IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Digital Object Identifier

Logically Synchronous Behaviour over Asynchronous Distributions

LOGAN KENWRIGHT[1], PARTHA ROOP[1], AND NATHAN ALLEN.[1]
[1]Department of Electrical, Computing and Software Engineering, The University of Auckland, New Zealand

This work was supported in part by funding from a Google Research grant

ABSTRACT Synchronous languages have long been lauded for their amenability to static analysis and determinism guarantees, thus being the paradigm of choice for safety critical applications such as avionics. However, the distribution of synchronous applications has been inefficient due to the need for a global notion of time across asynchronous clock domains. This paper explores a new operating model called logical synchrony. Logically synchronous systems enforce strong coupling between the local transmission and arrival time of signals with respect to a logical time. Consequently, distributed systems do not need to remain synchronised in real wall-clock time in order to execute deterministically and causally. The execution behaviour is similar to that of a Kahn Process Network. We examine two existing system architectures, Triggered Synchronous Block Diagrams and bittide, that demonstrate this characteristic. We demonstrate their execution behaviour and performance on some simulated network topologies, and justify their use-case over a typical globally-asynchronous-locally-synchronous system.

INDEX TERMS synchronous models, asynchronous distribution, logical synchrony, kahn process network, bittide, loosely time-triggered architectures, deterministic algorithms

I. INTRODUCTION

THE synchronous paradigm describes programs which execute in discrete instances of logical time, or 'ticks', rather than in continuous wall-clock time. Interactions with the outside world or with other concurrent threads are only evaluated at these discrete points, where we perceive the underlying operations as having executed 'instantaneously'. Compared to an asynchronous concurrent application, where parallel threads may interleave in a nondeterministic fashion, we arrive at a model of execution which provides very deterministic execution and is free of race conditions. Consequently, we can model complex behaviours accurately using powerful formalisms, and thus prove correct behaviour statically. Guaranteed behaviour is particularly important for safety-critical applications, such as avionics or healthcare. Notable languages that follow this paradigm include Esterel [1], SIGNAL [2], and Lustre [3].

However, such models have historically been challenging to implement on modern distributed systems. Concurrent threads rely on a shared global tick, which in a distributed setting must be facilitated through communication channels at a significant performance cost. To overcome this challenge, langagues such as SystemJ [4] or Multi-clock Esterel [5] employ the Globally-Asynchronous-Locally-Synchronous (GALS) programming model. In a GALS system, execution within a single clock domain remains synchronous and deterministic, while interactions between different clock domains execute asynchronously. Functional and safety policies for specific designs may be possible to verify on a per-application case [6], but are not deterministic in the general case.

This work explores designs which exist between between the notion of shared global time and the separation from it altogether. We describe *logical synchrony*, a coupling between distinct local notions of time, such that logically-temporal and causal relationships are preserved without enforcing a global time-scale. In doing so, deterministic synchronous behaviour is preserved from the perspective of each individual machine. To our knowledge, several designs [7], [8] already operate on this property but do not provide a formal definition. Section II discusses existing contemporary approaches to the distribution of synchronous models. Section III gives a definition for logical synchrony. Section IV describes two existing system architectures which operate logically synchronously. Section V discusses the behaviour and performance of distributions across these architectures. Finally, section VI concludes and notes directions for future works.

II. RELATED WORKS

Other works relax the assumption of a shared notion of time. Polychrony [9] describes the multiclock model of the GALS language SIGNAL. Informally speaking, under this model some SIGNAL programs are statically proven to be deterministic if their behaviour is latency insensitive, or *'endochronous'*. Similarly, Glabbeek et al. [10] explore equivalence between synchronous labelled transition systems and their distributed refinements, based on asynchronous Petri nets [11]. In these works, any amount of transmission latency and asynchrony are permissible as long as some equivalence relation holds. These equivalence classes often respect solely the precedence relation $\prec$ between causal places, and disregard temporal behaviour as 'unobservable'. For mostly latency-insensitive event-driven systems, no additional transformation is required for distribution. Consequently, they define an acceptable class of programs are those where latency is not semantically significant, or at the very least can be proved per-design to not violate some constraints.

The language Heptagon [12] models multirate parallelism using the concept of futures. A synchronous application may spawn asynchronous threads which are expected to return a value at some point in the future. The synchronous caller thread keeps a reference to the future variable and may keep free running until it actually needs to access the value within the reference. If the future has not already returned by the point it is requested, the system will block. This style of decoupling allows for powerful pipelined applications while preserving determinism. A related concept is weak endochrony [13], which identifies independent operations within a synchronous model. Each program design requires oversight to make sure asynchronous latencies are appropriately masked to prevent unnecessary blocking. We suggest that this model of computation is more intuitive when describing the distribution of a synchronous machine, rather than a composition of various synchronous machines that is more typical for the GALS approach.

Tangentially, many time management algorithms [14] exist for enforcing the causal processing of messages across a distributed system, based on vector clocks. These tend to be implemented at an a software level, and incur a harsh performance penalty. Such a technique could be applied to communicating locally-synchronous machines, although the onus is on the programmer to manually coordinate global state, as the machines will not execute in lockstep. This also opens up the risk of a buffer overflow from unbounded production/consumption rates. In contrast, a logically synchronous system will remain synchronous for every tick, dramatically reducing the state space for analysis.

III. LOGICALLY SYNCHRONOUS MODEL

Logical synchrony describes systems where there exists a fixed relationship between the logical sender clock when a signal is emitted, and the logical receiver clock when that signal is consumed. Figure 1 demonstrates three machines labeled $\mathscr{M}_i, \mathscr{M}_j, \mathscr{M}_k$. Each machine represents one arbitrary

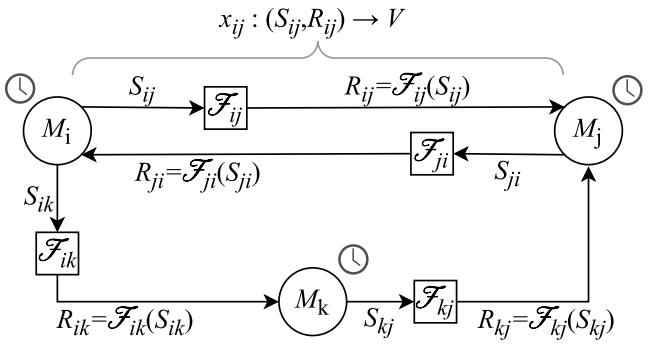


Figure 1: Three physically asynchronous nodes, communicating over a logically synchronous medium

synchronous application, executing with its own clock. The internal behaviour of each application is not relevant to the semantics of the composition, so we can assume it executes some valid synchronous model such as a state machine. Each machine operates in an asynchronous composition to one another, with some communication links. Inspired by [15], we will describe the behaviour of the distributed system as a set of stream functions over its communication links. The presence of a link does not imply that application-meaningful data will necessarily be communicated between any two applications. We define each of these links as a stream function:

$$\begin{aligned} s &: S \rightarrow V | S \in \mathbb{N} \\ r &: R \rightarrow V | R \in \mathbb{N} \end{aligned} \tag{1}$$

V is the set of signals being communicated between two processes. The range coincides for both streams. A signal itself may be a set of arbitrary values, or the empty set. s is the sender stream function, which maps the sender's clock count S to a set of outbound values. r is the receiver stream function, which maps the receivers clock count R to a set of inbound values. Furthermore, S is related to R by the *logical delay function*

$$\mathscr{F} : S \rightarrow R \iff s(S) = r(R) \tag{2}$$

Consequently, we arrive at the composite function representing the mapping each of the related send/receive pairings to a signal.

$$\begin{aligned} x &: (S, R) \rightarrow V \\ \therefore x &: (S, \mathscr{F}(S)) \rightarrow V \end{aligned} \tag{3}$$

$\mathscr{F}$ can be any monotonic function. A deterministic overlap in mappings as acceptable, as long as a combine function is defined. Distributed machines may also be logically multi-rate, again necessitating a combine function. Thus, we define a class of synchronous system where behaviour is deterministic with respect to local clock ticks, while making no judgment about the physical time behaviour. Consequently, a global observer may observe two machines having reached dramatically different ticks, while each machine locally observes its inputs arriving synchronously at the correct time.

A. KAHN PROCESS NETWORK

This concept may feel unintuitive at first, but consider the behaviour of a Kahn process network (KPN) [16]. KPN semantics may be considered a 'base case' for a physically asynchronous, logical synchronous model. KPNs model processes communicating over unbounded FIFOs, such that a single process fires only once values are available at every input. Since processes fire independently, it could be that any one process p fires numerous times before another process q, which depends on data from p and elsewhere, fires even once. Despite this, when q eventually fires for the first time, it will still consume the first value which was written to the intermediate FIFO. From its own local perspective, it needs not make a judgment about how much physical time has elapsed since its inputs were generated, yet it operates deterministically and causally. Thus we can say that from a local perspective, that process executed its cycle in a logically synchronous fashion despite it firing asynchronously to its peers.

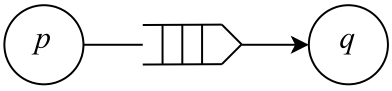


Figure 2: A basic two-node KPN

In the case shown in figure 2, there exists some truly minimal element p which awaits no input, p is free to produce tokens at will. Each token is stored, in order, in the unbounded FIFO channel. Consequently, a token produced in the first firing at p will be consumed at the first firing of q, the second firing at p by the second firing at q, and so forth... such that the logical delay function can be defined as

$$R_q = \mathscr{F}(S_s) = S_s$$

In the case that does not exist a minimal element, such as in the simple example shown in figure 3 it becomes necessary to provide an initial condition to ensure that the system doesn't immediately deadlock. By introducing some initial tokens into the FIFO queues of the system we can ensure liveness, in this case j tokens from $p \rightarrow q$ and k tokens from $q \rightarrow p$. In this scenario, the first token produced at p will be consumed by q on its $(j+1)_{th}$ firing, as it must first await all existing queue tokens to be consumed. Similarly, the first token produced at q will be consumed at p on its $(k+1)_{th}$ firing. We can name this value the inter-process logical delay $\lambda_{p\rightarrow q}$.

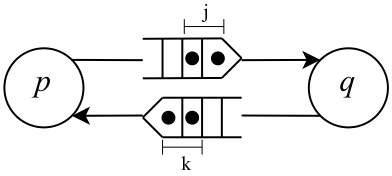


Figure 3: A KPN with no minimal elements, with some initial tokens in each directional unbounded FIFO

Observation 1: The value of the logical delay $\lambda_{p\rightarrow q}(t)$ at any time t consists of the occupancy $\beta_{p\rightarrow q}(t)$ of the FIFO queue from p to q, plus the fire count difference $fire_q(t) - fire_p(t)$.

$$\lambda_{p\rightarrow q}(t) = \beta_{p\rightarrow q}(t) + fire_q(t) - fire_p(t)$$

Lemma 1: The logical delay $\lambda_{p\rightarrow q}(t)$ is invariant for all t

Base case: $t = 0$:

$$\lambda_{p\rightarrow q}(0) = \beta_{p\rightarrow q}(0) + fire_q(0) - fire_p(0)$$
$$\lambda_{p\rightarrow q}(0) = \beta_{p\rightarrow q}(0) + 0 - 0 = \beta_{p\rightarrow q}(0)$$

Case 1: p fires first after time τ_p, producing a token

$$\beta_{p\rightarrow q}(t+\tau_p) = \beta_{p\rightarrow q}(t) + 1$$
$$fire_p(t+\tau_p) = fire_p(t) + 1$$
$$fire_q(t+\tau_p) = fire_q(t)$$
$$\therefore \lambda_{p\rightarrow q}(t+\tau_p) = \beta_{p\rightarrow q}(t) + 1 + fire_q(t) - fire_p(t) - 1$$
$$\therefore \lambda_{p\rightarrow q}(t+\tau_p) == \lambda_{p\rightarrow q}(t)$$

Case 2: q fires first after time τ_q, consuming a token

$$\beta_{p\rightarrow q}(t+\tau_q) = \beta_{p\rightarrow q}(t) - 1$$
$$fire_p(t+\tau_q) = fire_p(t)$$
$$fire_q(t+\tau_q) = fire_q(t) + 1$$
$$\therefore \lambda_{p\rightarrow q}(t+\tau_q) = \beta_{p\rightarrow q}(t) - 1 + fire_q(t) + 1 - fire_p(t)$$
$$\therefore \lambda_{p\rightarrow q}(t+\tau_q) == \lambda_{p\rightarrow q}(t)$$

In simple terms, the value of the logical delay will remain constant, but the composition of the logical delay will be exchanged between the occupancy of the buffer and the drift of the fire counts. The physical wall-clock latency between the two will be determined by the proportion of logical delay consisting of buffer occupancy. The maximum possible amount of fire count deviation is determined by the number of initial tokens.

The logical delay function of such a system can be described as

$$R_q = \mathscr{F}(S_p) = S_p + \lambda_{p\rightarrow q}$$

Of course, the practical design of such a function involves the design of a communication fabric which adheres to this property. There are difficulties with mapping the KPN model of computation to a physical implementation, notably the use of infinitely large intermediate FIFOs. The Synchronous Data Flow [16] model overcomes this limitation by generating a schedule of known production/consumption orderings. However, we strive for logically synchronous platforms which are modular and don't require total ordering, generalising for arbitrary compositions of synchronous models.

IV. ARCHITECTURES

There already exist proposed distributed architectures which satisfy our definition of logical synchrony. Triggered synchronous block diagrams [7], extending older works on backpressure LTTAs [15], are described as stream-equivalent to comparable synchronous models. More recently, the term logical synchrony was coined in the proposal of the bittide architecture [8], [17], developed at Google Research. However,

the existing bittide works largely examine the architecture from a control systems perspective, leaving a semantic gap to traditional synchronous computing. This section describes and examines both aforementioned architectures.

A. SYNCHRONOUS BLOCK DIAGRAMS

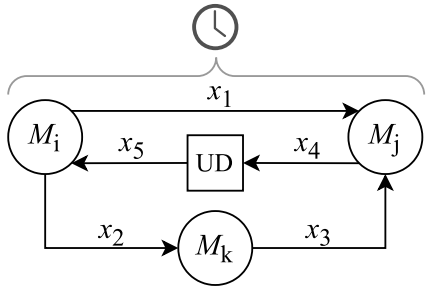


Figure 4: Three-block synchronous block diagram

A synchronous block diagram (SBD) describes a composition of several synchronous components, where each discrete synchronous behaviour is represented as a block. Blocks may be connected by links, which represents a synchronous signal communication. Once again, the behaviour of internal block logic does not affect the semantics of the composition as long as it executes a valid synchronous model. Each block is executed in *lock-step*, meaning each synchronous model in the composition runs a single cycle each global clock tick. Because instantaneous communication is permitted in the synchronous paradigm, partial orderings must be identified to establish a 'firing order' for the blocks to be executed in sequence during a single global cycle. The SBD shown in figure 4, adapted from [15], has the total ordering $\mathscr{M}_i \prec \mathscr{M}_k \prec \mathscr{M}_j$. The final link from $\mathscr{M}_j$ back to $\mathscr{M}_i$ is broken by a unit delay block to prevent a causality loop. On a centralised synchronous platform, such a firing order is simple to maintain.

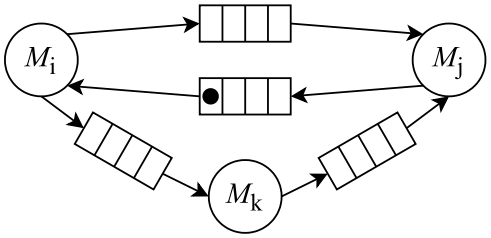


Figure 5: An equivalent FFP for distribution the three-block SBD. One token is placed in the unit delay buffer as an initial condition to guarantee liveness

Triggered synchronous block diagrams (TSBDs) [7] describe block diagrams where the firing of each locally synchronous block is guarded by 'trigger' function. When a block periodically attempts to fire based on its local clock, it first checks whether the trigger condition is met. If the trigger condition is not met, it 'stutters', and skips that firing cycle without incrementing its tick count. This skipping property allows the TSBDs to be further mapped to a distributed setting, where each block is separated by a finite length FIFO queue. This distribution is dubbed a 'Finite FIFO Platform' (FFP). Figure 5 shows an FFP with an equivalent topology to the SBD shown above.

Within a FFP, each block will periodically attempt to fire on its own local clock. First however, it must check the trigger condition. The block will only fire if tokens are available for consumption at every input FIFO, and if there is space to emit tokens at every output FIFO. This is an adaption of Kahn Process Network firing rules, with the additional rule of only firing when a bounded destination buffer is not full. As a skipped cycle does not affect the stream semantics of a KPN, it is acceptable to skip because of either condition at any point. For mealy machine processes, even a queue of size 1 is acceptable, although throughput can be improved with deeper queues. Correspondingly, the logical delay function will also be

$$R_q = \mathscr{F}(S_p) = S_p + \lambda_{p \to q} \tag{4}$$

Where $\lambda_{p \to q}$ is equal to the number of initial tokens in the link $p \to q$.

1) Implementing the FIFO API

It may not be apparent from the abstraction above how an intermediate FIFO can be modelled on a real system. It is desirable to be able to call the $isFull()$ and $isEmpty()$ functions without needing to make an expensive blocking request (otherwise one might as well use a more traditional rendezvous synchronisation).

One approach is have the receiver own the FIFO queue. In this case, the $isEmpty()$ becomes trivial to implement locally. Each communication link will also have an unbuffered reverse link. Figure 6 demonstrates this arrangement.

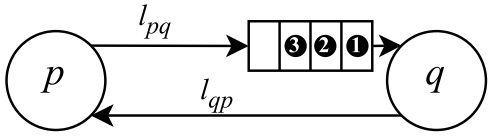


Figure 6: A communication link from p to q, with a back-pressure link transmitting the most recently received frame number

$isFull()$ is slightly trickier to implement. Consider the case where a node p transmits to q. After some link latency l_{pq}, q receives several tokens in its inbound buffer to be consumed when possible. Each token is timestamped with the the fire count $fire_p$ from which it was transmitted. The backwards link from q to p will transmit two values, one value being the the freshest count of $fire_p$ that exists in the tail of the buffer, and the other being its own current count $fire_q$. Sender p now has a snapshot of q, as of a link latency

l_{qp} ago.

$$\beta_q(t - l_{qp}) = receive_q(t - l_{qp}) - fire_q(t - l_{qp})$$
$$\text{Given that } receive_q(t) = fire_p(t - l_{pq}) :$$
$$\beta_q(t - l_{qp}) = fire_p(t - l_{qp} - l_{pq}) - fire_q(t - l_{qp})$$

In the worst case for fullness, where the remote receiver q has not fired even a single time, the most the occupancy can have increased by is the total number of firings of p since the snapshot.

$$\Delta\beta_q^{max} = fire_p(t) - fire_p(t - l_{qp} - l_{pq})$$

Therefore, we can define an $isFull()$ function on the sender side which is guaranteed to give a conservative estimate of buffer occupancy:

---

if $\beta(t-l) + fire_p(t) - fire_p(t - l_{qp} - l_{pq}) \geq \beta_{max}$ **then**
  return true
else
  return false
end if

The amount by which this function overestimates is proportional to the relative physical transmission latency. As a side-effect, one can expect that the occupancy of a buffer will never be allowed to use its full capacity. This may have some throughput implications, but the behaviour is safely preserved. For a link latency relatively small compared to your firing rate, the approximation may be more appropriate.

B. BITTIDE

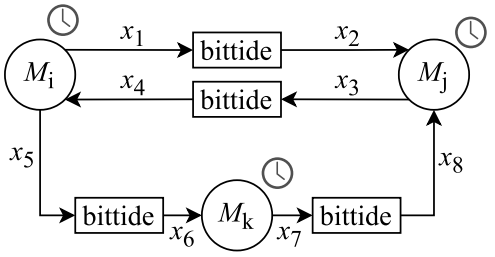


Figure 7: A basic three-node fully-connected bittide network

bittide [8], [17] is a proposed system architecture for distributed computing, developed at Google Research. Inspired by elastic circuits [18], variations in frequency are absorbed to maintain synchrony. A bittide system is comprised of a network of machines $\mathscr{M} = \{\mathscr{M}_0, \mathscr{M}_1, \dots\}$. Each of these computers are free running, in that they have a local clock phase θ which increments periodically in count. An edge between any two nodes models a bidirectional communication channel which consists of a FIFO queue at each node (the *elastic buffer*), and a finite length link in each direction $\mathscr{L} = \{\mathscr{L}_{ij} | \mathscr{M}_i, \mathscr{M}_j \in \mathscr{M}\}$. Figure 7 shows a three node system in this configuration.

Bittide continually exchanges quantums of data, or frames, between any two nodes. Figure 8 shows the communication

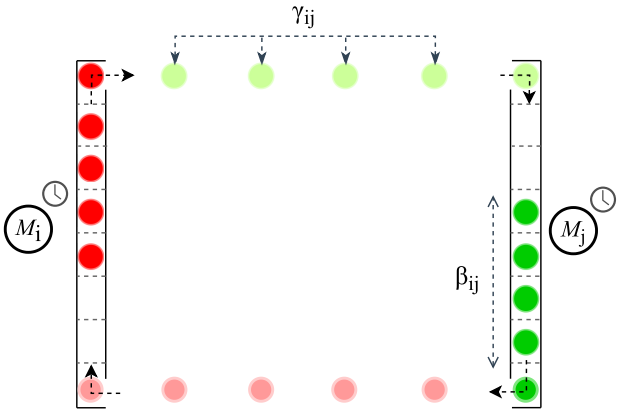


Figure 8: Interaction between two bittide nodes, demonstrating the elastic buffer and link action

loop between any two nodes in a network. Periodically, a clock at a machine $\mathscr{M}_i$ will 'tick', increasing its phase θ_j by one. In our synchronous model context, we assume discrete clocks.

$$\theta_i = \{\dots, 1, 2, 3, \dots\} \in \mathbb{Z}$$

At each tick, the node must 'pop' a frame off the head of its elastic buffer, and send a frame onto the link with occupancy γ_{ij}. When a frame of data arrives at $\mathscr{M}_j$ from the inbound link it is immediately (asynchronously) pushed to the current tail position (occupancy β_{ij}) of the elastic buffer corresponding to that bidirectional link. It will be later consumed by the receiver node once it propagates to head of the queue after β_{ij} local receiver ticks. The current occupancy β of a node's buffers is used as a feedback signal to determine clock rate relative to its neighbours, and thus apply a clock control policy 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. Consequently any number of control policies may be valid for a given network topology, although it is typically a variation of proportional or integral control. In fact, one valid control policy would be to simply implement the skipping mechanism employed by TSBDs, although this would sacrifice the clock stabilisation behaviour which allows "free running" without notable jitter.

Observation 2: A frame sent from a node $\mathscr{N}_i$ at send time t will arrive at receive time $t+\tau_{tx}$ at the head of node $\mathscr{N}_j$ after all preceding frames have been consumed. Therefore we can observe the phase relationship of the receiver at each time correspondingly:

$$\begin{gathered}
\theta_j(t + \tau_{tx}) = \theta_j(t) + \gamma_{ij}(t) + \beta_{ij}(t) \\
\text{let } \Delta\theta_{ji}(t) = \theta_j(t) - \theta_i(t) \therefore \theta_j(t) = \theta_i(t) + \Delta\theta_{ji}(t) \\
\therefore \theta_j(t + \tau_{tx}) = \theta_i(t) + \underbrace{\gamma_{ij}(t) + \beta_{ij}(t) + \Delta\theta_{ji}(t)}_{\lambda_{ij}(t)} \\
\therefore \theta_j(t + \tau_{tx}) = \theta_i(t) + \lambda_{ij}(t)
\end{gathered} \tag{5}$$

Therefore, we can define our logical time delay λ_{ij}. This describes the relationship between the phase at $\mathscr{M}_i$ when a frame is sent and the phase at $\mathscr{M}_j$ when it eventually gets received *regardless of how much physical wall-clock time has occurred in the meanwhile*.

Observation 3: The number of frames in circulation remains constant, as they may only be exchanged between buffers and links. The number is therefore the summation:

$$\beta_{ij}(t) + \beta_{ji}(t) + \gamma_{ij}(t) + \gamma_{ji}(t)$$

Taking the derivative gives us the exchange relationship between each half of the loop

$$\beta'_{ij}(t) + \beta'_{ji}(t) + \gamma'_{ij}(t) + \gamma'_{ji}(t) = 0$$
$$\beta'_{ij}(t) + \gamma'_{ij}(t) = -(\beta'_{ji}(t) + \gamma'_{ji}(t)) \tag{6}$$

Observation 4: Line occupancy increases when a sender node ticks, and decreases when a frame arrives at receiver buffer

$$\gamma'_{ij}(t) = \gamma'_{ij_{in}}(t) - \gamma'_{ij_{out}}(t) \tag{7}$$
$$\gamma'_{ij_{in}}(t) = \theta'_i(t)$$

Observation 5: Buffer occupancy decreases when phase increases, and increases when a frame arrives from a link

$$\beta'_{ij}(t) = -\theta'_j(t) + \gamma'_{ij_{out}}(t)$$
$$\theta'_j(t) = -\beta'_{ij}(t) + \gamma'_{ij_{out}}(t) \tag{8}$$

Substituting values for $\theta'(t)$ from equation (8) and equation (7):

$$\Delta\theta'_{ij}(t) = \theta'_i(t) - \theta'_j(t) = \gamma'_{ij_{in}}(t) + \beta'_{ij}(t) - \gamma'_{ij_{out}}(t)$$
$$\Delta\theta'_{ij}(t) = \beta'_{ij}(t) + (\gamma'_{ij_{in}}(t) - \gamma'_{ij_{out}}(t))$$
$$\therefore \Delta\theta'_{ij}(t) = \beta'_{ij}(t) + \gamma'_{ij}(t) \tag{9}$$

Lemma 2: From equation (5), we define our "logical delay" and prove its invariance:

$$\lambda_{ij}(t) = \gamma_{ij}(t) + \beta_{ij}(t) + \Delta\theta_{ji}(t)$$
$$\therefore \lambda'_{ij}(t) = \gamma'_{ij}(t) + \beta'_{ij}(t) + \Delta\theta'_{ji}(t)$$
$$\lambda'_{ij} = \gamma'_{ij}(t) + \beta'_{ij}(t) + (-\beta'_{ij}(t) - \gamma'_{ij}(t))$$
$$\lambda'_{ij}(t) = 0 \text{ for all } t$$

Therefore, we can say that the λ_{ij} is invariant, meaning the logically synchronous relation will always hold for any system state. This does not imply a value of zero, only that the relationship between send and receive phase is constant. The goal of *logical synchrony* in a bittide system is to maintain the appearance of synchrony from the perspective of any individual node across an asynchronous medium. The elastic buffer action absorbs the underlying asynchrony. From a global omnipotent observer, any one node may deviate in tick count numerous ticks ahead of another, as long as the elastic buffer does not underflow or overflow.

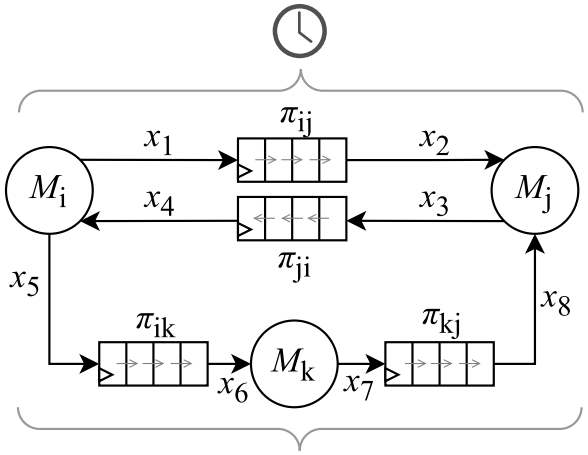


Figure 9: The stream-equivalent synchronous block diagram to the three-node bittide network

Returning back to our stream semantics, we can establish expressions for our sender and receiver streams:

$$s(\theta_R(t)) = r(\theta_S(t - \tau_{tx})) = V$$
$$\text{From equation (5)} : \theta_S(t - \tau_{tx}) = \theta_R(t) - \lambda_{SR}$$
$$\therefore s(\theta_R(t)) = r(\theta_R(t) - \lambda_{SR}) = V \tag{10}$$
$$\text{Our logical delay function is therefore:}$$
$$R = \mathscr{F}(S) = S + \lambda$$

1) Equivalent Centralised Model

Consider a purely synchronous composition of several machines $\mathscr{M}_0, \mathscr{M}_1, \ldots$ where each machine is separated by an intermediate queue of size 1 or greater, where data propagates one position per global tick from in the fashion of a shift register. Although we can theoretically model systems with a zero-delay, we cannot enforce a firing order which respects causality requirements without a centralised controller and consequently signal dependencies must be broken by at least a unit-delay. Concurrency may occur within a single block on a single machine, which obeys typical instantaneous casual behaviour.

Lemma 3: Processes distributed across bittide with some λ_{ij} is stream equivalent to pure synchronous blocks separated by shift registers of size $\pi_{ij} = \lambda_{ij}$:

The behaviour of a clocked shift register of size π_n is:
$$r(\theta(t)) = s(\theta(t) - \pi_n)$$
For a synchronous system: $\theta_r(t) = \theta_s(t) = \theta(t)$
$$\therefore r(\theta_r(t)) = s(\theta_r(t) - \pi_n)$$
letting $\pi_{ij} = \lambda_{sr}$, this is equivalent to equation (10)

As a result, when observing each system based on which signals are sent or arrive from a node at any of their respective local ticks, a bittide distribution appears entirely synchronous. There is no longer a single global timescale from which we can produce an output trace, but all causal behaviours are preserved and deterministic. This separation

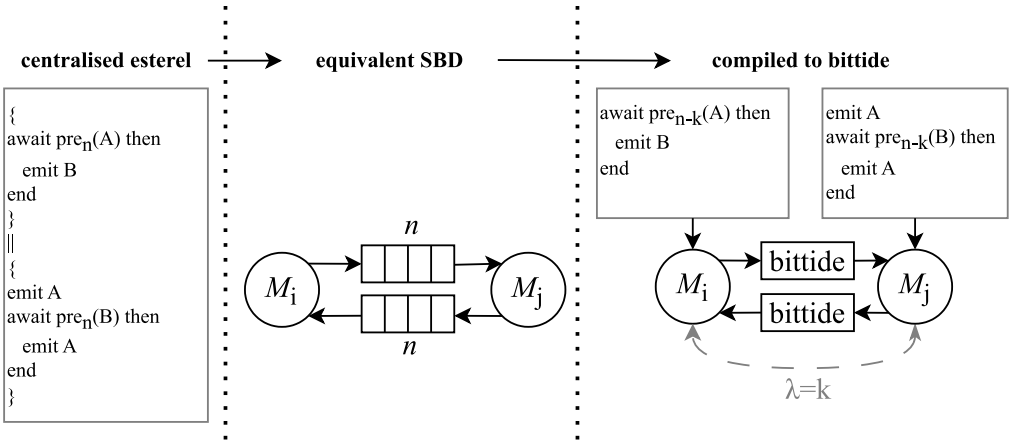


Figure 10: Equivalent exchange of software delays to logical delay λ_{ij}

from a global timescale implies that some local tick labelled $tick_n$ could occur at completely different times on any two nodes, although a timing bound can be calculated based on system parameters. Whether or not the different real-time behaviour as observed by a third party is appropriate for a given use-case is left up to the designer.

Many synchronous languages provide constructs for logical time delay, such as *pre()* in Esterel. Each delay operator behaves similarly to a unit delay, and can be cascaded to produce equivalent behaviour to a shift register-separated SBD.

More commonly, a designer may wish to create a centralised model first and then later distribute over bittide. As long as concurrent components contain logical delays (e.g. via *pre()*) such that any circular data dependencies are delayed by at least the round trip logical delay of the target bittide platform, they they can be deployed as such. Such a transformation is shown in figure 10. Each software module deployed on a single mode must have its software buffer reduced by an amount equivalent to the apparent bittide logical delay λ_{ij}.

2) Relabelling Ticks

Although lemma 2 shows λ_{ij} to be invariant, we can forcibly set the phase difference term $\Delta\theta_{ji}(0)$ at startup. Assuming the presence of some handshaking startup protocol, there is no necessity to start measuring logical ticks from the same value.

Observation 6: $\lambda_{ij} + \lambda_{ji}$ can be relabelled to any values that satisfy $(\lambda_{ij} + \lambda_{ji})$ = Frames in circulation = Logical Round Trip Time

Consider the case where $\mathscr{M}_i$ begins at $tick_{-(\gamma_{ji}+\beta_{ji})}$ compared to $\mathscr{M}_j$ which is begins at $tick_0$. In this scenario, the observable delays are:

$$\begin{aligned} \lambda_{ji}(0) &= \gamma_{ji} + \beta_{ji} - (\gamma_{ji} + \beta_{ji}) = 0 \\ \lambda_{ij}(0) &= \gamma_{ij} + \beta_{ij} + \gamma_{ji} + \beta_{ji} = \text{round trip time} \end{aligned} \tag{11}$$

Naturally, relabelling in this fashion cannot improve physical time reaction or response time. However, it may be meaningful from a designer point of view if they wish to preserve the syntax of a particular software design which may be better expressed as a large delay in one direction rather than a balanced delay in each.

As an interesting side effect, you can relabel λ_{ij} to a negative value and maintain determinism, although this case is difficult to rationalise as an equivalent SBD.

V. PERFORMANCE

It should be noted that a direct comparison between an FFP and a bittide distribution is not entirely fair. An FFP is trivial to implement over any arbitrary machine on which a FIFO queue can be modelled. In comparison, the bittide architecture is almost inseparable from an implementation with dynamic clock control, unless an adjustable derived clock is used.

Additionally, an FFP can implement essentially any synchronous composition with a 'bare minimum' quality of service due to its blocking behaviour. In comparison, a 'pure' bittide system is somewhat more fragile. The 'elastic buffer' behaviour relies on maintaining a buffer occupancy somewhere between full and empty at all times. An undersized buffer or an inadequate control policy can easily result in an overflow or underflow, for which the behaviour is undefined at a model level. A larger buffer implies a larger logical, and likely physical, communication latency.

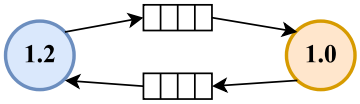
The tradeoff for the additional complexity in the bittide case is that a well-designed system should remain continuously free-running for an indefinite time period, resulting in higher and more consistent throughput. A FFP takes no action to 'correct' the relative speed of clocks, only to gate them via blocking. Thus, one can expect significantly more clock jitter and tail latency from an FFP system.

Despite originating from two different works, there is no incompatibility between these two approaches. The FFP skipping model could be an in-place replacement for the controller on a bittide implementation, such that bittide could be considered a special case of an FFP. The key difference is that bittide by definition includes an explicit clock controller

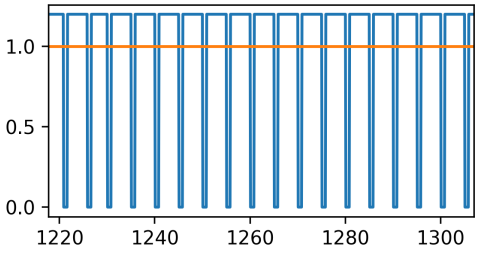
which mediates the production/consumption rates of date, whereas an FFP implicitly controls via clock gating.

A. EXPERIMENTAL RESULTS

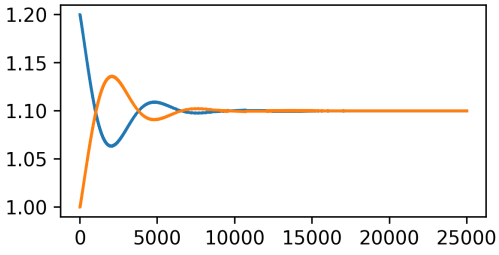
As a visual aide, we provide runtime behaviour for several arbitrary network topologies. We produced our own simulations for bittide and TSBD systems, which can be accessed at *https://github.com/PRETgroup/bittide_sim_wip*. Nominal firing rates are listed on the nodes. For consistency the initial token count is specified the same for both FFP and bittide implementations such that the design is appropriate for both platforms, even though an FFP can express a super-set of bittide designs. Frequency graphs for the FFP implementations have been zoomed to demonstrate the stuttering behaviour. Figure 11 shows a two-node pair, figure 12 shows a four-node ring configuration, and figure 13 shows a four-node mesh configuration.



(a) Topology

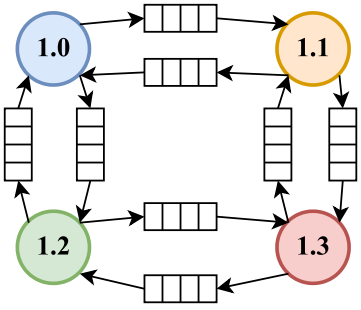


(b) Relative frequencies of FFP nodes vs. fire count

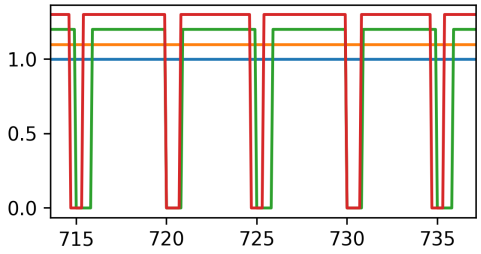


(c) Relative frequencies of bittide nodes vs. fire count

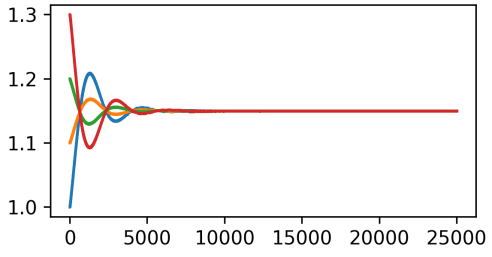
Figure 11: Two nodes, connected by a bidirectional link.



(a) Topology

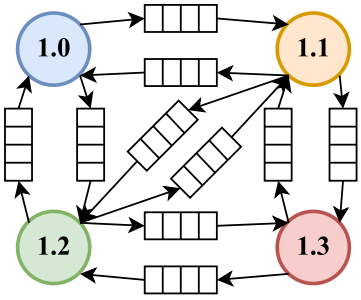


(b) Relative frequencies of FFP nodes vs. fire count

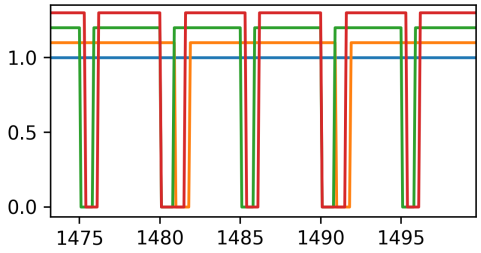


(c) Relative frequencies of bittide nodes vs. fire count

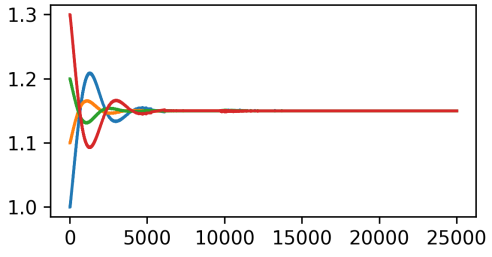
Figure 12: Four nodes, connected in a ring



(a) Topology



(b) Relative frequencies of FFP nodes vs. fire count



(c) Relative frequencies of bittide nodes vs. fire count

Figure 13: Four nodes, fully-connected

The execution behaviour is similar for all topologies shown. The bittide proportional-integral controller, after some transient, converges all node frequencies near the midpoint. Logical synchrony is still preserved during this transient due to the elastic buffer action, as long as the control policy is sufficient to prevent overflow or underflow.

The flow of tokens for FFP execution is limited by the slowest node in the system. This node can remain free running, while the others must 'stutter' to modulate to the same value as the slowest node on average.

Under these ideal conditions, bittide can provide on average superior throughput and avoid the frequency jitter caused by stuttering in FFP, which is likely to be unacceptable in some applications. Of course, these examples were designed with a permissible control policy and a defined set of initial conditions, tuned to ensure stability. Other bittide control policies, such as a reframing controller [19], may relax the designer input required compared to the PI approach. The FFP approach, which doesn't depend on explicit control, can be adapted to more general systems.

B. APPLICABILITY

As noted in [15], *"one may argue that stream semantics preservation is a questionable feature for real-time systems"*. Indeed, the definition we present for logical synchrony makes no judgment about the true time difference between two nodes. In theory, a system could be operating perfectly logically synchronously while having their respective ticks occur hours or days apart.

In practice, we pose a few counter-points. For one, not all systems rely on a wall clock reference. A large micro-simulation of city traffic may benefit from a distribution which is deterministic and causal, but is not affected by the execution behaviour in real-time. Additionally, the physical implementation of such a system will have bounded behaviour. The drift between clocks is limited by the buffer size and in the case of bittide, oscillator control policy. That is to say, logical synchrony does not imply excessive drift will occur, that is simply a decision for the underlying implementation. There may be even cases when it is beneficial for one clock domain to overtake another in a real-time system, for example when sampling periodic bursts of high-frequency data.

VI. CONCLUSION AND FUTURE WORK

In this paper, we demonstrated that locally synchronous models can exhibit a form of global synchrony when distributed across a logically synchronous architecture. Efficient distribution of synchronous models has proven a historical challenge, with most implementations settling for a GALS distribution. Rather than preserve the concept of a global shared timescale, logical synchrony describes a fixed relationship between all local timescales. We demonstrate this relationship for existing architectures: both for finite-FIFO platforms and bittide. We provide experimental results of distributed synchronous models, showing theoretical throughput for several network topologies. In general, the more recent bittide architecture demonstrates high-throughput free running behaviour, at the expense of the expressiveness provided by FFPs.

While this work focuses on semantics preservation, there is reason to believe that the existing benefits of the synchronous paradigm for real-time systems can also be extended to logical synchrony. Earlier works [17] describe the design of permissible controllers but not their relationship to timing behaviour, which we believe can be used to produce tight timing bounds for safety-critical cyber-physical systems. Future works will explore the applicability of logically synchronous architectures for deployment in real-time systems.

VII. ACKNOWLEDGMENTS

References

[1] Frédéric Boussinot and Robert De Simone. The esterel language. Proceedings of the IEEE, 79(9):1293–1304, 1991.

[2] Abdoulaye Gamatié and Thierry Gautier. The signal synchronous multiclock approach to the design of distributed embedded systems. IEEE Transactions on Parallel and Distributed Systems, 21(5):641–657, 2009. Publisher: IEEE.

[3] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. Proceedings of the IEEE, 79(9):1305–1320, 1991. Publisher: IEEE.

[4] Avinash Malik, Zoran Salcic, Partha S. Roop, and Alain Girault. SystemJ: A GALS language for system level design. Computer Languages, Systems & Structures, 36(4):317–344, 2010. Publisher: Elsevier.

[5] Gérard Berry and Ellen Sentovich. Multiclock esterel. In Advanced Research Working Conference on Correct Hardware Design and Verification Methods, pages 110–125. Springer, 2001.

[6] Heejong Park, Avinash Malik, and Zoran Salcic. Compiling and verifying SC-SystemJ programs for safety-critical reactive systems. Computer Languages, Systems & Structures, 44:251–282, December 2015.

[7] Yang Yang, Stavros Tripakis, and Alberto Sangiovanni-Vincentelli. Efficient distribution of Triggered Synchronous Block Diagrams on asynchronous platforms. In 2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), pages 113–122, July 2015.

[8] Sanjay Lall, Calin Cascaval, Martin Izzard, and Tammo Spalink. Modeling and Control of Google bittide Synchronization. arXiv preprint arXiv:2109.14111, 2021.

[9] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. POLYCHRONY for System Design. Journal of Circuits, Systems and Computers, 12(03):261–303, June 2003.

[10] Rob van Glabbeek, Ursula Goltz, and Jens-Wolfhard Schicke. On Synchronous and Asynchronous Interaction in Distributed Systems, December 2008. arXiv:0901.0048 [cs].

[11] Rob van Glabbeek, Ursula Goltz, and Jens-Wolfhard Schicke-Uffmann. On Distributability of Petri Nets, July 2012. arXiv:1207.3597 [cs].

[12] Albert Cohen, Léonard Gérard, and Marc Pouzet. Programming parallelism with futures in lustre. In Proceedings of the tenth ACM international conference on Embedded software, pages 197–206, Tampere Finland, October 2012. ACM.

[13] Dumitru Potop-Butucaru, Benoît Caillaud, and Albert Benveniste. Concurrency in Synchronous Systems. Formal Methods in System Design, 28(2):111–130, March 2006.

[14] Wentong Cai, Stephen J. Turner, Bu-Sung Lee, and Junlan Zhou. An alternative time management mechanism for distributed simulations. ACM Transactions on Modeling and Computer Simulation, 15(2):109–137, April 2005.

[15] Stavros Tripakis, Claudio Pinello, Albert Benveniste, Alberto Sangiovanni-Vincent, Paul Caspi, and Marco Di Natale. Implementing synchronous models on loosely time triggered architectures. IEEE Transactions on Computers, 57(10):1300–1314, 2008. Publisher: IEEE.

[16] Edward A Lee and Thomas M Parks. Dataflow process networks. Proceedings of the IEEE, 83(5):773–801, 1995.

[17] Sanjay Lall, Calin Cascaval, Martin Izzard, and Tammo Spalink. Resistance Distance and Control Performance for Google bittide Synchronization. arXiv preprint arXiv:2111.05296, 2021.
[18] Josep Carmona, Jordi Cortadella, Mike Kishinevsky, and Alexander Taubin. Elastic Circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 28(10):1437–1455, October 2009. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.
[19] Sanjay Lall, Calin Cascaval, Martin Izzard, and Tammo Spalink. On buffer centering for bittide synchronization. arXiv preprint arXiv:2303.11467, 2023.