IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 18 April 2024, accepted 4 June 2024, date of publication 7 June 2024, date of current version 14 June 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3411017

RESEARCH ARTICLE

Logical Synchrony Networks: A Formal Model for Deterministic Distribution

LOGAN KENWRIGHT[1], PARTHA ROOP[1], NATHAN ALLEN[1], SANJAY LALL[2,3], (Fellow, IEEE), CĂLIN CAŞCAVAL[3], (Fellow, IEEE), TAMMO SPALINK[3], AND MARTIN IZZARD[3]

[1]Department of Electrical, Computer and Software Engineering, The University of Auckland, Auckland 1010, New Zealand
[2]Department of Electrical Engineering, Stanford University, Stanford, CA 94305, USA
[3]Google Research, Mountain View, CA 94043, USA

Corresponding author: Logan Kenwright (logan.kenwright@auckland.ac.nz)

This work was supported by Google Research "unrestricted gift", "Designing Scalable Synchronous Applications over Google bittide", Awarded to the University of Auckland.

ABSTRACT In the modelling of distributed systems, most Model of Computations (MoCs) rely on blocking communication to preserve determinism. A prominent example is Kahn Process Networks (KPNs), which supports non-blocking writes and blocking reads, and its implementable variant Finite FIFO Platforms (FFPs) which enforces boundedness using blocking writes. An issue with these models is that they mix process synchronisation with process execution, necessitating frequent blocking during synchronisation. This paper explores a recent alternative called *bittide*, which decouples the execution of a process from the synchronisation behaviour. Determinism and boundedness is preserved while enabling pipelined execution for better throughput. To understand the behaviour of these systems we define a formal model – a deterministic MoC called Logical Synchrony Networks (LSNs). LSNs describes a network of processes modelled as a graph, with edges representing *invariant logical delays* between a producer process and the corresponding consumer process. We show that this abstraction is satisfied by the KPN model, and subsequently by both the concrete FFPs and bittide architectures. Thus, we show that FFPs and bittide offer two ways of implementing deterministic distributed systems, with the latter being more performant.

INDEX TERMS Distributed systems, models of computation, Kahn process networks, bittide.

I. INTRODUCTION

The modelling of distributed systems relies heavily on MoCs [2], which are used to formally describe how concurrent components of a system are composed, focussing on their computation and the communication. Many MoCs have been developed, which range from non-deterministic models such as process algebras [3], [4] and actor-based models [5] to deterministic variants based on KPNs [6] and synchronous models [7]. In distributed systems, the communication scheme between processes is a defining feature of the chosen MoC. Edwards et al. [8] categorize common communication schemes, which we summarise in Table 1.

The characteristics of these schemes include the number of writers and readers, buffer size, blocking I/O requirements, and whether a piece of data can be read at most once during a communication. Additionally, we introduce some metrics not present in [8]: triggering, referring to the mechanism driving each communication event, and decoupled execution, denoting schemes where synchronization behavior does not interfere with process execution. A summary of common schemes is as follows:

- *Unsynchronized*: No coordination between send/receive, offering no guarantee on data arrival.
- *Read-Modify-Write:* Processes communicate over shared memory, employing locking mechanisms.
- *Unbounded First In First Out (FIFO):* Sender-generated data tokens are consumed by receivers in order, allowing any number of tokens in the intermediate buffer.

The associate editor coordinating the review of this manuscript and approving it for publication was Leimin Wang.

TABLE 1. Communication Schemes for Synchronisation.

Scheme	Writers	Readers	Buffer Size	Blocking Read	Blocking Write	Single reads	Triggering	Decoupled Execution
Unsynchronized	Many	Many	One	No	No	No	None	No
Read-Modify-Write	Many	Many	One	Yes	Yes	No	Event	No
Unbounded FIFO	Single	Single	Unbounded	Yes	No	Yes	Event	No
Bounded FIFO	Single	Single	Bounded	Yes	Yes	Yes	Event/Schedule	No
Rendezvous	Single	Varies	One	Yes	Varies	Yes	Event	No
Time Triggered	Single	Varies	One/Bus-based	No	No	Yes	Physical Clock	No
Elastic Buffer [1]	Single	Single	Bounded	No	No	Yes	Logical Clock	Yes

- *Bounded FIFO:* Similar to unbounded FIFO, but with a finite buffer capacity. A sender cannot progress without free space available.
- *Rendezvous:* Processes synchronize through explicit messaging, potentially causing blocking until matching read/write events occur.

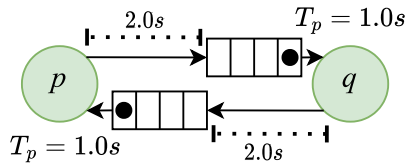
In addition, we note the following schemes which have been developed since [8]:

- *Time triggered:* Synchronization based on known timing bounds, utilizing either physical or logical clocks [9].
- *Elastic Buffer:* A novel mechanism found in bittide [10]. Provides bounded FIFO behaviour without blocking reads or writes, by varying communication speed using a control mechanism.

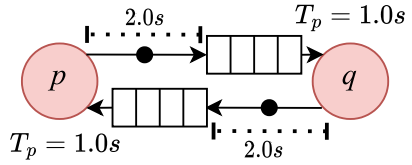
Among these schemes, deterministic models, particularly FIFO-based models, hold significance for designing safety-critical systems. Kahn's seminal work formalises KPN models [6], where a producer never blocks, while a consumer blocks while accessing an empty FIFO. When a consumer blocks on a given FIFO, it is prevented from context switching and hence this model is shown to be *determinate* [6], i.e. for any arbitrary execution order of the processes, the order of tokens in the buffers remains invariant. However, the scheduling problem using bounded memory is undecidable. Hence, variants such as FFPs [11] introduce blocking on both the producer side (when the FIFO is full) and consumer side (when the FIFO is empty). We observe that for synchronisation, existing deterministic models either use physical time or intertwine process execution with synchronisation, leading to undesirable blocking inefficiencies. How can this blocking time be mitigated?

A. MOTIVATION

Consider two machines, shown as circles in Figure 1a, which communicate over a bounded FIFO, shown as boxes. In order to perform a cycle of execution, each machine must consume a token from its input buffer, and produce a token on the output link. Each machine has an execution time of 1 second, meaning that each machine can nominally execute once per second. However, these two machines are separated by a lengthy transmission delay of 2 seconds, and have only a single initial token in each buffer. Consequently, more time is spent waiting for an in-flight token to arrive (Figure 1b) than the time spent doing useful execution. Thus, the effective execution time of both machines is 3.0 seconds.

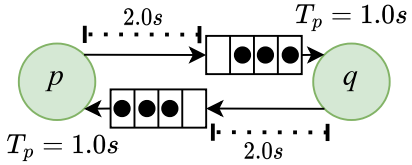


(a) Tokens available in the buffers, machines are live (green)

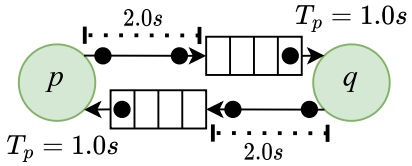


(b) Tokens are in-flight, machines are not live (red)

FIGURE 1. A simple FFP. Each machine has an execution time of 1 second and each link a delay of 2.0 seconds.



(a) Tokens are available in the buffers, machines are live



(b) Tokens are in-flight and in the buffer, machines are live

FIGURE 2. The example from Figure 1, with the initial marking changed to 3 tokens.

Now consider that the number of initial tokens in each buffer is increased to three, shown in Figure 2. There are now sufficient tokens in circulation such that tokens are available to be consumed from each buffer at any time during execution. Consequently the transmission delay no longer causes unnecessary blocking, and both machines can operate at their nominal rate of 1.0 second.

The optimal number of tokens (initial marking) is trivial to find in this example due to its symmetry. However, in the general case the differing execution rates of machines will always mandate some blocking. In these cases, the smallest marking resulting in maximum network performance is a special case of a maximum-flow circulation problem [12], which is NP-hard.

Introducing additional tokens is not behaviour-preserving for the application which is executing on each machine. In this case, the token-pushing systems are being used to model network-level synchronisation, where the contents of each token is indistinguishable (much like a petri net). The application model operates at a higher level of abstraction which will be covered in future works. The relationship between initial marking and throughput is well-known and not novel to this work. However, this pipelining property forms the basis of how the bittide model operates.

B. THE BITTIDE APPROACH

Recently, Google has developed a physical layer protocol called bittide [10]. Like FFPs bittide also uses point to point FIFO buffers (called *elastic buffers*) for communication.

bittide makes use of token pipelining in the fashion described above, making it both performant and deterministic. However, in bittide, the computation of a process (its execution model) is decoupled from its synchronisation model with other processes, such that blocking should never occur. Each process executes using its local clock. When the clock ticks, the sender process writes to its elastic buffer. Likewise, when the consumer clock ticks, a token is removed from its FIFO buffer.

It is obvious that mismatching frequencies of the producer and consumer will cause buffer overflow / underflow. However, in bittide, this is handled by distributed controllers at each process. The controllers examine the local buffer at a process to determine the relative speed of the neighbours. Using this as a feedback signal, appropriate controllers are designed, which ensure that frequencies of all processes stabilise to a common frequency, ensuring buffer boundedness.

This protocol has been shown to be both deterministic and performant. However, bittide is still very recent and hasn't been thoroughly examined either formally or experimentally. This paper aims to provide a formal model for bittide, and to compare it with existing models of computation.

II. RELATED WORK

The most common approach to distributed process synchronisation is through physical time protocols. Notably, the Network Time Protocol (NTP) [13] synchronises physical clocks across a network to a common time source. More precise protocols in this category include the Precision Time Protocol (PTP) [14] and the recent IEEE 802.1AS [15] standard, which requires hardware compatibility. However, none of these actually describe how processes should synchronise, only how they should agree on a common time. The recent DetNet [16] standard builds over these precision protocols to ensure the deterministic delivery of data over a network, but still at a much lower level than process synchronisation.

The family of Time Triggered Architectures (TTAs) [17] provide a mechanism for process synchronisation based on physical time. These are commonly used for safety-critical industrial systems and provide both clock synchronisation and a mechanism for process synchronisation. However, their initial design complexity is quite high, and relies on a well-defined hardware structure such that they are not easily adapted for general-purpose distributed systems.

At a higher level of abstraction, the coordination language Lingua Franca [18] enables both physical time and process synchronisation over arbitrary hardware. Its physical synchronisation protocol is based on the earlier Ptides [19]. Using known latency bounds, each process can locally decide a "safe-to-process" time to proceed. When these bounds are violated, the system flags a fault and invokes a handler. While robust, Lingua Franca primarily focuses on the mixture of logical and physical time to ensure the safe execution of safety-critical system, rather than our logical synchrony approach of abstracting away physical time entirely.

Lamport clocks, and the related vector clocks, are the most famous logical time protocols [20]. These logical clocks ensure that the order of events is consistently agreed on by all processes. Programmers may implement algorithms using these clocks to ensure process synchronisation, but that is still ultimately up to the designer to implement.

Loosely Time Triggered Architectures (LTTAs) [21] describe a family of protocols, which preserve synchronous semantics over quasi-periodic architectures. LTTA protocols are typically described at a much higher level than their namesake TTAs, focusing more on process-level communication. While the specific protocol varies, all designs synchronise logical tick progression through the use of explicit messaging. As a result, throughput degrades with higher transmission delays. In general, the communication delays are assumed to be small relative to the task duration.

Globally Asynchronous Locally Synchronous (GALS) models are those which preserve synchronous semantics within a single machine, but communicate over an asynchronous medium. The term GALS itself is very broad, often describing any system which combines synchronous and asynchronous components, but also often specific programming paradigms [22]. Usually such designs introduce non-determinism at a system level. Determinism is often enforced over a GALS architecture through the synthesis of wrappers which stall a process when an unresolved dependency is reached. This is commonly employed in the Polychrony [23] toolkit of the SIGNAL synchronous language [24]. A similar approach was employed by the Esterel tool 'ocrep' [25]. These processes which remain behaviourally correct whether distributed over a synchronous or an asynchronous medium are designated *endochronous*. Determining endochrony in general is a difficult task, and if frequent synchronisations are required then the system faces the same performance pitfall as LTTAs when lengthy transmission delays are present.

In short, various approaches to process synchronisation exist within a variety of contexts. However, these are rarely general-purpose and often require significant designer input either at the hardware or software level.

We make the following contributions for the design of deterministic and performant distributed systems:

1) We formalise LSNs, recently introduced in [1], in the context of deterministic MoCs for distributed systems.
2) We show KPNs are a LSN instance.
3) We show that bittide [10] is also a realisable LSN instance. We introduce a variant of FFPs called Logically Synchronous FIFO Platforms (LSFPs) for making a fair comparison with bittide (see Section V-B).
4) We present an empirical evaluation of the performance of LSFPs and bittide, and we show that bittide is more performant in the average case.

This paper is organized as follows: in Section III, we introduce our definition of LSNs, adapted from [1]. Two realisable implementations are demonstrated. In Section IV we show that the well-known KPN model can be used to express LSNs. In Section V we show how FFPs implement KPNs, and thus propose LSFPs as a restriction which implements the LSN model efficiently. Similarly, the recent physical-layer protocol bittide is shown to express LSNs in Section VI. Then, Section VII compares the performance of FFPs relative to bittide. Finally, Section VIII makes concluding remarks including future directions.

III. LOGICAL SYNCHRONY NETWORKS

We provide a generic, graph-based abstraction of a distributed system comprising a network of machines $\mathscr{M}_i$, denoting the $i-th$ machine, which labels the vertex v_i. Furthermore, we formalise the LSN model in this paper, which was introduced briefly in [1] but not formalised as a MoC, and we show that existing models of computations exhibit the LSN property of *invariant logical delays*, especially those based on KPNs.

Machines execute at discrete points called events or ticks. Each machine has its own notion of ticking, which may be different from other machines. The event count $\theta_i \in \mathbb{N}$, corresponding to the machine $\mathscr{M}_i$ increases by 1 every logical tick. The defining feature of LSNs is the **invariant logical delays**, meaning there always exists an invariant offset between the event count of any production events and their associated consumption events along a channel.

An event is denoted $(\mathscr{M}_i, \theta_i)$, where $\mathscr{M}_i$ is the machine experiencing the event when θ_i is its event counter. We say that two events $(\mathscr{M}_i, \theta_i) \prec (\mathscr{M}_j, \theta_j)$ iff $\theta_j - \theta_i = \lambda_{i\to j}$. At each event, a token is read from each input edge and a token is produced to each output edge. The event count $\theta_i \in \mathbb{N}$ increases by 1.

Definition 1: A LSN is a tuple $< G, \Theta, \lambda >$, where:

- *G=$<V, E>$ is a digraph where V denotes a set of vertices, and $E \subseteq V \times V$ denotes a set of edges such that $v1 \neq v2$ for all $(v1, v2) \in E$.*
- *Each vertex $\mathscr{M}_i \in V$ corresponds to a machine executing a synchronous program that generates an event every logical tick. $\theta_i \in \Theta$ represents the event counter for the machine $\mathscr{M}_i$ and the value of $\theta_i \in \mathbb{N}$. As the next event is generated by a machine as its ticks, its event counter is incremented by 1.*
- *Edges are labelled with the logical delay mapping $\lambda : E \to \mathbb{Z}$. Moreover, $\lambda((\mathscr{M}_i, \mathscr{M}_j)) = \lambda_{i\to j}$ implies the following relation holds between the events associated with this edge: $(\mathscr{M}_i, \theta_i) \prec (\mathscr{M}_j, \theta_j)$. Consequently, $\theta_j - \theta_i = \lambda_{i\to j}$.*

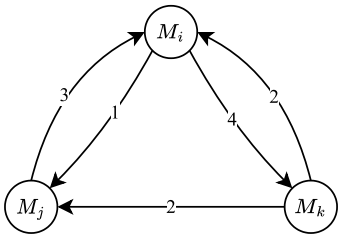


FIGURE 3. A three-node LSN.

Example 1: An example LSN is shown in Figure 3, consisting of three machines M_i, M_j, M_k. An example edge is between (M_k, M_j), which is labelled with a value 2, which indicates the invariant logical delay, denoted $\lambda_{k\to j}$, between production of some event at M_k and consumption at M_j at any tick of the two machines when the production and the corresponding consumption happens. For this edge $\lambda_{k\to j} = 2$.

The defining feature of LSNs is the *invariant logical delays*, meaning there always exists an invariant offset between the event count of any production events and their associated consumption events along a channel. The same invariance applies over both edges and paths. Multiple paths, with different cumulative logical delays, may exist between any two machines. $\lambda_{i\to j}$. This does **not** imply that the machines have any relationship between event counts at any one physical time when observed by a global entity; any amount of physical time may have elapsed between the two events. Logical delays are defined over the set of integers, including negative values. Whether this is used in a physical system is an implementation detail. Lall et. al [1] define an equivalence class of LSNs, and in doing so show that all LSNs with negative edge weights have at least one equivalent all-positive LSN.

Example 2: Figure 4 shows an event sequence for the LSN in Figure 3 with a potential execution trace of the three machines and their relationship in both logical and physical time. The three machines are desynchronised in their event timings in physical time. However, the logical time offset remains invariant.

Here we show that LSNs, as a model, are deterministic: In order to reason about LSN behaviours, we start by abstracting machines as mathematical functions. Each machine $\mathscr{M}_i \in V$ has a corresponding function $f_i \in \mathscr{F}$, where $\mathscr{F}$ is the set of all machine functions, which can be described as a synchronous Mealy machine:

$$f_i : X^n \times S_i \to X^m \times S_i \tag{1}$$

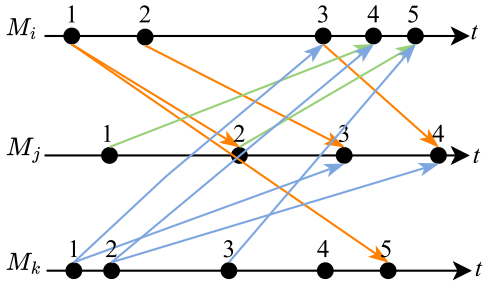


FIGURE 4. The execution trace of the three-machine LSN (Figure 3), as measured by a global observer.

where X^n denotes a vector of n inputs and X^m denotes a vector of m output values from that machine. S_i denotes the set of states for $\mathscr{M}_i$. The function takes in all current inputs and the current state, and produces some outputs and the next state.

Definition 2: A LSN execution forms a Complete Partial Order (CPO) of events, called the extended graph $G_{ext} = <V_{ext}, E_{ext}>$.

- V_{ext} *is the set of all machine events in the network, described by* $V_{ext} = \{(\mathscr{M}_i, n) | \mathscr{M}_i \in V, n \in \theta_i\}$
- E_{ext} *describes the directed edges showing dependencies between events, which are of one of two types.*
  1) *Edges between events at different machines* $(\mathscr{M}_i, n) \rightarrow (\mathscr{M}_j, m)$ *are called communication edges, corresponding to an edge* $(\mathscr{M}_i, \mathscr{M}_j) \in E$ *in the LSN. The clock difference* $m-n$ *is the associated logical delay* $\lambda_{i \rightarrow j}$
  2) *Edges between successive events at the same machine* $(\mathscr{M}_i, n) \rightarrow (\mathscr{M}_i, n+1)$ *are called computation edges, which capture monotonicity of local machine events.*

Due to the condition on LSNs that all cycles in the graph are positive, the corresponding ordering relation is monotonic and acyclic by construction. G_{ext} for the example shown in Figure 3 is demonstrated in Figure 5. We assume a finite execution, meaning there is a unique start-up event $\perp$, which precedes the first event at each machine, and termination event $\top$, which succeeds the final event at each machine.

Each event is associated with a corresponding function instantiation, where f_i^0 represents the first invocation of machine $\mathscr{M}_i$ at event $(\mathscr{M}_i, 0)$, and f_i^{k-1} the *kth* invocation at $(\mathscr{M}_i, k-1)$. The input and output edges of an event form the input and output vectors respectively and the state corresponds to the event. Naturally, an event cannot receive logically delayed input from a non-existing negative event, so to accommodate missing inputs we introduce initial conditions within $\perp$ containing the pre-calculated inputs to the initial function invocations.

Determinism has a variety of different interpretations in literature as summarised in [26]. In this context we interpret it as follows:

Definition 3: A LSN system is determinate if for any two or more executions with consistent initial conditions, the resulting output traces are consistent, where:

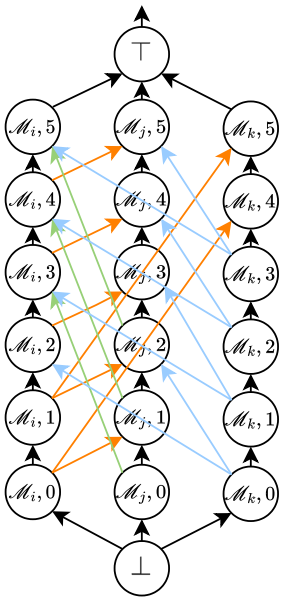


FIGURE 5. G_{ext} of the three-machine LSN (Figure 3). Computation edges shown in black and communication edges coloured to differentiate source machine.

- *The output derived from the initial condition consists of the function instance applied to the initial inputs* $\perp$.
- *An output trace for a single machine consists of the sequence of outputs from each function instance* $\langle \mathbf{f^i} \rangle = \langle f_i^0 :: f_i^1 :: \ldots :: f_i^{n-1} \rangle$ *where* $::$ *is the concatenation operator.*

Lemma 1: Consider any event $(M_i, n) \in V_{ext}$. *The value of any edge (incoming/outgoing) is determinate.*

Proof: Any edge is a composition of functions and hence determinate. ■

Theorem 1: LSN execution is always determinate.

Proof: Based on Lemma 1 and by induction on the depth k of the extended graph corresponding to the LSN. Note that any event is only dependent on events that happened prior to it. ■

Of course, such guarantees are only valid for the abstract model of LSNs. In practice, the implementation of the machines must be proven to conform to the abstract model. Specifically, we show this for the KPN and bittide implementations in the following sections.

IV. KPN AS AN INSTANCE OF LSN

KPNs are a well-known model of computation for deterministic systems. We show that this existing MoC satisfies our LSN model. KPNs model a network of machines communicating over unbounded FIFO channels. Each machine in a KPN executes over discrete firings, analogous to LSN events, where at each firing a token is consumed from every inbound FIFO, and a token pushed onto each outbound FIFO.

Definition 4: A KPN is denoted as a graph $G' =< V', E' >$ *and a labelling function* $\alpha : E' \rightarrow B$ *where any* $b \in B$ *is an unbounded FIFO queue. Each buffer has an associated occupancy of tokens given by* $\beta : B \times \mathbb{R} \rightarrow \mathbb{N}$,

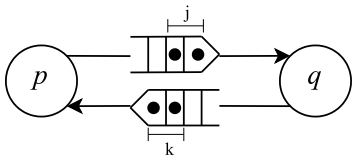


FIGURE 6. A cyclical KPN, with some initial tokens in each directional unbounded FIFO.

where $\mathbb{R}$ *is the current physical time and* $\mathbb{N}$ *the count of tokens in the FIFO.*

A basic two-vertex KPN is shown in Figure 6, with the initial occupancies $\beta(\alpha(p,q),0) = j$ and $\beta(\alpha(q,p),0) = k$. A KPN with cyclic components will immediately deadlock if there are no initial tokens. Every directed cycle in a graph must always have at least one token to be live. We call these values the *initial marking*.

KPNs alone do not describe when a machine may fire, so we introduce a set of *firing rules* which dictate firing times. Numerous firing orders may be possible with different temporal interleaving of events, as long as FIFOs do not underflow. Conventionally this is avoided through use of blocking channel reads.

By observation of the graphical similarities between LSNs and KPNs, and the effect of initial FIFO markings on delays, we can deduce a simple transformation:

Observation 1: Given a KPN we can produce an equivalent LSN using the following algorithm:

1) For the KPN graph $G'=<V', E'>$, form a LSN graph $G=<V, E>$. For every $v' \in V'$, there exists a $v \in V$, and for each $e' \in E'$, there exists an $e \in E$.
2) Label each edge $(p,q) \in E$ with the logical delay $\lambda_{p\to q} = \beta_{p\to q}(0)$, equal to the initial occupancy of the FIFO in the original KPN.

Consider the FIFO in Figure 6. There exist j initial tokens from $p \to q$. As a result, the first token produced at p will be consumed by q on its $(j+1)_{th}$ firing, as it must first await all initial tokens to be consumed.

Observation 2: The logical delay $\lambda_{p\to q}(t)$ *at any time* t *consists of the FIFO occupancy* $\beta_{p\to q}(t)$*, plus the fire count difference* $\theta_q(t) - \theta_p(t)$*.*

$$\lambda_{p\to q}(t) = \beta_{p\to q}(t) + \theta_q(t) - \theta_p(t)$$

Lemma 2: The logical delay $\lambda_{p\to q}(t)$ *is invariant for all physical wall-clock times* t*:*

Proof:

Case 1: $t = 0$*:*

$$\lambda_{p\to q}(0) = \beta_{p\to q}(0) + \theta_q(0) - \theta_p(0)$$
$$\lambda_{p\to q}(0) = \beta_{p\to q}(0) + 0 - 0 = \beta_{p\to q}(0)$$

Case 2: p *fires after time* τ_p*, producing a token*

$$\beta_{p\to q}(t+\tau_p) = \beta_{p\to q}(t) + 1$$

The firing count of p *increases* :

$$\theta_p(t+\tau_p) = \theta_p(t) + 1$$
$$\lambda_{p\to q}(t+\tau_p) = (\beta_{p\to q}(t)+1) + (\theta_q(t)) - (\theta_p(t)+1)$$
$$\lambda_{p\to q}(t+\tau_p) = \beta_{p\to q}(t) + \theta_q(t) - \theta_p(t)$$
$$\therefore \lambda_{p\to q}(t+\tau_p) = \lambda_{p\to q}(t)$$

Case 3: q *fires after time* τ_q*, consuming a token*

$$\beta_{p\to q}(t+\tau_q) = \beta_{p\to q}(t) - 1$$

The firing count of q *increases* :

$$\theta_q(t+\tau_q) = \theta_q(t) + 1$$
$$\lambda_{p\to q}(t+\tau_q) = (\beta_{p\to q}(t)-1) + (\theta_q(t)+1) - (\theta_p(t))$$
$$\lambda_{p\to q}(t+\tau_q) = \beta_{p\to q}(t) + \theta_q(t) - \theta_p(t)$$
$$\therefore \lambda_{p\to q}(t+\tau_q) = \lambda_{p\to q}(t)$$

Case 4: p *and* q *both fire simultaneously*

Linear combination of 2 *and* 3*, invariance holds.*

■

The value of the logical delay remains invariant; the composition of this invariant value is exchanged between the buffer occupancy and the relative difference of fire counts. Henceforth we drop the time variable from the invariant λ_{pq}.

Theorem 2: Every KPN is a LSN.

Proof: Follows from Observation 1 and Lemma 2 ■

There are difficulties mapping a KPN model to a physical implementation due to the unbounded FIFOs. Synchronous Data Flow (SDF) [27] overcomes this limitation by generating a schedule of predetermined production/consumption events, but implementing efficient scheduling on distributed devices is challenging. Here we present two concrete implementations of LSNs, each with a different mechanism to bound buffer sizes.

V. FINITE FIFO PLATFORMS

FFPs are a realisation of the KPN model, extended with blocking writes to prevent unbounded growth, first introduced in [11]. We adopt them here as an example of an implementable LSN model from existing literature. A process can only fire if tokens are available for consumption at every input, and if there is space to emit tokens at every output. Otherwise it 'stutters', and skips that firing cycle. Figure 7 shows a FFP with a single initial token.

As a skipped cycle does not affect the semantics of a KPN, it is acceptable to stutter at any time. Just as in the pure KPN model, $\lambda_{p\to q}$ is determined by the number of initial tokens from p to q to three.

In this approach, buffer boundedness is enforced by locating the intermediate buffer on the receiver side of any communication channel. Blocking reads are trivial to implement by checking the local buffer occupancy. Blocking writes are enforced by using a heuristic to conservatively estimate

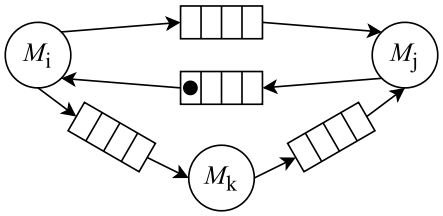


FIGURE 7. A three-machine FFP where one token is placed in the unit delay buffer to ensure liveness.

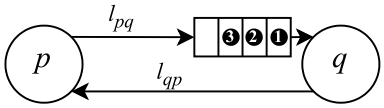


FIGURE 8. A buffered communication link from producer *p* to consumer *q*, with a back-pressure link from *q* to *p*.

remote buffer occupancy. This heuristic is mentioned in [11] but explained in greater detail here.

A. EFFICIENT FULLNESS CHECKING

Consider two machines p and q, separated by a transmission link with latency l_{pq}. The real occupancy of some buffer $\beta_{real}(t)$ at time t is equal to the number of tokens that have arrived over the link from a producer p, minus the number consumed at q, plus the initial occupancy:

$$\beta_{real}(t) = \theta_p(t - l_{pq}) - \theta_q(t) + \beta(0) \tag{2}$$

The producer p cannot know the current value of $\theta_q(t)$ nor the number of its own tokens that have reached the remote buffer $\theta_p(t - l_{pq})$. To provide this information, we include an unbuffered reverse link from consumer to producer. Figure 8 demonstrates this arrangement.

Over this reverse link p measures the delayed firing count $\theta_q(t - l_{qp})$. p can also measure its own current firing count $\theta_p(t)$. The difference between these two counts is an estimate of the worst-case number of tokens that may still be in transit between the two machines, thus by substituting both of these into Equation (2):

$$\begin{aligned} \theta_p(t) &\geq \theta_p(t - l_{pq}) \\ \text{and } \theta_q(t - l_{qp}) &\leq \theta_q(t) \\ \therefore \beta_{true}(t) &\leq \theta_p(t) - \theta_q(t - l_{qp}) + \beta(0) \end{aligned} \tag{3}$$

Thus we can define a fullness checking function at p which always gives a conservative occupancy estimate. This flow is given by the sequence diagram shown in Figure 9. The overestimation error is proportional to latency, so the peak buffer occupancy will be less than the full capacity.

B. MODIFIED FFPS FOR ENHANCED THROUGHPUT

One limitation of the FFP model is that our performance degrades as we introduce communication delay, because the tokens required to progress spend some time in-flight. As discussed earlier, if a system has more initial tokens in circulation processes can execute more often due to pipelining. At some token count we reach a saturation the

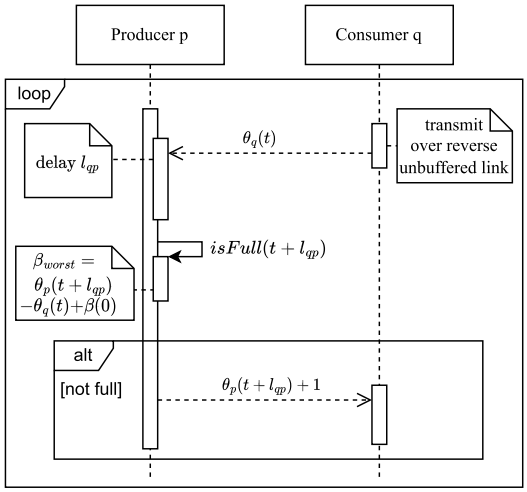


FIGURE 9. Sequence of isFull() FFP behaviour.

throughput does not increase with additional tokens because the communication link is fully populated with in-flight tokens. Such is the case in the bittide model, where the link is assumed to always be fully populated. As a result, a bittide system may have large logical delays (λ), but no blocking is required while awaiting inputs. Here we take inspiration from the delay masking of the bittide model and apply it to the FFP model, and introduce a special class of FFPs which we denote as LSFPs:

Definition 5: A LSFP is a special case of a FFP where the initial marking in each buffer is governed by a heuristic as follows used to enhance throughput:

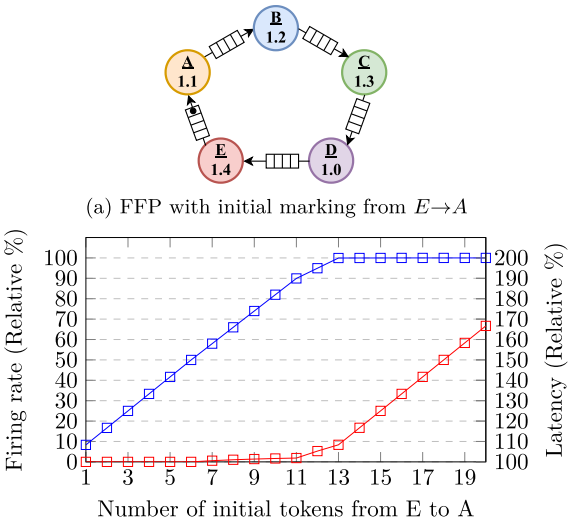
- *For an edge $(\mathscr{M}_i, \mathscr{M}_j) \in E$ with some transmission time $l_{i\to j}$ and a consumer frequency ω_j, the initial occupancy $\beta_{i\to j}(0) = \frac{l_{i\to j}}{\omega_j}$*

The increase in throughput as token count increases is demonstrated for illustrative purposes in Figure 10 for a simulated FFP (see Section VII). This topology was selected arbitrarily, but the expected trend is not topology-specific. Note that the physical response time is not meaningfully worsened until after the saturation point. Now, each process becomes much closer to free-running execution. Consequently, predictable execution is gained with minimal impact to efficiency compared to an asynchronous system. These logical delays may be used in interesting ways at an application level, discussed in future works.

VI. bittide

bittide [10], [28] is a recent physical-layer protocol for distributed computing, developed at Google Research. Bittide serves as the inspiration for this more abstract LSN MoC. Inspired by elastic circuits [29], variations in frequency are absorbed by an intermediate FIFO to maintain syntony.

In a bittide network, a communication link corresponding to an edge in a LSN between some $\mathscr{M}_i$ and $\mathscr{M}_j$ is implemented by the following:



(a) FFP with initial marking from $E \rightarrow A$

(b) Firing rate in blue (higher is better) and latency in red (lower is better) versus initial marking

FIGURE 10. A ring FFP network with frequencies labelled on each node. Links have a delay of 2.0 seconds.

1) A physical communication link, holding a number of in-flight frames called link occupancy $\gamma_{i\rightarrow j}$
2) A FIFO co-located at the receiver with current occupancy $\beta_{i\rightarrow j}$

Periodically, a clock at a machine $\mathscr{M}_i$ will 'tick', increasing its logical clock count θ_i by one.

$$\theta_i = \{0, 1, 2 \ldots\} \in \mathbb{N}$$

At each tick, a machine consumes a frame (token) from each inbound FIFO, and emits a frame on each outbound link. Each link will at any point hold some frames in flight, denoted by the link occupancy. When a frame of data arrives at $\mathscr{M}_j$ from the inbound link it appends to the tail of the recipient FIFO. It will be later consumed by the receiver machine once it propagates to head of the queue after $\beta_{i\rightarrow j}$ receiver ticks.

Compared to a KPN approach, the main feature of bittide is that each machine remains completely free running, rather than blocking. This is possible due to a dynamic clock control which balances frequencies.

A. CLOCK CONTROL SYSTEM

Typically, bittide implementations assume bidirectional links, each edge in the LSN therefore forming a circuit of token flow. Figure 11 shows the communication loop between two machines $\mathscr{M}_i$ and $\mathscr{M}_j$ in a network.

The current occupancies of a machine's buffers are used as a feedback signal to determine clock rate relative to its neighbours, and thus a clock control policy can be applied to take corrective action and stabilise the buffer occupancy within reasonable bounds. The control system is designed with the goal of preventing buffer overflow or underflow, but does not impact the logical synchrony property (so long as frames aren't lost). Various control policies may be valid for a given network topology, such as using a proportional

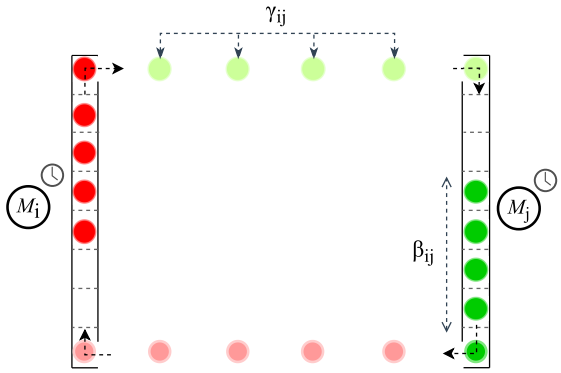


FIGURE 11. Interaction between two bittide machines, demonstrating the FIFO and link action.

controller [10], a proportional-integral controller [28], and a novel "reframing" controller [30]. Bidirectional links are not a hard requirement for clock control, but have so far been assumed in previous works.

The free-running nature of bittide is that frames are assumed to always be in circulation. Unlike in the KPN model with blocking reads, a bittide system should not stall while awaiting a dependent signal. There must be sufficient initial frames, and thus sufficiently deep logical delays, such that the elastic buffer is never completely exhausted while awaiting inbound frames.

B. LOGICAL DELAY INVARIANCE

We show that the bittide architecture satisfies our definition of LSNs.

Observation 3: A frame emitted from $\mathscr{M}_i$ at time t will arrive at $\mathscr{M}_j$ at time $t+\tau_{i\rightarrow j}(t)$ after all preceding frames on the path.

$\tau_{i\rightarrow j}(t)$ represents the physical time elapsed up to consumption, consisting of link delay and the time spent queuing through the FIFO. One frame is consumed per logical tick, therefore the clock θ_j will also have increased by the number of frames at t on the $i \rightarrow j$ path, consisting of the link and the buffer:

$$\theta_j(t + \tau_{i\rightarrow j}(t)) = \theta_j(t) + \gamma_{i\rightarrow j}(t) + \beta_{i\rightarrow j}(t) \tag{4}$$

Equation (4) describes the evolution of the consumer clock during one transmission. Let's re-write this to include an expression for the producer clock θ_i:

$$\text{let } \Delta\theta_{i\rightarrow j}(t) = \theta_i(t) - \theta_j(t)$$

$$\text{then from Equation (4):}$$

$$\theta_j(t + \tau_{i\rightarrow j}) = \theta_i(t) + \Delta\theta_{i\rightarrow j}(t) + \gamma_{i\rightarrow j}(t) + \beta_{i\rightarrow j}(t) \tag{5}$$

Collect all terms except for the producer and consumer clock terms to define a relationship between them:

$$\theta_j(t + \tau_{i\rightarrow j}) = \theta_i(t) + \underbrace{\gamma_{i\rightarrow j}(t) + \beta_{i\rightarrow j}(t) + \Delta\theta_{i\rightarrow j}(t)}_{\lambda_{i\rightarrow j}(t)}$$

$$\theta_j(t + \tau_{i\rightarrow j}) = \theta_i(t) + \lambda_{i\rightarrow j}(t) \tag{6}$$

Thus, we define a logical delay $\lambda_{i\to j}(t)$, describing the offset between the clock at $\mathscr{M}_i$ when a frame is produced and the clock at $\mathscr{M}_j$ when consumed. To satisfy the LSN definition, we show that $\lambda_{i\to j}(t)$ is invariant. $\lambda_{i\to j}(t)$ is invariant if the difference equation $\lambda'_{i\to j}(t) = (\lambda_{i\to j}(t + \tau_{i\to j}) - \lambda_{i\to j}(t))$ is 0 for all send-receive physical time pairings:

$$\begin{aligned} &\text{from (6): } \lambda'_{i\to j}(t) = \gamma'_{i\to j}(t) + \beta'_{i\to j}(t) + \Delta\theta'_{i\to j}(t) \\ &\text{prove } \gamma'_{i\to j}(t) + \beta'_{i\to j}(t) + \theta'_j(t) - \theta'_i(t) = 0 \end{aligned} \tag{7}$$

Observation 4: Line occupancy is incremented during a producer tick, and is decremented when arriving at recipient FIFO.

The line occupancy $\gamma_{i\to j}$ is equal to the number of frames that have been pushed onto the line by $\mathscr{M}_i$, minus the number that have arrived at the buffer, plus the initial condition. We separate these two terms as the cumulative number pushed $\gamma^{in}_{i\to j}$ and the cumulative number received $\gamma^{out}_{i\to j}$

$$\begin{aligned} \text{let } \gamma_{i\to j}(t) &= \gamma^{in}_{i\to j}(t) - \gamma^{out}_{i\to j}(t) + \gamma_{i\to j}(0) \\ \text{then difference } \gamma'_{i\to j}(t) &= \gamma'^{\,in}_{i\to j}(t) - \gamma'^{\,out}_{i\to j}(t) \end{aligned} \tag{8}$$

As one frame being added to the link corresponds to one logical tick at $\mathscr{M}_i$, we can say that:

$$\begin{aligned} \gamma'^{\,in}_{i\to j}(t) &= \theta'_i(t) \\ \text{from (8) } \gamma'_{i\to j}(t) &= \theta'_i(t) - \gamma'^{\,out}_{i\to j}(t) \\ \text{i.e. } \theta'_i(t) &= \gamma'_{i\to j}(t) + \gamma'^{\,out}_{i\to j}(t) \end{aligned} \tag{9}$$

Observation 5: Buffer occupancy $\beta_{i\to j}$ decreases when θ_j ticks, and increases when a frame pops off the link.

$$\begin{aligned} \beta'_{i\to j}(t) &= \gamma'^{\,out}_{i\to j}(t) - \theta'_j(t) \\ \textit{i.e. } \theta'_j(t) &= \gamma'^{\,out}_{i\to j}(t) - \beta'_{i\to j}(t) \end{aligned} \tag{10}$$

We now have expressions for $\theta'_i(t)$ and $\theta'_j(t)$.

Lemma 3: Logical delay $\lambda_{i\to j}(t)$ is invariant for all t:

Proof:

Substitute expressions (9),(10) into (7):

$$\begin{aligned} \lambda'_{i\to j}(t) &= \gamma'_{i\to j}(t) + \beta'_{i\to j}(t) \\ &\quad + (-\beta'_{i\to j}(t) + \gamma'^{\,out}_{i\to j}(t)) - (\gamma'_{i\to j}(t) + \gamma'^{\,out}_{i\to j}(t)) \\ &= \gamma'_{i\to j}(t) + \beta'_{i\to j}(t) + (-\beta'_{i\to j}(t)) - (\gamma'_{i\to j}(t)) \\ &= 0 \end{aligned} \tag{11}$$

■

Due to the inclusion of an initial link occupancy, and the ability to arbitrarily label clocks, $\lambda_{i\to j}$ is not simply the initial placement of frames in the FIFO (as it is in the KPN model), but also the number of frames in-flight on the link, plus any clock offsets.

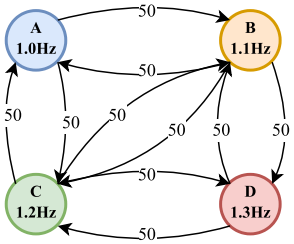


FIGURE 12. LSN example topology for simulations. The nominal/initial frequency of each machine is annotated.

VII. BENEFITS OF ELASTIC BUFFERS

Both the LSFP and bittide implementations are very similar in principle. Both are Kahn-like token pushing systems with intermediate FIFO buffers. The main difference is the process synchronisation behaviour, where LSFPs use the bounded-FIFO model of blocking reads and writes whereas bittide uses a novel clock control system to maintain the bounds of the so-called Elastic FFPs and hence be blocking free. To highlight the differences between the two proposed implementations of LSNs, we briefly compare their runtime behaviour via a discrete-event simulator, which is publicly available.[1] We don't simulate non-LSN architectures (e.g. LTTAs), as the structural and paradigmic difference makes the direct comparison difficult. These are designed to be illustrative of the typical execution characteristics of each platform rather than providing a concrete benchmark (as these are only abstract models). The simulation models a list of periodically executing machines, communicating over a set of links with transmission delay. Each machine has a local clock, and a buffer for each incoming link. The simulation progresses by finding the next deadline from either the machine clocks or in-flight messages, and then advancing the simulation time to that point. When the deadline for a machine elapses, a logical tick occurs during which it performs some calculation and updates its next deadline, based on its assigned control policy (e.g. PID).

We consider the LSN shown in Figure 12 implemented over both LSFP and bittide. This topology is arbitrarily chosen, but the clock behaviours are indicative for other topologies also.

A. THROUGHPUT

First, we examine the throughput of each architecture. Throughput here refers to the number of logical ticks that occur within some time period for some given machine, and hence how many calculations are completed. Because our machines are periodic, we consider the frequency of each logical clock as equivalent to the throughput of the machine.

For our example topology, Figure 13 shows the frequency of each local clock as it evolves during an execution for both LSFP and bittide systems. For the LSFP implementation (Figure 13a), the flow of tokens is limited by the slowest machine in the system. The slowest machine can remain free

[1] https://github.com/PRETgroup/bittide_sim_wip

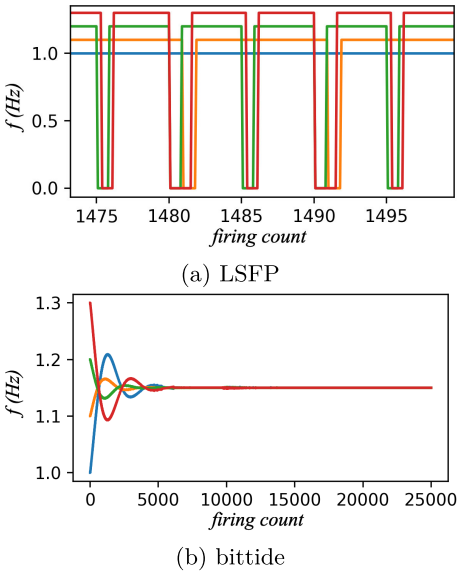


FIGURE 13. Comparison of LSFP and bittide tick/firing rate on a fully-connected four-machine network. Each coloured line represents a unique machine clock.

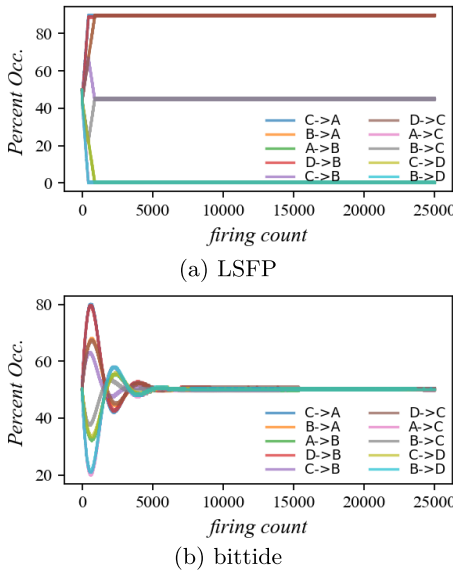


FIGURE 14. Comparison of LSFP and bittide channel buffer occupancy on a fully-connected four-machine network. Each coloured line represents a single channel.

running, while the others must 'stutter' to modulate to the same value as the slowest machine on average.

In contrast, the bittide proportional-integral controller (Figure 13b) converges all machine frequencies near the midpoint without ever blocking. In systems where task speed-up is possible, the bittide approach may have superior peak throughput. When task rates are very close to begin with, the throughput will be similar for both implementations, but the stabilising effect of elastic buffers will still reduce jitter in the bittide model. Note that logical synchrony is still preserved during the transient or other instability as long as no frame data is lost due to overflow.

Observation 6: Every machine in a LSFP or bittide network must have the same average firing rate.

Due to FIFO bounds, for each producer-consumer pair the average rate of production must be the same as the average rate of consumption, barring some small initial difference before a steady-state is reached. If rates were unequal, FIFOs would experience unbounded growth or loss during an extended execution and cause overflow.

B. PHYSICAL LATENCY

Physical latency between machines refers to the time taken for a token to travel from the producer to the head of the consumer buffer. This is a distinctly different measure of performance to throughput, as latency measures the fastest time a machine can respond to data produced by another machine, rather than how much bulk data is processed. $\tau_{i\to j}(t)$ denotes the time elapsed between a production at $\theta_i(t)=s$ and its consumption at $\theta_j(t+\tau_{i\to j}(t))=s+\lambda_{i\to j}$. This is not equivalent to transmission delay l alone, because the token will be held in the recipient buffer for some time.

Observation 7: The latency $\tau_{i\to j}(t)$ between two machines consists of the transmission delay and buffer propagation time

$$\tau_{i\to j}(t) \approx l_{i\to j} + \frac{\beta_{i\to j}(t)}{\omega_j(t)} \tag{12}$$

Note the implicit assumption that receiver frequency does not vary much during transmission. Because transmission delay is common to the latency for both implementations, we will compare just the buffer occupancy behaviour of each. Figure 14 shows each communication channel's buffer occupancies for the bittide and LSFP example during a slice of execution.

Due to the clock controller, bittide buffers tend towards their initial occupancy. As a result, each bittide channel will (at steady state) have approximately the same latency at

$$\tau^{bt}_{i\to j}(t) \approx l_{i\to j} + \frac{\beta^{max}_{i\to j}}{2\omega_j(t)} \tag{13}$$

This is because $\beta_{i\to j}(t)$ from Equation (12) will evaluate to the buffer midpoint $\beta^{max}_{i\to j}/2$ at steady-state.

In contrast, in a LSFP a buffer from a slower node to a faster node will tend to be near-empty and a buffer from a faster node to a slower node will tend to fill up before the skipping action kicks in. Note that the fast-to-slow link $D \to B$ in the simulation has a high occupancy, and the slow-to-fast link $B \to D$ has an almost-zero occupancy at all times. Consequently, links which end up empty will have shorter latencies than those which end up full:

$$l_{i\to j} \lessapprox \tau^{ffp}_{i\to j}(t) \lessapprox l_{i\to j} + \frac{\beta^{max}_{i\to j}}{\omega_j(t)} \tag{14}$$

Thus, for two identical systems the elastic buffer approach will tend to have a more equal distribution of latencies across all channels, while the LSFP approach will have a more variable distribution. Although, some overhead is incurred

by the elastic buffer system, which needs a larger number of initial tokens than LSFP for use as control system feedback.

C. DISCUSSION

When comparing the two platforms as LSN implementations, the more advanced clock control mechanism of the bittide approach tends to improve throughput and make latency more consistent (after a transient period) compared to a blocking LSFP approach. Figure 13 and Figure 14 are generated from a specific topology, but the nature of the graphs is invariant to the topology given that a stable bittide controller exists. Thus, we demonstrate the key benefits of the decoupled elastic buffer MoC over blocking approaches, for the first time.

This is not entirely surprising given that we implicitly present the assumption in these results that bittide clocks can increase in speed. Even if we remove this assumption and make the control one-sided (not exceeding the initial frequency), then clocks would be expected to settle at the rate of the slowest machine, as is the case in LSFP. The benefits to latency and jitter would still be present however, as the system remains blocking-free.

A major redeeming property of LSFPs is that the governing blocking mechanism can be implemented over many generic hardware platforms. In contrast, the dynamic clock control of bittide systems poses a reasonable implementation hurdle, and we generally assume that it remains a physical-layer protocol for bespoke hardware. This does not preclude the possibility that bittide-like control could be implemented at a more abstract software level.

VIII. CONCLUSION

This paper addresses a gap of realisable formal models that preserve determinism and free running behaviour in distributed systems. We formalise a distributed network of machines as a LSN. LSNs provide a synchronous abstraction of a distributed system. We show that KPNs, and their implementation as LSFPs, preserve the LSN model. Subsequently, we show that the bittide protocol for distributed systems also preserves the LSN model. Thus, LSFPs and bittide offer two distinct ways of realising LSN behaviour. We observe that in both LSFPs and bittide, the Kahn-reminiscent effect of pipelining via buffers is used to achieve a greater overall throughput compared to signalling approaches used in GALS models, while preserving determinism, which is difficult to achieve over GALS. We also show that the decoupling of execution from synchronisation in bittide leads to higher performance.

While this paper paves the way for formalising distributed synchronous systems using LSNs, we have several avenues for future research. Firstly, we need to consider the design of application models which leverage logical synchrony. There is scope for introducing novel constructs into synchronous languages which enable the seamless distribution of programs over bittide-like platforms. Moreover, precise delays expressed in logical-time have implications for efficient dataflow scheduling.

ACKNOWLEDGMENT

The authors give special thanks to other members of the Bittide Team, including but not limited to Pouya Dormiani, Chris Pearce, and Robert O'Callahan, for their continuous feedback and support. Kenwright and Roop also acknowledge constructive feedback received from Michael Mendler and Gerald Lüttegen from Bamberg University. Sanjiva Prasad of IIT Delhi provided significant editorial feedback contributing to the revised manuscript.

REFERENCES

[1] S. Lall, C. Cascaval, M. Izzard, and T. Spalink, "Logical synchrony and the bittide mechanism," 2023, *arXiv:2308.00144*.
[2] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 17, no. 12, pp. 1217–1229, Dec. 1998.
[3] C. A. R. Hoare, *Communicating Sequential Processes*. Englewood Cliffs, NJ, USA: Prentice-Hal, 1985, vol. 178.
[4] R. Milner, *Communication and Concurrency*, vol. 84. Englewood Cliffs, NJ, USA: Prentice-Hall, 1989.
[5] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.
[6] G. Kahn, "The semantics of a simple language for parallel programming," *Inf. Process.*, vol. 74, pp. 471–475, Aug. 1974.
[7] A. Benveniste, P. Caspi, S. 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.
[8] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: Formal models, validation, and synthesis," *Proc. IEEE*, vol. 85, no. 3, pp. 366–390, Mar. 1997.
[9] R. Maier, G. Bauer, G. Stöger, and S. Poledna, "Time-triggered architecture: A consistent computing platform," *IEEE Micro*, vol. 22, no. 4, pp. 36–45, Jul./Aug. 2002.
[10] S. Lall, C. Cascaval, M. Izzard, and T. Spalink, "Modeling and control of bittide synchronization," in *Proc. Amer. Control Conf. (ACC)*, Jun. 2022, pp. 5185–5192.
[11] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincentelli, P. Caspi, and M. Di Natale, "Implementing synchronous models on loosely time triggered architectures," *IEEE Trans. Comput.*, vol. 57, no. 10, pp. 1300–1314, Oct. 2008.
[12] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows*. Alfred P. Sloan School of Management, Massachusetts Institute of Technology, 1988.
[13] D. L. Mills, "Internet time synchronization: The network time protocol," *IEEE Trans. Commun.*, vol. 39, no. 10, pp. 1482–1493, Oct. 1991.
[14] *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, IEEE Standard 1588-2008 (Revision IEEE Standard 1588-2002), 2008, pp. 1–269.
[15] *IEEE Standard for Local and Metropolitan Area Networks—Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks*, IEEE Standard 802.1AS-2011, 2011, pp. 1–292.
[16] V. Addanki and L. Iannone, "Moving a step forward in the quest for deterministic networks (DetNet)," in *Proc. IFIP Netw. Conf. (Networking)*, Jun. 2020, pp. 458–466.
[17] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proc. IEEE*, vol. 91, no. 1, pp. 112–126, Jan. 2003.
[18] 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, p. 36, May 2021.
[19] P. Derler, T. H. Feng, E. A. Lee, S. Matic, H. D. Patel, Y. Zheo, and J. Zou, "PTIDES: A programming model for distributed real-time embedded systems," Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, CA, USA, Tech. Rep. EECS-2008-72, 2008.
[20] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.
[21] G. Baudart, A. Benveniste, and T. Bourke, "Loosely time-triggered architectures: Improvements and comparisons," *ACM Trans. Embedded Comput. Syst.*, vol. 15, no. 4, pp. 1–26, Sep. 2016.
[22] A. Malik, Z. Salcic, P. S. Roop, and A. Girault, "SystemJ: A GALS language for system level design," *Comput. Lang., Syst. Struct.*, vol. 36, no. 4, pp. 317–344, Dec. 2010.

[23] T. Gautier, P. Le Guernic, L. Besnard, and J.-P. Talpin, "The polychronous model of computation and Kahn process networks," *Sci. Comput. Program.*, vol. 228, Jun. 2023, Art. no. 102958.
[24] 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, Sep. 1991.
[25] P. Caspi, A. Girault, and D. Pilaud, "Automatic distribution of reactive systems for asynchronous networks of processors," *IEEE Trans. Softw. Eng.*, vol. 25, no. 3, pp. 416–427, May/Jun. 1999.
[26] S. A. Edwards, "On determinism," in *Principles of Modeling*. Cham, Switzerland: Springer, 2018, pp. 240–253, doi: 10.1007/978-3-319-95246-8_14.
[27] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.
[28] S. Lall, C. Cascaval, M. Izzard, and T. Spalink, "Resistance distance and control performance for bittide synchronization," in *Proc. Eur. Control Conf. (ECC)*, Jul. 2022, pp. 1850–1857.
[29] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin, "Elastic circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 10, pp. 1437–1455, Oct. 2009.
[30] S. Lall, C. Caşcaval, M. Izzard, and T. Spalink, "On buffer centering for bittide synchronization," in *Proc. 9th Int. Conf. Control, Decis. Inf. Technol. (CoDIT)*, Jul. 2023, pp. 2348–2353.

LOGAN KENWRIGHT received the B.E. degree (Hons.) in computer systems engineering from The University of Auckland, where he is currently pursuing the Ph.D. degree, under the supervision of Partha Roop. His research interests include programming models for synchronous and deterministic systems.

PARTHA ROOP received the B.E. degree in computer science and engineering from CEG, Anna University, Guindy, the M.Tech. degree in computer science and engineering from IIT Kharagpur, and the Ph.D. degree in computer science and engineering from UNSW Sydney. He is currently a Professor and the Associate Dean of International with the Faculty of Engineering, The University of Auckland. He is involved in the global partnership on AI for future pandemic resilience. His research interests include formal methods for safety-critical software, AI, and machine learning, especially focusing on safety and real-time systems. He is a Steering Committee Member of the IEEE/ACM International Conference on Formal Methods and Models of Codesign (MEMOCOE) and an Associate Editor of IEEE EMBEDDED SYSTEMS LETTERS.

NATHAN ALLEN received the B.E. (Hons.) degree in computer systems engineering and the Ph.D. degree in computer systems engineering from The University of Auckland, New Zealand, in 2015 and 2021, respectively. He is currently a Research Fellow with the Department of Electrical, Computer, and Software Engineering, The University of Auckland. His research interests include synchronous programming languages, formal methods, and embedded systems, which were all applied to the design of biomedical embedded systems in his Ph.D. research and subsequently to the use of spiking neural networks in robotic applications.

SANJAY LALL (Fellow, IEEE) received the B.A. degree (Hons.) in mathematics and the Ph.D. degree in engineering from the University of Cambridge, England, in 1990 and 1995, respectively. From 2018 to 2019, he was the Director of the Autonomous Systems Group, Apple. He was a Research Fellow with the Department of Control and Dynamical Systems, California Institute of Technology. Prior to that, he was a NATO Research Fellow with the Laboratory for Information and Decision Systems, Massachusetts Institute of Technology. He was a Visiting Scholar with the Department of Automatic Control, Lund Institute of Technology. He is currently a Professor of electrical engineering with the Information Systems Laboratory, Stanford University. He is also a Visiting Researcher with Google. 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. His research interests include algorithms for control, optimization, and machine learning.

CĂLIN CAŞCAVAL (Fellow, IEEE) received the joint M.S. degree from the Technical University of Cluj, Romania, and West Virginia University, USA, and the Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign, in 2000. He is currently the 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; and 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 Thomas J. Watson Research Center.

TAMMO SPALINK was born in Germany. He received the B.S. degree in computer science from Carnegie Mellon University, the M.S. degree in computer science from The University of Arizona, and the Ph.D. degree in computer science from Princeton University. He has spent his career to date at Alphabet, where he has contributed to Android, ChromeOS, Loon, and numerous internal projects. He is currently an Engineering Director with Google Research, where he is responsible for a range of projects from silicon design to machine learning compilation.

MARTIN IZZARD received the B.S.E.E. and M.S.E.E. degrees from Natal University (today the University of KwaZulu-Natal) and the Ph.D. degree from Trinity College Cambridge. He was with Texas Instruments, Dallas, TX, USA, for 21 years, starting in research and then holding a variety of technical- and general-management roles with the Digital and Analog Division, TI. He co-founded an Ethernet Switch Chip Company, in 2017. He joined Google, in 2019, as a Research Director. He was awarded a title "A Fellowship" at Trinity.