Communication synchronization

Tammo Spalink
Princeton University
TR-722-05

Abstract

Distributed systems generally employ asynchronous communication, despite most theoretical models of such systems being synchronous, and despite synchrony having established performance and robustness benefits for distributed applications. One reason is that synchronous communication is often impractical. Existing mechanisms for providing synchronous abstractions either are not general or do not scale. To address this shortcoming, this document proposes a scalable communication synchronization framework to provide shared logical time. Communication synchronization aligns all network communication to create a single shared sequence of communication events. This sequence is then used as a time reference. The decentralized "average neighbor algorithm" is proposed to maintain synchrony despite drift between local clocks in the system, and this algorithm is proven to be correct and robust. Simulation results indicate that the overhead costs of providing synchrony are minimal, requiring only a small (e.g. 0.1%) fraction of the bandwidth at each network link, and small fixed incoming data buffers. Thus, the framework allows adoption of synchrony in many circumstances where it has previously been impractical.

1 Introduction

Asynchronous communication is the current standard for computer networks, including the Internet. However, when multiple distributed applications share such a system, they must be designed with flexible performance expectations, as they must tolerate fluctuating contention for potentially limited resources. Formally reasoning about the behavior or correctness of applications in the face of this non-determinism is often unmanageable. Reflecting this, most theoretical models of distributed systems are synchronous [7] – readily allowing powerful temporal reasoning.

While systems exist that match the formal models, such as SONET [25] and most digital circuits, the methods used for creating synchrony do not scale. Higher level techniques, which can be called "synchronous overlays", have alternatively sought to create a synchronous abstraction above an asynchronous communication layer [1, 19, 23, 10, 2]. These overlays also do not scale, however, requiring broadcast communication to maintain synchrony.

This document proposes a framework of algorithmic techniques to synchronize a distributed system precisely with a shared logical time, and approximately with physical time. By leveraging low-level resource control, the overhead cost of the framework is minimized. Scalability is possible because overhead is independent of applications and network size, a function only of bandwidth and clock stability. For example, simulation results indicate that single microsecond synchrony for a network, with 0.1% clock instability and 1 Gbps links, requires at most 1 Mbps communication overhead (0.1%, matching the clock instability) and 2000 b of buffering for each direction on each link. This degree of efficiency enables synchrony in many situations where only asynchrony has previously been practical.

The principle synchronization technique of the framework is network-wide alignment of communication into a single lock-step sequence. This allows each node in the network to independently but equivalently define logical time in terms of the shared communication sequence. By controlling the frequency of communication, the fidelity of logical to physical time can be controlled – limited by the stability of the clocks in the network.

A distributed system with synchronous communication provides a highly desirable platform for implementing distributed algorithms, including concurrency control and more traditional clock synchronization. The useful properties of synchronous communication include:

- globally shared frequency without local drift
- ability to test for absence of messages

Drift is the process where nodes in a distributed system, that were once synchronized, lose their synchrony over time. Eliminating drift can dramatically improve performance for some applications. For example, Byzantine fault-tolerant clock synchronization can be imple-

mented with only a single iteration of distributed consensus. Compare this with existing systems, where the expensive synchronization process must be repeated periodically.

Testing for the absence of data is a key feature in formal synchronous models. In a distributed system, this capability allows more efficient communication protocols where some information can be exchanged implicitly, without actual messages. Key examples include fault-tolerant distributed concurrency control [12], as well as most digital circuit design, where binary data is normally defined as either the presence or absence of signal.

1.1 Clock synchronization

Consider a distributed system composed of communicating but spatially separate processes, where each process has access to a local clock. Assume that, in principle, the clocks all operate at an equal nominal frequency. In practice, clocks are never perfectly accurate and small differences in frequency will manifest over time. Synchronization is the process of correcting or tolerating this problem.

The terms "clock synchronization", "clock distribution", or "network synchronization" are commonly understood to mean, for a network of nodes with independent clocks, adjusting the time reported by each clock to fall within a bounded interval of that reported by an external reference (such as GMT or UTC, via GPS or LORAN). Techniques for addressing this issue have been exhaustively researched [6, 5, 13, 18, 21], and some techniques (such as NTP [17]) are widely deployed.

Communication synchronization is a different process where time is defined logically, to allow perfect agreement between processes at all times. Clock synchronization is more complex than communication synchronization. The trouble with clock synchronization is reaching global agreement on a unit of shared state, specifically a time value. Sharing explicit global state in a distributed system is expensive, and canonically known as "distributed consensus" [4]. If system components can fail in a malicious or methodically destructive manner, which is known to result from programming errors, much additional complexity is introduced. This conclusion, which has significant impact on distributed system design, is known as the FLP result [8]. The difficulty of consensus is well illustrated by the famous Byzantine generals analogy [14].

While this document does not directly address clock synchronization, communication synchronization can be used (among other things) to simplify *existing* clock synchronization approaches by eliminating drift. Without drift, the complex consensus procedure need only be applied once to agree upon an initial time value, after which nodes can increment that value independently, without risk of skew. This is a dramatic improvement over the traditional approach, where nodes must re-synchronize on a periodic basis to recalibrate diverging clock values.

Messerschmitt [15] is recommended as a reference for other synchronization terminology.

1.2 Logical time and event ordering

Communication synchronization depends on the ordering of communication events to define shared logical time. Consider that each process in a distributed system can be considered as a totally ordered set of events. The correctness of a distributed application may depend on the interaction between its component processes, and hence on the relative ordering between their events. Logical time has been shown to be important for both the robustness and performance of distributed applications [12].

In the paper introducing event-based time to distributed systems [11], Lamport defines the *happened before* event relation, to partially order events. Lamport explains that, given any discrete measurement of time, it is not generally possible to impose a total ordering on all system events. Because processes in a distributed system operate concurrently, events may occur simultaneously within multiple processes. Thus the happened before relation imposes only a partial order. Lamport also defines *logical clocks* as those relations between simultaneous events which extend happened before to totally order events, possibly arbitrarily. There can be multiple logical clocks for a system, as only the partial ordering is uniquely determined by system events.

Communication synchrony provides a happened before ordering for all system communication events by aligning them in a globally symmetrical manner. By forcing all communication between a pair of neighboring nodes to occur as a symmetrical exchange of messages, the logical ordering of messages is observed by both sides. By placing this constraint upon all neighboring pairs in the entire network, global message ordering is imposed.

Ordering only communication is sufficient to allow applications to create meaningful logical clocks for other events, by capturing all potential event causality. Consider that non-communication events may occur within processes between cycles of communication, but that these events can only have local effects. By focusing on communication, these local-only events are simply ignored by the communication synchrony ordering. Since communication events are the only opportunity for processes to affect one another, however, the resulting partial order is sufficient to expose any potential causality between events.

1.3 Efficiency requirements

The communication synchronization framework proposed below provides event ordering very efficiently by leveraging low-level control over deterministic communication resources. Specifically, the framework requires that network links support time-division multiplexing, meaning they must have fixed latency, fixed bandwidth, and no minimum transfer size – e.g. serial connections. This means that the implementation level for the framework corresponds approximately with the "data link" layer of the OSI model [26], or with the "network access" layer of the Internet (DoD) model.

To reconcile this (effectively) mandated low-level implementation with the end-to-end argument [22], which generally demands that functionality be provided at the highest possible level, recall the cost of high-level approaches. Synchronous overlays, introduced in Section 1, are forced to broadcast continuously between nodes to maintain synchronization. In fact, Awerbuch [1] formally shows that continuous communication between neighbors in a network is optimal, less communication being insufficient to maintain synchrony. Although the framework obeys this Awerbuch bound, it leverages low-level implementation to avoid any negative performance impact. Communication at the overlay level requires explicit exchange of packets, whereas direct access to the actual links between neighbors can provide synchronization feedback implicitly, for free.

Furthermore, the framework is significantly more robust than any of the overlays. No overlay is tolerant of Byzantine failure. Conversely, by avoiding a high-level protocol, the framework can avoid even the possibility of Byzantine failure – there is no need for explicit consensus of any kind, and hence no opportunity for traitorous behavior.

1.4 Synchronous multiplexing

Low-level communication synchrony is not new. SONET [25], also called the "synchronous digital hierarchy" (SDH) outside the US, is the foundation protocol for most of the global telecommunications infrastructure. SONET is designed around a process called *synchronous multiplexing*, which depends on communication synchrony to coordinate the intersection of multiple time division multiplexed links at each node in its network. In brief, data is encoded on each network link in such a way that frames of data can be switched between the links intersecting at a node based solely on time, not on any in-band signals.

Switching SONET frames occurs at 8KHz on all nodes, meaning that frames are read from all network links during each interval of 125us. The data within these frames is then demultiplexed, possibly reordered across multiple frames, and finally multiplexed and transmitted again. This process only works if exactly the right amount of data to form a frame is available at each node during each interval. If an upstream node were to transmit slowly and send less than the expected amount of data during an interval, demultiplexing will fail and result in communication failure.

To synchronize communication, SONET requires a sophisticated clock distribution network. This process is similar to making a single clock available across a digital circuit, but is adapted for the physical scales of a global system. Each node has a highly accurate and reliable clock, combined with dedicated communication bandwidth for a clock synchronization protocol. For redundancy, there are multiple tiers of clock quality, allowing for reliable operation during periods where either communication fails or some reference clocks are unavailable. By guaranteeing clock synchronization of at least 125us, SONET can guarantee communication synchrony.

Although this approach is successful for SONET, being used nearly universally in global telecommunications infrastructure, it is too "brute-force" to transfer well to other application domains. Using clock synchronization to create communication synchrony is expensive, especially when redundant high quality clocks are needed for robustness. This document provides an alternative algorithmic approach, which is highly robust while tolerating inferior clocks. Thus, it can be deployed on a larger set of implementation platforms. It can also be tuned to different synchronization granularities, which can allow application both a small and large scale.

1.5 Overview

The rest of this document is organized to incrementally introduce the communication synchronization framework. There are three stages of increasing complexity, and each will be dealt with in turn. First, a limited formal version of the framework is defined to address only logical communication synchronization. Although logical synchronization provides all the desired properties in principle, it is completely intolerant of failure and thus impractical. The second stage utilizes physical timers to address this shortfall, but unfortunately suffers from drift between these timers ultimately destroying any synchrony. The third stage addresses the drift problem. A decentralized "average neighbor algorithm" is introduced to coordinate correcting drift as it occurs across the network, which utilizes only local timing information to measure it. This algorithm is proven to be both correct and robust. To provide more quantitative examples of framework performance, results of some simulations are discussed. Finally, the conclusion speculates on the implications of ef-

ficient communication synchronization for future network designs.

2 Logical synchrony

Communication synchrony can be established for a network by imposing a set of invariants on the behavior of all nodes. Whenever any pair of nodes wish to communicate, their communication must coincide with communication between all other nodes as well. This allows each node to participate in each "cycle" of communication, and thereby establish a globally shared logical ordering for communication events.

More precisely, let a *network* be a distributed system composed of *nodes*, which are connected using bidirectional point-to-point *links*. Nodes perform computation, and each node executes a single sequential process. Nodes communicate by sending *messages* to their immediate neighbors. Define the set of nodes and their neighbor sets:

$$N = \{\ i \mid node\ (\ i\ )\ \}$$
$$\eta_i = \{\ j \mid j \in N \wedge link\ (\ i\ ,\ j\ )\ \}$$

Constrain this model with the following two invariants:

> INVARIANT *symmetry* : Communication between connected nodes must occur as a sequence of symmetrical message exchanges. Once a node *A* has sent a message to a neighbor *B*, the node *A* must wait for a message from *B* in return before sending any further messages to *B*.
>
> INVARIANT *yoke* : Messages must be sent in equal quantity by a node to all connected neighbors. If a node *A* exchanges messages with a connected neighbor *B*, the node *A* must also exchange messages with all other connected neighbors before sending any further messages to *B*.

Define nodes to be *logically synchronous* if they observe enough information to construct equivalent orderings for their shared communication. The symmetry invariant creates this agreement for two nodes. Applying only symmetry between all connected nodes would establish independent shared orderings for each link. Nodes would not, however, be able to reason about order across conversations with different neighbors. The yoke invariant aligns orderings across the network, creating an eternal sequence of "atomic steps" where nodes communicate with each of their neighbors during each step.

Since all connected nodes must exchange messages for any nodes to communicate, consider that messages may be effectively empty – acting simply as placeholders to comply with the invariants.

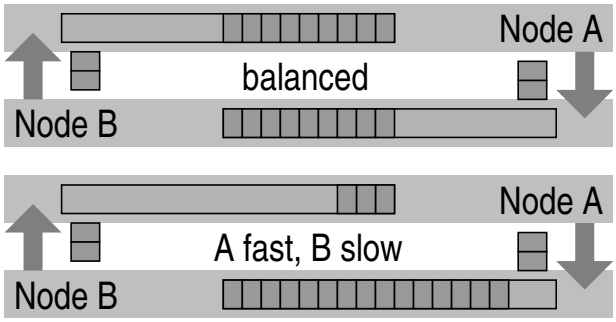

Figure 1: Illustration of the "law of conservation of messages", meaning that the total message count persists across any link, despite variation in node processing frequency.

2.1 Self-timing and initialization

The above invariants impose what can be considered a "law of conservation of messages" for each link. When a message is removed from a link in one direction, another must be added in the opposite direction. This constraint is quite similar to the desired equilibrium behavior of TCP [20]. To ensure robust behavior during congestion, a "conservation of packets" approach is used, also called self-timing [9], which tries to match the generation rate for packets on a connection to the consumption rate.

To prevent self-timing from limiting link bandwidth, introduce a *link initialization* phase to fill the delay-bandwidth product of each link, "priming" it with messages. If applied immediately to empty links, the invariants would limit each link to a single message in each direction per round-trip time – a significant performance limitation. Because the rate of message exchanges must be equivalent for all links (yoke invariant), the throughput of all links would be further bounded by the maximal link delay, since the link with the longest propagation delay must finish its message exchange in lock-step with all other links.

During the initialization phase, nodes transmit messages but perform no receive processing. Assuming that the capacity of each link is known, nodes can generate the appropriate number of messages to fill each link. Once links are full, the invariants are imposed and the conservation property ensures that the number of total messages carried by each link remains constant.

This process addresses the inequality created by varying link propagation delays, and allows for longer links to have more messages "in flight" than shorter links. Note that the invariants need not change to accommodate these additional messages.

2.2 Variable bandwidth

It might seem that the framework imposes bandwidth limits on links, by forcing all links to only exchange the same number of messages. However, the framework does not restrict message size, allowing higher bandwidth links to use correspondingly larger messages. Only the frequency of message exchanges is governed by the invariants.

2.3 Robustness

A real problem with the framework, as currently defined, is intolerance for *communication faults* – meaning circumstances where a link or node becomes unable to transmit messages, or where messages are lost. Communication faults result in "starvation", where all nodes will wait forever for messages and the system as a whole will halt. Specifically, the destination nodes for any missing messages will wait for them, and thereby interrupt the global message exchange cycle. To quote Lamport [11]:

> "... the entire concept of failure is only meaningful in the context of physical time, there is no way to distinguish a failed process from one which is only pausing between events."

Because the invariants ignore time, they imply unbounded blocking of nodes if messages are expected from neighbors but have not yet been received. Tolerating communication faults appears to require that nodes implement timeouts and behave in a temporally regular manner, adding significant additional complexity. This is the focus of the next sections.

3 Physical synchrony

Communication faults can be identified by bounding the duration for which nodes await messages. This means that measurement of time must be introduced into the framework. Accordingly, extend the framework with the following invariant:

> INVARIANT *isochrony* : Nodes must produce messages on each connected link at a constant frequency (isochronously).

While the initial two invariants created a global sequence of message exchanges, this new invariant imposes the temporal regularity of a *communication frequency* upon that sequence. Let the term *period* refer to the constant wavelength of this communication frequency, meaning the amount of time between communication cycles.

Extend also the network model, by assuming that links have fixed latency and bandwidth. The resulting process of transmitting data using a fixed frequency over a deterministic medium is commonly called time division multiplexing (TDM). Accordingly, let the term *frames* replace messages as the name for information exchanged by nodes – indicating the change of focus to TDM. Each link transmits one frame in each direction during each period. Let frames be of fixed size for each link, also called the link *width*, and let link *length* be the duration of frame transmission in periods.

In principle, the isochrony invariant allows each node in the system to identify communication faults. At the end of each period, if a frame has not arrived on each link at each node, the node expecting the missing data can assume a fault has occurred and simply abandon the link in question to maintain communication with its other neighbors.

In practice, perfect isochrony is impossible and dealing with communication faults is more complex. The signals generated by different nodes are never identical, and transmission over any distance also imparts natural variation to a signal. Herein lies the real difficulty of achieving physical synchrony. Designing the framework to tolerate such variation over time is the focus for the rest of this document.

3.1 Jitter and drift

Assume that each node has access to an independent local *oscillator* with a frequency resolution matching at least the data rate of its widest connected link (the product of link width and communication frequency). Assume that these oscillators are used to encode the outbound transmissions of each node. The word "clock" is intentionally avoided, to highlight that only frequencies and durations are needed by the framework for timing, not any specific counter values that refer to absolute time.

Despite nominal isochrony, the waveform produced by any oscillator will vary naturally over time, due to effects of the environment on involved physical materials. Link transmission introduces similar timing irregularity upon carried data signals. Under normal conditions, let each transmitted signal be encoded by a node i at the nominal frequency of its oscillator F_i, and constrained to vary within the range $F_i \pm \varepsilon_i$, for a known finite value ε_i. Thus, the transmission frequency for a node i is:

$$f_i\,(\,t\,) \text{ s.t. } F_i - \varepsilon_i \leq f_i\,(\,t\,) \leq F_i + \varepsilon_i$$

Outside this range, the transmission is defined to have failed and may cause a communication fault. This maps well to practice, where oscillator manufacturers generally publish these performance properties for their products. Nodes are defined to operate correctly if they maintain the invariants for all neighbors with non-faulty transmissions.

Define *mesochronous* signals as those with equal average frequency, but where individual cycles in the waveform may occasionally be out of phase (not be perfectly

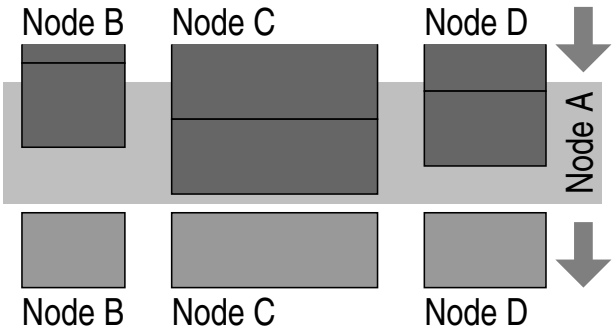

Figure 2: Visualization of some node *A* with frames arriving into its buffers from neighbors *B*, *C*, and *D*. Incoming frames are not perfectly aligned but outbound frames are. The misalignment is the result of drift.

aligned, i.e. "jittering" back and forth). Define *jitter* as the difference between mesochronous signals. Define *plesiochronous* signals as those where frequencies are nominally equal, but have diverged over time with no reasonable expectation of re-convergence. Define *drift* as the difference between plesiochronous signals. In principle jitter and drift are the same thing, the distinguishing factor being the time-frame over which frequency differences are considered.

To give an example of these terms, parallel signals sent from the same source along similar links will likely experience jitter due to physical differences in the paths, but are unlikely to experience drift since they share the same source timing. Signals from different sources have likely drifted, because oscillators are assumed to be independent. On the other hand, heterochronous signals, meaning those with nominally different frequencies, are obviously not meaningful to compare.

3.2 Loopback timing

Because of jitter and drift, the incoming signals for each link may not be perfectly synchronized with the local oscillator. Assume that some mechanism exists to tolerate this variation, capture the incoming data, and record it to a buffer. The process of recovering the timing, and thereby the data, for an independently timed signal is called "loopback timing". This allows a node to consume data frames at its own frequency. Such mechanisms are common in practice, and are normally based on phase-locked loops (PLLs).

Suppose that one frame of buffer memory is available for each incoming signal. This allows the data recovery process to tolerate jitter between bits of data smaller than frames. However, jitter that affects the timing of entire frames may still cause communication faults. If too much or too little frame data arrives during a period, additional mechanisms are needed.

3.3 Buffering

Jitter in the timing of data frames can be addressed in a straightforward manner. By adding additional receive buffer memory to each link, the length of a link can be artificially extended to create more frame arrival timing flexibility. By buffering more than just one frame, the chances can be improved that at least one frame is always available for node consumption. Jitter then simply effects frames that are only needed in the future, and thus can be ignored. The performance downside for buffering is the increase in link latency.

The quantity of available buffering determines the quantity of jitter which can be tolerated. Consider that jitter is analogous to burstiness of data in a signal. To best tolerate both bursts and idle periods, the buffer should ideally be, on average, only half full. This allows for both kinds of jitter effects, bursts and idleness, which fill or empty the buffer respectively.

Let buffers be used circularly, with consumption of frames "chasing" arriving data around the buffer. The arrival rate for bits of data into the buffer is determined by the incoming data signal. Data departs from the buffer by the node consuming one frame on each local period threshold, determined by the local oscillator.

Accordingly, the definition of communication faults can be refined to account for buffering. Let a communication fault occur either when the consuming node requires a frame that has not yet completely arrived, or when insufficient buffer is available to store arriving data without destroying data yet un-consumed. In other words, failure occurs when the producer and consumer processes operate at different frequencies for sufficient time to "collide", exceeding the tolerance created by the buffer.

3.4 Flow control

The only solution for drift is elimination. Buffering does nothing to help tolerate drift. Any persistent difference in frequency between the consumption rate of a receiving *rx* node and the rate of an incoming *tx* signal, will either under-run or over-run any finite buffer. In fact, jitter and drift are cumulative in their negative effects. Drift reduces the jitter tolerance for a buffer by changing the balance, and thus decreasing the buffer amount available for the corresponding jitter effect.

Drift can be eliminated only by changing one or both of the *rx* or *tx* frequencies. In other words, synchronizing the plesiochronous signals with flow control over time. Flow control is a common technique, and the one proposed here is similar to those used by other link-layer protocols, such as PAUSE frames in Ethernet (IEEE 802.3x) [16].

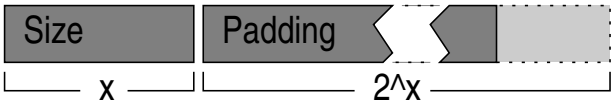

Figure 3: Format of correction frame with $\leq 2^x$ bits of padding.

Let the term *data frame* indicate what has so far simply been called a frame, meaning untyped higher-level data, and let the *effective frequency* for a signal be that of these data frames. Let the term *correction frame* indicate a new kind of frame which does not carry any useful data, but rather acts as variable sized padding. Accordingly, revise the invariants to apply only to data frames, and redefine f_i as the effective frequency of node i instead of the actual oscillator frequency.

Interleave correction frames between data frames during transmission in a fixed periodic manner, and let the correction frames be discarded immediately upon arrival. As illustrated in Figure 3, a correction frame carries only variable sized padding and a size field, in-band signaling to indicate the padding quantity.

Separating effective from actual frequency places the least restrictions on implementation platforms, where tuning the actual frequency of a signal may be impossible or at least undesirable. From the perspective of an *rx* node, only the effective frequency really matters. In summary, correction can be thought of as artificially induced drift to counteract natural drift, in a form that can be precisely controlled and thereby create a stable drift-free effective frequency.

4 Self-stabilization

Define the *common rate* for a network as the frequency of data frame transmissions which is nominally maintained by all nodes to meet the isochrony invariant. This section presents a simple algorithm, which can be applied at each node, to self-stabilize [3] the network around an approximation of the common rate by using only locally available information to estimate appropriate flow control correction over time. This algorithm is formally shown to be both robust and correct, in that it is resistant to failure and that node frequencies always converge over time.

The difficulty with flow control is that correction must be applied equally by each node to all outbound transmissions, to ensure meeting the yoke invariant. While correcting transmissions independently may be sufficient to prevent communication faults, the transmission signals might drift with respect to one another and ultimately cause some communication sequences to get illegally ahead of others. Thus, all transmissions must be corrected together, ideally to the common rate.

Because only local oscillators are available at each node, direct measurement of infidelity to the common rate is not possible. To do so would require treating the local frequency or one of the incoming data frequencies as a reference, and all of these are assumed to be imperfect and thus insufficient for this purpose. Instead, nodes can compare their own transmission frequencies to those of incoming data signals, and adjust theirs such that all node frequencies converge to a global average over time. Note that this is sufficient to meet the invariants, and thus create communication synchrony. The nominal common rate simply serves as a target, and is the middle point in the range of possible convergence frequencies.

4.1 Inverse buffer symmetry

Just as no references exist to allow direct comparison between node frequencies and the common rate, nodes cannot directly measure incoming data frequencies. To calculate the deviation between the local frequency and those of neighbor nodes, nodes must compare data arrival with local consumption over time.

Conveniently, communication synchrony creates a symmetric relationship between the buffers on opposite sides of each bidirectional link. This is best explained using the "law of conservation of messages" from Section 2.1, which ensures a "closed system" of data. If the buffer on one side of a link is short data, then the buffer on the other side must be long an equal amount of data.

Assume that buffers for a link are of equal sizes at both ends. Consider directly connected nodes i and j, with buffers of size $B_{ij} = B_{ji}$, where communication initialization leaves buffers half-filled, and where the amount of actual data in buffers at each node at period t is $b_{ji}(t)$ and $b_{ij}(t)$ respectively. Define the *imbalance* for a buffer as:

$$\phi_{ij}(t) = b_{ij} - (B_{ij} / 2)$$

The law of conservation of messages implies that the following formulas hold:

$$b_{ij} + b_{ji} = (B_{ij} + B_{ji}) / 2 \Leftrightarrow$$
$$(b_{ij} - (B_{ij} / 2)) = -(b_{ji} - (B_{ji} / 2)) \Leftrightarrow$$
$$\phi_{ij}(t) = -\phi_{ji}(t)$$

Hence, the imbalance for a buffer is the difference between its current fullness and it being half full. A buffer is perfectly balanced if the imbalance value is zero.

Notice that inverse buffer symmetry depends upon an assumption that the length-width products of associated links are constant. In practice, of course, few things are precisely constant. Natural factors may slightly change

nominally fixed link lengths. In these circumstances, attempts to achieve precise balance may actually cause nodes to trade slight imbalances back and forth. Alternatively, a threshold could be added to the calculations below to hide minor imbalances. Because this issue is not expected to be severe, it is ignored by the rest of this document.

To calculate the relationship between a node's local frequency and those of neighbor nodes, the node can exploit this buffer relationship. By observing local buffer imbalance over time, the relative deviation between the local consumption and remote transmission frequencies can be quantified, and this information used to adjust flow control correction. Buffer imbalance could be considered a form of implicit feedback, incidental shared state which is created for free, as opposed to information which requires explicit feedback communication (overhead) to acquire.

4.2 Measuring drift

Drift between local consumption and the incoming signal frequency for a link is reflected in the slope of the function of imbalance values for the corresponding buffer. If the the imbalance is constant, data arrival must match consumption, meaning the frequencies are synchronous. If imbalance is changing, the degree of change in balance over time is the difference in the two frequencies over that time.

Supposing that each node records the imbalance for all local buffers once each period, it can approximate the drift between itself and each neighbor. Of course, nodes cannot know if the drift was caused by local or remote frequency variation, but they can determine if it happens and the amount.

Assuming that drift is constant over a measurement interval, its effect on imbalance values should be that of a linear function. Hence, it should be well estimated by a linear regression across the imbalance values in that interval. For each node i, over the interval of periods from $t - m$ to t, use the imbalance values $\phi_{ij}\ (\ t\ )\ \forall\ j \in \eta_i$ to calculate the drift $\delta_{ij}\ (\ t\ )$, which is the slope component of the standard regression formula:

$$
\begin{aligned}
& t - m \leq p \leq t \\
& \phi_{avg} = (\ \textstyle\sum \phi_{ij}\ (\ p\ )\ )\ /\ m \\
& h_{top} = \textstyle\sum (\ (\ \phi_{ij}\ (\ p\ ) - \phi_{avg}\ ) \times (\ p - (\ m\ /\ 2\ )\ )\ ) \\
& h_{bot} = \textstyle\sum (\ (\ \phi_{ij}\ (\ p\ ) - \phi_{avg}\ )^2\ ) \\
& \delta_{ij}\ (\ t\ ) = h_{top}\ /\ h_{bot}
\end{aligned}
$$

4.3 Synchronization granularity

Providing synchrony in the face of natural drift is intuitively similar to the well understood problem of providing reliable communication over noisy channels [24]. In both cases, order is imposed upon a noisy or chaotic system at some overhead cost. For synchrony, the degree of order imposed is the granularity of the common rate relative to local oscillator rates. The overhead cost is a tradeoff between bandwidth sacrificed to correction and oscillator stability requirements.

Synchronization grows progressively less efficient as the ratio between the common rate and oscillator frequencies approaches parity. Synchrony at the granularity of single oscillator cycles is obviously impossible for imperfect oscillators, since they may all operate for a different number of cycles during any absolute time interval. Correction allows nodes to hide any "extra time" caused by drift, by using it to encode correction padding. Thus, the less oscillators can drift, the less correction is needed.

4.4 Calculating frame sizes

The bandwidth allowance for correction frames, and thereby for data frames, depends on the least stable oscillator in the network. Given an oscillator at node i, with frequency range for correct operation of $f_i\ (\ t\ ) \in F_i \pm \varepsilon_i$ cycles per period of the common rate, define the *instability* of the oscillator as the maximum fraction which the oscillator may be slow or fast:

$$\sigma_i = \varepsilon_i\ /\ F_i$$

An example instability value for commonly available components is $1/1000$. Define σ_{max} as the largest (meaning worst) instability in the network:

$$\sigma_{max} = max\ (\ \sigma_i\ \forall\ i \in N\ )$$

Since data is encoded at the oscillator rate, the maximum possible transmission width W_i for the node is equal to its maximum possible frequency:

$$W_i = F_i + \varepsilon_i$$

The network must operate at the rate of the weakest component with which synchrony must be maintained. Thus, a node with maximum transmission width W_i must reserve at least $W_i \times \sigma_{max}$ bits for correction padding per period. The network is like a marching army, which must proceed at the rate of the slowest soldiers or leave them behind.

Let the *correction interval* be the span of τ periods at which correction frames are interleaved. In principle correction frames could be interleaved between all pairs of data frames. However, if data frames and padding are small, the size field in correction frames becomes relatively expensive and is best amortized over more padding and a larger interval. Also, the padding field must allow

for at least two bits. Balanced operation would transmit one bit, leaving one bit of correction in either direction. Correction frames are best interleaved at intervals sufficiently large to require at least such whole numbers of padding bits. For convenience in reasoning about the framework, assume that correction intervals are equal for all transmissions.

Calculate the maximum correction frame size as the sum of maximum padding bits and sufficient size field bits to quantify the padding. Specifically, for a link of maximum width W_i, the maximum padding Φ_i and resulting correction frame size C_i are:

$$P_i = \tau \times W_i \times \sigma_{max}$$
$$C_i = P_i + log_2 (P_i)$$

Calculate the corresponding data frame size D_i to utilize the remaining capacity:

$$D_i = ((\tau \times W_i) - C_i) / \tau$$

These frame size definitions allow for correction to occupy all the "extra" cycles in the range of natural variation. Although the calculation above implicitly assumes that all links have maximum width to exactly match oscillator frequencies, consider that smaller links could be supported by scaling oscillator frequencies to allow multiple W_i values. However, the fraction of bandwidth reserved for correction on each of these links must remain fixed, as explained above.

4.5 Average neighbor algorithm

Given the above techniques for measuring and correcting frequencies, this algorithm is a decentralized process for choosing correction amounts, which causes global convergence of transmission frequencies. Each node iteratively seeks an outgoing frequency that matches the average across incoming data frequencies.

Consider a node *regular* if its buffers are balanced and each adjacent link has zero drift. Nodes must constantly seek regularity by correcting for drift as it varies over time. Correction must also serve the dual purposes of restoring buffer balance as it becomes disturbed by drift.

Since both nodes sharing a link will apply the same algorithm, it is important that both nodes do not each correct the entire drift. Correction is cumulative for a link, and this would likely result in more drift caused by overcompensation. A conservative approach is for both nodes to assume equal sharing, and themselves attempt only half of any necessary correction.

Define the *measurement interval* as the span of m_c correction intervals, thus $m = m_c \times \tau$ periods, over which drift is estimated and between which correction is calculated appropriately. This value should be chosen to match the relationship between buffer sizes and oscillator instability. The interval should be long enough to allow for meaningful measurement of drift, but not long enough for it to have significant impact.

The aim of correction is to maintain zero imbalance. For each link between nodes i and j, at the start period t of each measurement interval, use the techniques above to estimate the drift $\delta_{ij} (t)$ over the past m periods. Let $\phi_{ji} (t)$ be the buffer imbalance at node i. The estimated drift will cause an expected imbalance during the next interval of $\delta_{ij} (t) \times m$. This allows calculation of an aggregate padding for the coming measurement interval to create enough drift to correct any imbalance. If imbalance is zero, simply choose padding to counteract any measured drift. Thus, to correct drift and existing buffer imbalance, choose per period padding between node i and each adjacent neighbor j:

$$\rho_{ij} (t) = ((\delta_{ij} (t) \times m) + \phi_{ji} (t)) / 2m$$

However, all transmissions at each node must be corrected equivalently to prevent relative drift. As its name suggests, the average neighbor algorithm uses an averaging process to resolve the potentially differing correction demands between links. Let the $\alpha_i (t)$ be the actual padding for the correction frames following time t between node i and all of its neighbors j:

$$\alpha_i (t) = \lceil (\sum \rho_{ij} (t) \forall j \in \eta_i) / card (\eta_i) \rceil$$

In summary, each node calculates correction padding for each adjacent link for each measurement interval independently, averages these padding results together, and then uses one half of this average. This approach allows each node to iteratively seek regularity in a completely decentralized manner with only local information. To extend this process for links of differing widths, the ρ_{ij} values can be normalized, then averaged together, and finally the average re-scaled to match the width of each link. The algebra to support this was omitted here to simplify reasoning about the system.

Consider that any algorithm for calculation of correction is sufficient if it converges and avoids communication failure. It seems likely that other such algorithms exist. This document simply proposes one algorithm, which achieves the purpose of establishing proper correction and avoiding failure. There is no claim to optimality – there may exist better algorithms, for example with lower buffering requirements.

4.6 Convergence

For the average neighbor algorithm to be practical and correct, it must ensure that the transmission frequency and

the input signal frequencies converge at all nodes within finite time. If convergence always occurs in finite time, there exists a finite amount of buffering sufficient to ensure no data is lost during the convergence process.

The formal proof of convergence can be found in Section 7. It uses the following intuition. During each correction interval, all those nodes which are "outliers" must be improved. Outlier nodes are those which have either uniform positive or negative frequency differences with their neighbors. Because correction is updated during each iteration to match the average of neighboring rates, the degree of difference for these nodes must decrease. Since this property is true for each interval, and since the amount of decrease must be discrete, convergence in finite time is guaranteed. The proof depends upon a couple of assumptions, namely that drift amounts do not change while the algorithm is adapting to them, and that estimation of drift is accurate.

Notice that the proof only guarantees convergence in finite time. This is however sufficient to show communication robustness, in that finite buffers can prevent drift from causing communication failure. While finite does not necessarily mean small, consider that small buffers are likely to be sufficient in practice. Drift is generally a gradual process, occurring at the timescale of at least seconds. For gradual drift in localized regions of the network, the algorithm is likely to correct it within a single correction interval under good circumstances. The proof primarily serves to show that no diabolical situations exist which can cause collapse of all network communication.

4.7 Failure isolation

The average neighbor algorithm is robust, in that it generally prevents failures from cascading and causing additional failures elsewhere in the network. Proving that failures are isolated is simpler than proving convergence.

All faults must appear as communication faults to properly functioning *good* nodes, meaning the the over or under-flow of a communication buffer. Actual failures may include physical damage to links and nodes, or malfunction of an oscillator – with the result that transmissions from the associated node occur at frequencies outside of the allowed range. All of these failures cause communication faults at adjacent good nodes.

Robustness implies that one communication fault may not artificially cause another by negatively affecting the transmission frequency of good nodes. Until the actual fault occurs, the buffer imbalance associated with the failure cannot be distinguished from simple drift and will affect the calculation of correction amounts and thus transmission frequency. However, for a node with n neighbors, each imbalanced buffer can only change the correction amount by $1/n$ of the maximal amount. Meanwhile, the degree of drift for the bad neighbors is greater than that for any of the good neighbors by definition. Thus, a fault must occur, the failure be detected, and the associated communication abandoned before additional artificial faults occur.

Notice that managing failure for the average neighbor algorithm is purely local. To guarantee an artificial fault, communication with more than half of the neighbors at a node must fail simultaneously. This is in contrast to algorithms which depend on shared state, meaning they face potential Byzantine failures, and thus require distributed consensus for robustness.

5 Simulation results

To quantify the potential performance of the communication synchrony framework, some simple and some challenging scenarios are evaluated more closely using simulation. The results confirm expectations that drift can be mitigated rapidly, even for highly inconvenient network topologies.

This section is not intended to be exhaustive. The many degrees of freedom allowed by the framework prevent doing so with any brevity. Instead, interesting topologies are selected to illustrate potential good and bad case behavior. Specifically, consider very tightly and very sparsely connected networks. The more connected a network is, the more feedback is generally available to the average neighbor algorithm about the global average rate. Conversely, when nodes are far apart and only weakly connected, incompatibility between their rates may take a long time to be noticed and reconciled. The prototypical network topologies for these cases are stars and chains respectively.

All of the following simulation trials reflect equal node parameters:

$D_i = 1000$ bits (data frame size)
$B_{ij} = 2 \times D_i$ bits (buffer size per link)
$\sigma_i = 0.001$ bits per period (instability)
$m = 100$ periods (measurement interval)

Furthermore, all links are assumed to be of zero length. These parameters are fixed to limit the quantity of results and simplify reasoning about them. Except for link length, which plays little role so long as it remains a small fraction of m, the other values are believable values. The goal of simulation is primarily to show that reasonable network configurations lead to desirable algorithm behavior.

Drift is modeled in two ways. The effective frequencies of all nodes are randomized (within the valid range) at the

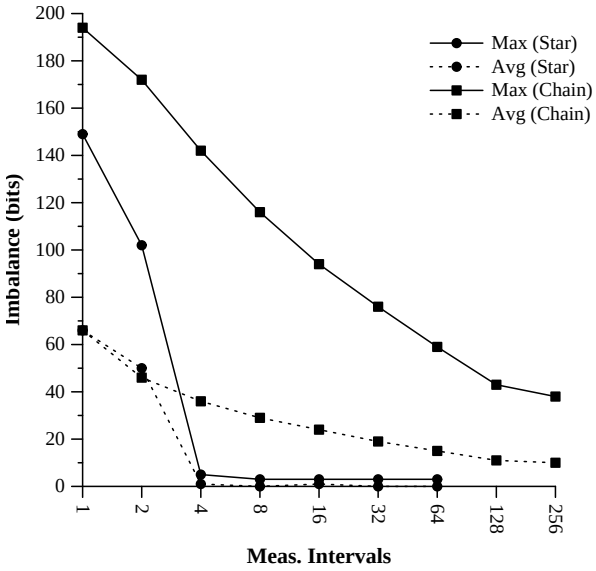

Figure 4: Effective frequency convergence for both chain and star topologies of 1001 nodes each, with randomized frequencies but no drift. Each data point is the average over 100 trials. The absolute maximum imbalance seen in any trial was 200 bits.

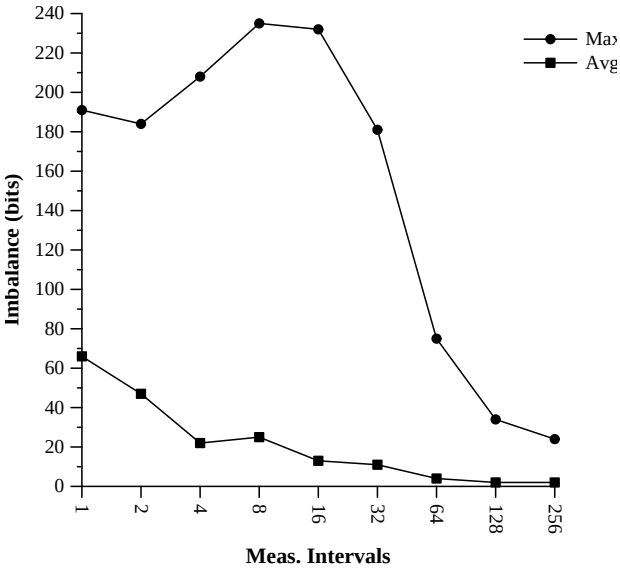

Figure 5: Effective frequency convergence for a tree topology with 426 nodes, with all branches of degree 5, and with randomized frequencies but no drift. Each data point is the average over 100 trials. The absolute maximum imbalance seen in any trial was 280 bits.

start of each trial. For the first set of trials, convergence is observed without any additional frequency variation. The second set models ongoing drift by re-randomizing the frequency of each node with 1% probability once each measurement interval. This crude drift model is intended to be simple but conservative, as it is likely much more volatile than drift in practice.

Figure 4 shows the behavior of the algorithm for the two prototypical topology types, over trials of up to 256 measurement intervals. The star is a root node with 1000 neighbors, all of which have degree 1. The chain is effectively a linked list of 1001 nodes, with all but the endpoints of degree 2. As expected, convergence for the star is very rapid, while the chain is slower. However, it is gratifying to see that, even for a chain of this great length, the imbalance is rapidly brought to manageable levels.

Figure 5 shows a more complex tree topology, to indicate behavior for more complex networks, again for up to 256 measurement intervals. Only tree topologies are considered as they show conservative behavior – more connected networks are less challenging, as demonstrated by the star example. This tree is formed such that the root, and all internal branch nodes, have degree 5.

The final simulation imposes the re-randomizing drift model on the tree topology. A simulation interval of 100,000 measurement intervals is used to capture latent negative effects. The result for a single trial has the absolute maximum imbalance of 560 bits, with per-trial averages of 163 bits maximum and 15 bits average. 427224 instances of drift re-randomization occurred.

6 Conclusion

The contribution of this document is a framework for synchronizing all communication in a distributed system to efficiently create shared logical time. This framework is shown to be highly robust, operating in a decentralized manner and tolerating multiple simultaneous failures, and being structured such that Byzantine failure is meaningless. The framework is further shown to impose minimal communication overhead. Simulation results indicate that likely costs are only a small fraction of the bandwidth on each communication link, and that only two data frames of buffering are needed to tolerate the drift caused by common oscillators.

This efficiency is closely tied to the primary drawback of the framework, which is the requirement for implementation at the low level, with direct control over deterministic communication resources. Low-level implementation allows for implicit synchronization signals and implicit feedback about the relative timing of nodes, which is in contrast to higher level overlay approaches that must communicate all information explicitly.

Aside from this implementation constraint, the platform requirements of the framework are quite general and implementation across a wide range of scales is possible,

both in terms of number of nodes as well as granularity of the synchronization frequency. For example, the framework can be adapted to tolerate arbitrary clock instability, allowing use of inexpensive components. The degree of overhead introduced by the framework scales inversely with the quality of the platform and with the expected precision of synchrony. This is similar to the information theoretic tradeoff between overhead costs and the signal-noise ratio of communication channels.

Given the extent to which the advantages of communication synchrony outweigh the disadvantages, the primary limiting factor for immediate widespread adoption of synchronous communication is the significant difference in philosophical approach between the proposed synchrony and the prevailing asynchrony. For some similar technologies, such as SONET, the improved efficiency and robustness available from the framework may make adoption straightforward. For other networks, however, consider that the framework specifies only a link-layer architecture. This allows seamless and incremental deployment beneath existing higher level architectures, such as the Internet. Exposing the superior features of synchrony can thus be made orthogonal to adoption.

7 Convergence proof

A more formal proof for the claim from Section 4.6 is included here to show the formal power which communication synchrony can provide. The proof assumes a network which has experienced some drift but is still synchronous. Hence, all nodes share the same timeline, allowing formal reasoning about node relationships at precise discrete times. Proving properties with this level of complexity about asynchronous systems is often either impossible or requires orders of magnitude more logic. Thus this proof demonstrates (in addition to its formal argument) how synchrony allows for greater objective rigor in a domain currently plagued by imprecise or subjective analysis.

HYPOTHESIS: Within finite time, the frequency of each node in the network will converge to match a global network average:

$$R_{avg}(t) = (\sum(f_i(t) \forall i \in N)) / card(N)$$

Formally, for $t \to \infty$ and $\forall i \in N$,

$$f_i(t) \to R_{avg}(t)$$

PROOF: Consider the properties of the algorithm for *outlier* nodes, those whose frequencies are maximally deviant from R_{avg} at any time. Define the set of such nodes O as follows:

$$O(t) = \{ i \mid i \in N \wedge (\Delta_i(t) \geq \Delta_j(t) \forall j \in N) \}$$

Where Δ_i is the frequency deviation for node i from the average:

$$\Delta_i(t) = | f_i(t) - R_{avg}(t) |$$

Consider three cases. 1. The outlier set may be empty. 2. The outlier set may be a singleton. 3. There may exist multiple outlier nodes.

CASE 1: If $O(t) = \emptyset$ then obviously all nodes must have the same frequency, equal to the average, and the hypothesis is met. Notice that $O(t) \neq N$, since equality would imply that all frequencies are equal to one another but not to the average, a contradiction.

CASE 2: If $O(t)$ is a singleton set, containing only node i, then i is either the fastest or slowest node, meaning that drift with all neighbors must be in the same direction as R_{avg}, meaning that i will correct in that direction. Supposing that i is fast, then:

$$f_i(t) > R_{avg}(t) \Rightarrow$$
$$f_i(t) - R_{avg}(t) = | f_i(t) - R_{avg}(t) |$$

It is known that, by i being the only node in $O(t)$:

$$\forall j \in N, j \neq i$$
$$\Delta_i(t) > \Delta_j(t) \Rightarrow$$
$$f_i(t) > f_j(t)$$

Assume that δ and ϕ have the same sign for each link, meaning that current imbalance can be explained by existing drift. Assume also that correction works correctly, meaning:

$$f_i(t+1) = f_i(t) \times \alpha_i(t)$$

Which allows:

$$\rho_{ij}(t) < 0 \Rightarrow$$
$$\alpha_i(t) < 0 \Rightarrow$$
$$f_i(t+1) < f_i(t)$$

If, instead $f_i(t) < R_{avg}(t)$, meaning that node i is slow. The same reasoning can be turned around to show that i must speed up.
CASE 3: $O(t) \neq N$ implies that some node $i \in O$ must have at least one neighbor $j \notin O$. By reasoning quite similar to that in CASE 2, including the need to consider both fast and slow possibilities, it can be shown that i must change speed to approach $R_{avg}(t)$. For example, if i is fast:

$$f_i(t) > f_j(t) \Rightarrow$$
$$\alpha_i(t) < \alpha_i(t-1) \Rightarrow$$
$$f_i(t+1) < f_i(t)$$

CONCLUSION: In CASE 1, the hypothesis is simply met. For CASE 2, the maximum Δ_i value in the system is reduced. For CASE 3,

$$card(O(t+1)) < card(O(t))$$

Between these two constraints, and given that correction is discrete

$$card(O(t)) \rightarrow 0 \quad \text{as} \quad t \rightarrow \infty$$

proving the hypothesis.

References

[1] Baruch Awerbuch. Complexity of network synchronization. *Journal of the Association for Computing Machinery (JACM)*, 32(4):804–823, October 1985.

[2] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 123–138, Austin, TX, November 1987.

[3] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the Association for Computing Machinery (CACM)*, 17(11):643–644, November 1974.

[4] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the Association for Computing Machinery (JACM)*, 34(1):77–97, January 1987.

[5] Danny Dolev, Joseph Y. Halpern, Barbara Simons, and Ray Strong. Dynamic fault-tolerant clock synchronization. *Journal of the Association for Computing Machinery (JACM)*, 42(2):143–185, January 1995.

[6] Danny Dolev, Joseph Y. Halpern, and H. Ray Strong. On the possibility and impossibility of achieving clock synchronization. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 504–511, Washington D.C., April 1984.

[7] E. Allen Emerson. Temporal and modal logic. In *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 995–1072. The MIT Press, Cambridge, MA, 1991.

[8] Michael J. Fischer, Nancy A. Lynch, and Michaael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinery (JACM)*, 32(2):374–382, April 1985.

[9] Van Jacobson. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM*, pages 314–329, Stanford, CA, August 1988.

[10] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, July 1985.

[11] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the Association for Computing Machinery (CACM)*, 21(7):558–565, July 1978.

[12] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(2):254–280, April 1984.

[13] Leslie Lamport and P.M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the Association for Computing Machinery (JACM)*, 32(1):52–78, January 1985.

[14] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *Journal of the Association for Computing Machinery (JACM)*, 30(3):668–676, July 1983.

[15] David G. Messerschmitt. Synchronization in digital system design. *IEEE Journal on Selected Areas in Communications*, 8(8):1404–1419, October 1990.

[16] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the Association for Computing Machinery (CACM)*, 19(7):395–404, July 1976.

[17] David L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communication*, 39(10):1482–1493, October 1991.

[18] Boaz Patt-Shamir and Sergio Rajsbaum. A theory of clock synchronization (extended abstract). In *Proceedings of the ACM Symposium on Theory of Computing*, pages 810–819, Montreal, Quebec, Canada, May 1994.

[19] Larry L. Peterson, Nick C. Buchholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication.

ACM Transactions on Computer Systems, 7(3):217–246, August 1989.

[20] Jon Postel. Transmission control protocol (TCP). Request for Comments (RFC) 793, Internet Engineering Task Force (IETF), September 1981.

[21] Parameswaran Ramanathan, Kang G. Shin, and Ricky W. Butler. Fault-tolerant clock synchronization in distributed systems. *IEEE Computer*, 23(10):33–42, October 1990.

[22] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[23] Fred B. Schneider. Synchronization in distributed programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):125–148, April 1982.

[24] Claude Elwood Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 1948.

[25] Telcordia Technologies, Inc. Synchronous optical network (SONET) transport systems: Common generic criteria. Document Number GR-253, September 2000.

[26] Hubert Zimmermann. OSI reference model – the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communication*, 28:425–432, April 1980.