon design to machine learning compilation.