ABSTRACT

Title of Dissertation: ACHIEVING GLOBAL SYNCHRONY IN A DISTRIBUTED SYSTEM WITH FREE-RUNNING LOCAL CLOCKS

Michael Bao Trinh, Doctor of Philosophy, 2006

Dissertation directed by: Prof. Ashok Agrawala
Department of Computer Science

In any distributed system, there is an intrinsic need to coordinate the events between the nodes. In such a system, even though each individual node only has access to its local clock, global coordination often has to be carried out the basis of time, called the global time, that is common to every nodes. One way to achieve this globally synchronous behavior is to synchronize all local clocks with an external time source such as a Cesium clock. Local time then becomes in effect global time. There are, however, several drawbacks to such clock synchronization method. Highly precise clocks are expensive and can add to the cost of the hardware node. Some approaches depend on a certain network characteristic such as a symmetric latency, a broadcast medium, or a bus structure. Many are also centralized in that they assume the existence of one or more master clocks to which the remaining clocks synchronize with, and this adds additional complexity

when the master node fails and/or when the network has to reorganize in order to select another master.

In this dissertation, we introduce a new method for achieving global synchrony without performing clock synchronization. In our approach, called the Cyclone Network Synchronization (CNS) scheme, the local clocks are free-running and are not modified in any way. CNS relies on the ability of each node to send data at a time of its choosing. Such data are sent at regular interval, with the next instance being determined based only on the local information available at the node. Once the scheme converges, the interval for all nodes becomes exactly the same, supporting a synchronous operation across the whole network. CNS takes into account the finite precision arithmetic and measurements it has to use, while still maintaining global synchrony with very small jitter values.

The scheme can be used in many synchronous cyclic networks, and does not require a broadcast medium or depend on a symmetric latency. CNS is a decentralized scheme with no master server, as all of the nodes execute the same set of instructions, and can tolerate most topology changes without the need to reconfigure. There is very little overhead since no explicit synchronization messages are sent. A high degree of accuracy can be achieved with the algorithm, and both clock drift as well as latency perturbation are tolerated. Furthermore, this accuracy is not a function of the clock drift rate, as is the case for most clock synchronization approaches.

ACHIEVING GLOBAL SYNCHRONY IN A DISTRIBUTED SYSTEM WITH FREE-RUNNING LOCAL CLOCKS

by

Michael Bao Trinh

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

DEDICATION

To my family, thank you for waiting.

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the support and guidance of my mentor and advisor, Dr. Ashok Agrawala.

I also wish to thank Dr. Douglas Szajda for his help with the analysis aspect.

And finally, thank you to all of the friends I have met at College Park. It has been a wonderful journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

In any distributed system, there is an intrinsic need to coordinate the occurrences of events at various nodes [38]. This synchronization requirement is particularly important for high-speed synchronous cyclic networks [27, 32, 8], especially those that support real-time and/or stringent quality of service (QOS) applications [10, 47]. An example of such system is Cyclone [37], a network which requires that all resources be known and scheduled in advanced of actual usage. These network resources include not only the traditional buffers, but the time instances at which these buffers are being moved from one node to another. In other words, the sends and receives in Cyclone must be coordinated ahead of time and with as much precision as possible, in order to maximize the effectiveness of the overall network.

Global coordination must be done with respect to a common basis of time, logically called the "global clock", that is visible to all nodes in the system. However, the only clock that is usually available at each node is the "local clock", which consists of a quartz crystal and a clock counter [20]. The crystal oscillates at some nominal frequency, e.g. 1 MHz (one million times per second), generating a pulse every microsecond which is used to increment the clock counter. Time readings at each node are performed simply by accessing the value of this clock counter. While the oscillating frequencies of various clock oscillators are expected to be the same, in practice they do differ, making two clocks drift with respect to each other, or to the global clock, which is considered to be perfect[1]. Furthermore, since the frequency of the crystal can also fluctuate over time depending on external factors such as temperature changes, the local clock at a node is further affected. It is not uncommon to have drift rates of say 10 parts per million (10 PPM or 10^{-5} [45]) for the clocks used in most PCs. As a result, local clocks, with typical drift rate in this range, cannot be used to achieve global synchrony without additional steps. We note while that most local clock systems permit the modification of the clock counter, they rarely, if ever, support the adjustment of the oscillating frequency.

Local clock synchronization is one way to achieve the desired globally synchronous behavior. Here, the local clocks at each node are synchronized with each other, usually by modifying the clock counters accordingly, such that a time reading by one local clock will not differ by more than some fixed amount from that read by another local clock at any given instance. More specifically, one of the local clocks will be synchronized with an attached "external clock", and the remaining nodes will synchronize with this special node. External clocks such as a Cesium clock or a GPS clock are considered to be highly accurate in that their drift rates are very small ($< 10^{-12}$ [2]) over time compared to the global clock. With clock synchronization, local time then becomes in effect equivalent to global time. There are, however, various drawbacks to the clock synchronization method. Highly accurate clocks are expensive, and can add to the cost of

[1]By perfect, we mean a clock which maintains the true real time, e.g. UTC.

the hardware node. Also, the signals have to propagate from the master clock to the local clock, taking finite amount of time and contributing to some degree of temporal uncertainty. Some approaches depend on a certain network characteristic such as a symmetric latency, a broadcast medium, or a bus structure. The existence of one or more external clocks, acting as master clocks to which the remaining local clocks synchronize to, also implies a centralized scheme. Additional complexity are incurred when these master clocks (or the nodes they are attached to) fail and/or when the network has to reconfigure in order to select/locate another master clock.

1.1 Contributions

In this dissertation, we introduce a new method for achieving global synchrony in a distributed system. Our technique, called the Cyclone Network Synchronization (CNS) scheme, does not require the local clocks to be synchronized. CNS relies on the ability of each node to send data at a time of its choosing. Such data are sent at regular interval, with the next instance being determined based only on the local information available at the node. Once the scheme converges, the interval for all nodes becomes exactly the same, supporting a synchronous operation across the whole network. CNS takes into account the finite precision arithmetic and measurements it has to use, while still maintaining global synchrony with very small jitter values. The scheme can be used for event coordination in synchronous high-speed cyclic networks such as Cyclone [37], where timing knowledge of all network resources such as buffer sends and receives must be available a priori, in order to provide real-time and/or quality of service guarantees. CNS provides the following innovations and advantages over traditional clock synchronization approaches:

- In CNS, the local clocks are free-running, and thus can drift at their own rate. There is no synchronization of these local clocks, either among themselves or with external clocks.

- Since CNS does not perform clock synchronization, it does not incur any overhead in message passing required for this task, and is therefore extremely light-weight. Synchronization among the nodes is achieved solely based on the regular periodic network traffic.

- CNS is a decentralized scheme which is highly scalable, with no special distinction between the different nodes. All of them execute the same set of instructions. In case of a node failure, there is no need to reconfigure the network, as is usually the case with approaches using a centralized setup.

- CNS does not depend on specialized or expensive hardware components [31], and can be implemented using standard ones (e.g. timestamp counter). The algorithm also does not assume any particular network characteristic such as a broadcast medium or a bus topology. In addition, there is no symmetric latency requirement (where the latency on the forward path is the same as that of the return path between any two nodes) or bidirectional connection requirement.

- CNS can provide a very high degree of synchronization accuracy, which is not dependent on the drift rates of the local clocks, as is the case for most existing clock synchronization approaches. The algorithm can tolerate jitters associated with both clock drift values as well as latency values.

1.2 Organization

The remaining chapters of the dissertation are organized as follow. In Chapter 2, because the CNS scheme was developed as part of the Cyclone project, we briefly describe aspects of the Cyclone architecture that are relevant to the presentation and discussion of CNS. In Chapter 3, we first provide the intuition behind the working principle of CNS. We assume a distributed computing system with infinite precision and negligible latency delay, and show how timing on such systems can be synchronized without the need for highly accurate clock or expensive hardware. We then remove these assumptions, formally describe CNS in details, and provide an analysis of the convergence behaviors of the scheme. In Chapter 4, two enhancements to the basic scheme that help facilitate the implementation on actual hardware are discussed. In Chapter 5, we present our simulation results, showing the degree of accuracy that can be achieved with CNS, even in the presence of clock and latency perturbations. In Chapter 6, we compare and contrast CNS with some current clock synchronization approaches. Finally, the concluding remarks and opened questions are presented in Chapter 7.

Chapter 2

Background and Motivation

In this chapter, we present a brief overview of the Cyclone Network [37], describe aspects of its design that are relevant to the dissertation, and demonstrate why the synchronization provided by the CNS scheme is a critical part of the overall Cyclone architecture. Our objective is to show that Cyclone is an example of the type of network that can benefit from using a scheme such as that of CNS, as well as being the motivation behind its development.

2.1 Cyclone Technology

A large number of performance problems that computer networks suffer today are the consequence of resource contention at network nodes. This contention is a direct result of resources requested by the traffic and resource allocation policies employed at the nodes to handle the traffic. The current practice is to use on-demand, priority-based, and event-based management, which inherently leads to resource saturation, congestion, loss, and jitter problems. An alternative is to use time-based resource management. Cyclone technology, designed end-to-end in time-based manner, carries out time-based resource management in a synchronous manner. Any connection that exploits time-based resource management of Cyclone, called scheduled traffic, reserves the use of resources in time and space. As a consequence, there are no losses, jitters, or contentions for any resources. In order for time-based approaches to function properly on the components of a network, a high degree of synchronization is essential. The CNS scheme proposed in this dissertation meets the need of these time-based approaches. Note that CNS can also be used in other networks which may not have the exact same characteristics as Cyclone, yet still have a need for accurate timing requirements (e.g. Sonet [19], DTM [9]).

2.2 Components

A Cyclone network consists of a collection of nodes, called Cyclonode and connected via unidirectional links, operating in a strictly time *controlled manner.* In Cyclone, the traffic with strictly defined timing requirement is called *scheduled traffic.* Each connection carrying scheduled traffic goes through connection establishment and tear down processes. In addition, Cyclone also handles traffic without timing requirement, called *on-demand traffic.* Incoming data are temporarily placed in a buffer. The operations of an outgoing link are controlled by a *calendar* maintained at a node. Entries in the calendar specify the time and the location of the data that has to be moved during that time.

For the Cyclone network, there are two invariant quantities, a *chunk* and a *period.* A chunk is defined in terms of a fixed number of bytes, and a transmission period is defined in terms of a fixed length of time required to send a fixed number of chunks. In order to simplify the design of Cyclone network, all data is organized, managed, and moved in terms of chunks, and all operations at a node are organized in terms of a period. Chunks arriving at a node may have to wait in a buffer for transmission on an outgoing link. Conceptually, a buffer is partitioned so that each partition, called a *slot*, can store one chunk. Consider continuous writing of chunks into sequential slots of a buffer. We can then assign a unique time instance to each slot based on the time at which the writing of data in the slot begins. The time associated with each slot is referred to as the *time tag* of the slot. The time tag difference between any two consecutive slots is called a *slot time*, which clearly depends on the speed of the link.

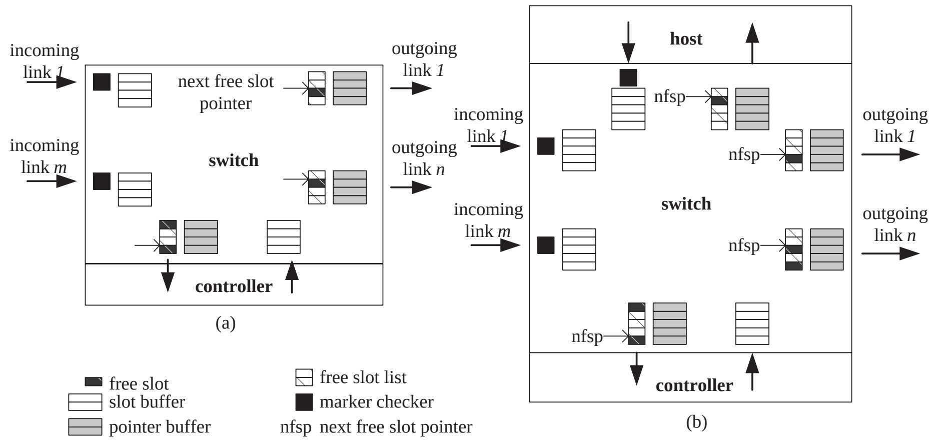

Figure 2.1: Cyclonode

Figure 2.1(a) shows the basic structure of an m x n cyclonode, which consists of two parts: a *switch* and a *controller*. For each incoming link, a *slot buffer* is assigned so that arriving chunks are stored in successive slots of the slot buffer that is treated as a circular buffer. For each outgoing link, there is a *pointer buffer*, each entry of which stores the address of a slot in a slot buffer. The *free slot* list maintains the list of slots that are not scheduled. The *next free slot* is the first entry in the free slot list. On-demand traffic and control chunks utilize unscheduled slots of the outgoing link of interest. A *marker checker* checks the first byte of a chunk, the *marker*, as it arrives at a node. The node carries out different actions based on the type of the chunk. The switch handles communications with the controller in the same way it handles an incoming and an outgoing link by maintaining appropriate buffers for them. The controller is responsible for processing control chunks, managing connections, handling failure, maintaining routing information, and maintaining synchronization[1]. Figure 2.1(b) shows the basic design of an m x n temporal regulator consisting of three parts: a host, a switch, and a controller. A temporal regulator is a cyclonode with a host attached to it. The way the switch handles communications with the host is identical to the way it handles communication with the controller.

2.3 Operations

2.3.1 Transfer of Data

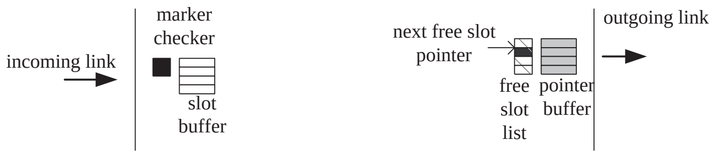

Figure 2.2: Data Transfer

The pointer buffer is essentially the schedule for the outgoing link. Each slot in the pointer buffer contains the pointer to the slot buffer whose content has to be moved by the outgoing link at the time corresponding to the time of that slot in the pointer buffer. Once that time has elapsed and the switch has taken the appropriate actions, the content of the slot in the pointer buffer is updated to reflect the time that the slot is to be used again. The free slot list is updated

[1]This includes performing the operations required by the CNS scheme.

to reflect the usage of time slots on the outgoing link when the pointer buffer is updated. The next free slot pointer also advances to the next unscheduled time slot on the outgoing link. When a scheduled traffic chunk arrives, no further action is required since there already is a pointer set up in the pointer buffer of the outgoing link for this scheduled chunk.

When the controller is ready to send a control chunk on an outgoing link, it first identifies the outgoing link to use. The controller checks if there is available slot in the pointer buffer for the outgoing link. It then writes the chunk in its slot buffer. Otherwise, the controller keeps the chunk in its internal buffers. As described in Section 2.2, one time slot in each period is reserved for control chunks' use only. The controller makes an entry in the pointer buffer of the outgoing link using reserved slot or the next free slot whichever comes first. When the host is ready to send, it writes the chunk into its slot buffer. Thereafter, the operations are the same. When a chunk arrives for the host, it is handled in the same way except that the pointer buffer for the host is used instead of a pointer buffer of an outgoing link.

2.3.2 Scheduling

In scheduling, each Cyclonode carries out all operations with respect to its view of time (i.e. according to its local clock). Cyclone technology completely supports such local views. The only requirement is that the variability in the phases of the local clocks as seen by the node be bounded and known to the node[2].

Figure 2.3 shows the time line of an incoming link (a) and examples of time lines of two possible outgoing links (b,c). A bit arriving at a node via this link is

[2]This is a function of the CNS scheme.

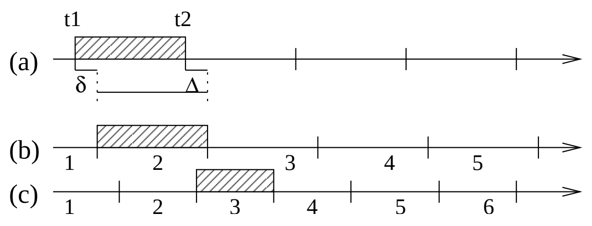

Figure 2.3: Timeline

eligible for outgoing transmission after δ, where $\delta = \Delta$. The value of δ (or Δ) is selected taking into consideration the clock jitter and hardware characteristics. The value of δ may not be the same as Δ when incoming and outgoing link speeds are different. Thus the chunk in Figure 2.3(a) can be scheduled for transmission as early as the 2^{nd} time slot in Figure 2.3(b). However, the 3^{rd} time slot in Figure 2.3(c) is the first time slot available since the 2^{nd} time slot ends before the later portion of the chunk becomes eligible.

Recall that there is finite buffer space for each incoming link, and the buffer space is used circularly to store chunks. Therefore, a chunk must be removed from this buffer within the number of time slots corresponding to the finite buffer space. Once the first chunk of a period is scheduled on the outgoing link, all subsequent chunks for this connection within this period must be scheduled within the number of time slots corresponding to the buffer size. When a new request comes, the calendar of the outgoing link is examined taking into account scheduling conditions described above. For each incoming slot in the request, the first available and eligible slot for outgoing transmission is assigned. This is consistently done for the available buffer space.

2.4 Synchronization

From the description above, it is evident that accurate timing is an important, if not the most important, aspect of the Cyclone network, especially since routing information are encoded implicitly in the arrival and departure time of a chunk. If the Cyclonodes are not sufficiently synchronized, and the arrival time of a chunk is miscalculated, there will simply be no way to determine this error after the fact. If chunk N arrived at the wrong time, say at the time when N+1 is supposed to be arriving, then chunk N will simply be routed to where chunk N+1 would have been sent. To make matter worse, this timing error will cascade indefinitely from this point onward (e.g. chunk N+1 will be sent to where chunk N+2 would have been sent, etc.), as the Cyclone network is simply not designed to handle this type of timing failure.

Accurate timing is also important when it comes to the scheduling of resources, or buffers, at a Cyclonode. The more precise the timing, e.g. if we know a chunk is scheduled to arrive between 4:59 and 5:01 instead of between 4:30 and 5:30, the more flexibility we have in making reservations in the calendars, e.g. we can schedule this chunk to be sent out anytime after 5:01 instead of having to wait until after 5:30. Better schedules can translate to potentially more connections being accepted at setup time and smaller end-to-end delays in the network. In particular, the δ value shown in Figure 2.3(a) should be as predictable as possible, so that the timing relationship between the incoming and outgoing links can be established ahead of time.

By using the Cyclone Network Synchronization scheme present in this dissertation, each Cyclonode can coordinate with all the other nodes in the network the time at which chunks are scheduled to be sent and received within a period. For synchronization purposes, we introduce the concept of a **cycle**, which consists of the *data transmission period*, followed by an *adjustment gap* (Figure 2.4). Since the chunks in a given cycle are transmitted back to back, and since a Cyclone node will determine when to transmit on its outgoing links ("outgoing cycles"), each node only needs to determine the time when the first chunk of a cycle will arrive on each of its incoming link ("incoming cycles"). The arrival/departure time of the first chunk in a cycle is also referred to as the **start time** of that incoming/outgoing cycle.

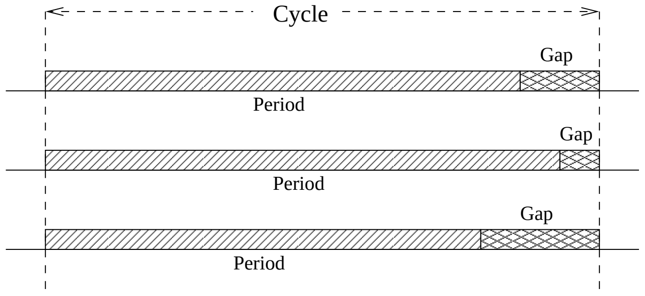

Figure 2.4: Cycle

Conceptually, the CNS scheme allows each Cyclone node to (i) determine the start times of all the incoming cycles, and based on these info, (ii) determine an appropriate start times of all the outgoing cycles. This is done by modifying the *adjustment gap* in each cycle accordingly, such that the resulting cycle length (*period* plus *adjustment*) is exactly the same across the entire Cyclone network, as measured by the *global clock*. Specifically, let C denotes the cycle length and let P denotes the length of the data transmission period. Since P is the time requires to transmit a fixed number of chunks, its value as measured by the local clock at each node will be the same. In terms of the global clock time, however, P will vary from node to node due to the different clock drift rates. In contrast, once convergence has been reached, the C value, though different at each node when measured in terms of the local clock, will be the same value when considers in terms of the global clock. This is how “global synchronization” is thus obtained by the CNS scheme.

Finally, we note once again that, in CNS, only locally available information are used, in the sense that there is no explicit exchanging of global information by the nodes. Furthermore, the synchronization scheme has to work despite the fact that the clocks on the individual Cyclone nodes are free-running and not synchronized in any way, are not required to be highly accurate (e.g. Cesium clocks), and can potentially drift at different rates (although for practical operations we do require that the clock jitter, or drift rate perturbation, be bounded).

Chapter 3

Description and Analysis

In this chapter, we give a detailed description of the basic algorithm in the Cyclone Network Synchronization scheme. We begin by specifying the network and the clock models. We then provide an high-level overview of the algorithm as well as the intuition behind its working principle. Next, we formally state the algorithm, and follow with analysis on both the converged cycle length as well as the convergence behavior.

3.1 Network Model

We assume that nodes are connected with point-to-point uni-directional links. Operations on any particular node are controlled by the clock local to that node. The clock drift rate is not known to the node. All links send or receive continuously, and that a node can record the arrival time of any bit, as reported by its local clock[1].

We assume that the graph that represents the Cyclone network topology is connected, i.e. that there is a path from any node to any other node. Network

[1]Our algorithm actually only depends on the ability of each node to record the arrival time of the first bit in each incoming cycle.

topology naturally introduces the concept of neighbors. We call node j a *neighbor* of node i, and write $j \to i$, if there is a link from node j to node i. The neighborhood of node i is defined as the set

$$U_i \equiv \{j : j \to i\} \cup \{i\}.$$

Note that we include i itself in the set of its neighbors. We denote the latency of the link $j \to i$ by l_{ji}. Finally, if there is both a $j \to i$ as well as a $i \to$j links, l_{ij} may not necessarily be the same as l_{ji} (i.e. there is no symmetric latency requirement).

3.2 Clock Model

Our analysis requires consideration of clock readings in several contexts: local clock times, global clock time, and relations between clock times at neighboring nodes. We let $s_j^i(k)$ denote the start time of cycle k on node j, as interpreted by the clock on node i. In general, our clock notation follows the following conventions.

- The superscript indicates which clock is recording the given interval or event of interest.
- The subscript indicates the node on which the event occurred.
- The "argument" provides the cycle number. The network begins operations with cycle 0, although we do not require that all nodes start operating at absolute time 0.
- An absence of a superscript indicates an absolute (also called *global*) time reading.

Every cycle, C, consists of a *data transmission period* (or simply *transmission period*), P, followed by an *adjustment gap*, Y. We set the initial value of C to be equal to P (thus Y is 0), or slightly longer if we assume the computations performed by CNS takes a non-zero amount of time. In addition, we define the kth *observation period* to be the interval consisting of adjustment period $k-1$ followed by transmission period k. Since every node can record the arrival time of any bit on its input links, information about the length of a cycle at any neighbor can be collected by observing consecutive start times of the data transmissions received from that neighbor, with discretization errors due to finite clock granularity.

Our clock model assumes a constant drift rate. Specifically, we assume that an interval of length Δt^i as measured by clock i is related to the absolute time Δt according to the relation

$$\Delta t^i = r_i \Delta t, \tag{3.1}$$

where the clock drift rate r_i is fixed for each i. We assume that link latencies are fixed as well. Although in practice, there can be perturbation in the latencies, these are so small that over the length of a single cycle the effect is negligible. Any long term effect caused by the perturbation will be corrected by the algorithm. Similarly, clock drift rate variations will have a second order effect. Again, this variation is small and slow enough so that the algorithm corrects for it. Assuming static latencies and clock drift rates allows us to provide a static analysis of the behavior of the synchronization algorithm.

3.2.1 Clock Drift Rate Ratio

In the absence of a global clock, it is impossible for nodes to determine individual clock drift rates. They can, however, determine clock drift rate ratios. Specifically, an interval of absolute length Δt is measure as $\Delta t^i = r_i \Delta t$ on node i and as $\Delta t^j = r_j \Delta t$ on node j. Thus

$$\Delta t^i r_i = \Delta t^j r_j,$$

or equivalently

$$\Delta t^j = \left(\frac{r_j}{r_i}\right) \Delta t^i. \tag{3.2}$$

3.3 Overview and Intuition

The working principle of the algorithm is straightforward:

Assume there are N nodes in the network. For $i = 1, 2, \ldots, n$ let D_i be a constant with magnitude on the order of the desired cycle length C. During steady state operation of the algorithm, node i records the start times (if any) of all incoming cycles it receives from its neighbors during observation period k. Node i sets the start time for cycle $k+1$ to the average of these neighbor start times (including its own) plus the fixed value D_i.

3.3.1 Averaging Clock Drift Rates

Intuitively, CNS is an *averaging algorithm*, and the values we are taking the average of is basically the clock drift rates, the r values. Let δ_i be a value maintained and updated by each node using the following relation after every cycle:

$$\delta_i(k) = \frac{r^i_{av}(k)}{r_i}$$

where $r^i_{av}(k)$ is the current estimate of the average value of r by node i.

In order to show the way the algorithm works, let us take a look at the actions taken by node i in deciding the value of the next cycle length after its current cycle of length $C^i(k)$. Let us assume that the node i has n incoming connections from nodes $x, y, z, \ldots$. It calculates the new value for the cycle $C^i(k+1)$ as follow:

$$C^i(k+1) = \frac{n}{\left[\frac{1}{C^i(k)} + \frac{1}{C^i_x(k)} + \ldots\right]} \tag{3.3}$$

The scheme starts by each unit setting $C^i = C$.

We observe that the values of the cycle lengths converge to a constant time value, as measured by the global clock. In order to see how these computations work, let us see the first few steps for the two node direct-connected case (nodes i and j).

We start by setting $\delta_i = \delta_j = 1$. Since we normally want $C^i = \frac{C}{\delta_i}$, we initially set $C^i(1) = C^j(1) = C$. Note that $C^i_j = \frac{r_i}{r_j}C^j = \frac{r_i}{r_j}(\frac{C}{\delta_j})$. Thus when we use Equation (3.3) to calculate the new value, we get

$$\begin{aligned} C^i(2) &= \left(\frac{2}{\frac{1}{C^i(1)} + \frac{1}{C^i_j(1)}}\right) \\ &= \left(\frac{2}{\frac{\delta_i(1)}{C} + \frac{\delta_j(1) r_j}{r_i C}}\right) \\ &= \left(\frac{2Cr_i}{r_i + r_j}\right) \end{aligned}$$

Thus $\delta_i(2) = \frac{r_i + r_j}{2r_i}$. The general expression can thus be written as

$$\delta_i(k+1) = \frac{r^i_{av}(k) + r^j_{av}(k) + \ldots}{nr_i}$$

At the k^{th} cycle, the value of δ at any node represents the average of the $r_{av}(k)$ terms for the nodes connected to it on the incoming link, as modified by its own drift rate. In a connected network, the numerator in this expression converges to the true average of the drift rate for all of the nodes. Thus the value of δ for each node converges to a constant adjusted for its own drift rate.

3.3.2 Finding Eigenvalues

Another way of looking at the convergence properties of the CNS scheme is to take a look at the way the average is getting computed above. We can represent the topology of the network as a stochastic matrix, H, in which the (i,j) term is non-zero if there is a link from node i to node j (this includes the (i,i) term). The non-zero value is set to $1/m$ if there are m incoming links to node j (i.e. we normalize H). Let there be n nodes in the network. Then H is an n by n matrix. Let $R(0)$ represents an n vector whose components represent the actual drift rates of the corresponding nodes. We calculate

$$\begin{aligned} R(k+1) &= HR(k) \\ &= H^{k+1}R(0) \end{aligned}$$

Since H is a stochastic matrix, its largest eigenvalue is one. The computation converges to the eigenvector which will have all components equal to the average value of the n drift rates [4]. The rate of convergence in this case is determined by the second largest eigenvalue. If we consider a fully connected network, then the second largest eigenvalue is zero and the computation converges in one step. We note that the convergence in this case is exponential and monotonic.

In the description above, we have presented the convergence property of CNS in terms of the drift value r, and how as the r_{av} values at each node becomes the same, the cycle length will become the same as well. This can also be achieved by making the new cycle length to be the average of the incoming cycle length. We note that this process goes on based strictly on the cycle length measurements made at a node according to its local clock. The scheme still converges to a constant cycle length as measured by the *global clock*.

3.4 CNS Algorithm

In our overview discussion, we had assumed that the delays caused by latency values are negligible, such that cycle start times will arrive in the same cycle they are sent out at. When realistic link latencies are considered, and a start time may take multiple cycles (possibly hundred or thousand) to arrive at its destination, the averaging algorithm would still converge, but this time to a value that is dependent in part on the latency values. The inclusion of the D_i value into the algorithm is specifically designed to counteract this, removing any dependency of cycle length on latencies. Simulations performed by not including the D_i values have shown that the converged cycle length could end up being 40% longer than the initial desired cycle length. D_i effectively helps to keep the padding or waiting time to a minimum. In addition, we had also assumed that the arithmetic operations have infinite precision. If we wish to reduce the complexity of the CNS scheme so that it can be implemented on relatively simple hardware components, this may not be possible [11]. Subsequently, when calculations are performed with finite precision, one then has to be concerned with round-off effects, and specifically in making sure that these values do not accumulate over time. For example, while the cycle lengths may be only a fraction of a clock tick apart, over the period of a few thousand or million cycles, these minor differences could desynchronize the whole system if allowed to accumulate and not properly accounted for. In our formal description of the algorithm presented below, as well as in our simulation results given in a subsequent chapter, both of these assumptions have been removed.

The algorithm consists of two phases, an initialization phase during which the D_i are the same across all nodes, and the primary operation phase during which the D_i values differ from node to node. Operation in both phases is identical except for the change in D_i value—the purpose of the initialization phase is to allow the algorithm to converge to a steady state during which D_i values can be determined. Conversion from initialization to primary operation phase does not require synchronization among the nodes, but can instead take place over the course of several (even thousands of) cycles with some nodes in initialization mode and others in primary operation mode. In practice, nodes can be programmed to switch from initialization to primary operation at a specific (local) cycle.

Formally, the algorithm is as follows. Let $s(k)$ (with appropriate subscripts and superscripts) denote the start time of cycle k, U_i denote the set of neighbors of node i, and $|U_i|$ denotes the cardinality of U_i.

Initialization Phase

1. Each node initially transmits for an interval of P time units, as measured by its local clock.

2. At the end of the kth transmission period, each node sets the start time of transmission period $k + 1$ to the average of all the start times observed (if any, and including its own) during observation period k, plus the fixed value C. That is,

$$s_i^i(k+1) = C + \frac{1}{|U_i|} \sum_{j \in U_i} s_j^i(k) \tag{3.4}$$

The adjustment required to change the start time of the subsequent cycle is "absorbed" by the adjustment period.

3. Once a predetermined cycle instant, called the *alpha cycle or point*, is reached, the nodes start observing (for several cycles, say k in $\mathcal{K}$) the difference,

$$v_i(k) \equiv \frac{1}{|U_i|} \left(\sum_{j \in U_i} s_j^i(k) \right) - s_j^i(k) \tag{3.5}$$

between the average start time of its neighbors (including itself) during a cycle and its own start time during the same cycle. D_i is then defined by

$$D_i = C - \frac{1}{|\mathcal{K}|} \sum_{k \in \mathcal{K}} v_i(k) \tag{3.6}$$

Once D_i has been computed, the node moves into primary operation mode. The value of D_i remains fixed until either the network goes offline or a complete restart is required, in which case the nodes start the process all over again beginning with the initialization phase. In Equation (3.6) above, we require that $\mathcal{K}$ satisfies the criteria

$$\mathcal{K} > \frac{max(l_{ji})\ \forall j \in U_i}{C} \tag{3.7}$$

as this will ensure that the latency will be accounted for when D_i is finally computed.

Primary Operation Phase

1. At the end of the kth transmission period, each node sets the start time of transmission period $k+1$ to the average of all start times observed (if any, and including its own) during observation period k, plus the fixed value D_i. Assuming that start times are received from all neighbors during each cycle, we have

$$s_i^i(k+1) = D_i + \frac{1}{|U_i|} \sum_{j \in U_i} s_j^i(k) \tag{3.8}$$

Figure 3.1 shows a summary of the cycle length computation performed at the different phases. The interval from the start until the alpha point is to allow the nodes to spread the initial or default drift values around the system (using C_i, which is C according to the node's local clock). Once this is done, the next step is to compute the value v_i (which will ultimately be used in the computation of D_i) for $\mathcal{K}$ cycles. The reason we require that $\mathcal{K}$ satisfies Equation (3.7) is to ensure that the v_i (D_i) computation takes into account the latency delays. At the end of the $\mathcal{K}$ interval, we replace C_i with D_i in the next cycle computation, and use the new formula from this point onward.

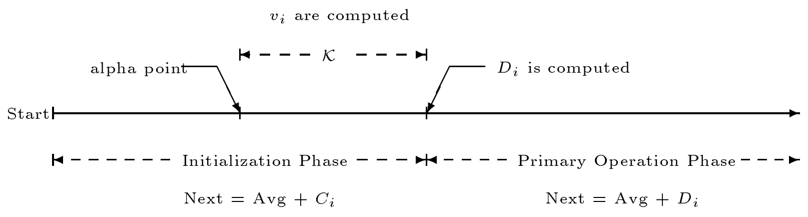

Figure 3.1: Phases of the algorithm.

3.5 Analysis

We assume initially that link latencies are small enough that in steady state operation, start times of the kth transmission period are observed during the kth observation period. In practice, the start of the kth cycle on one node may not be observed on a neighboring node until several hundred cycles later.

We begin our analysis with a look at the core algorithm: setting the start time of a cycle to the average of the previous start time plus a fixed (and possibly node specific) constant. So, for each integer $i \in [1, N]$ let M_i be fixed and assume in addition that the M_i values are relatively close[2]. Consider the following generalization of Equation (3.8).

$$s_i^i(k+1) = M_i + \frac{1}{|U_i|} \sum_{j \in U_i} s_j^i(k). \tag{3.9}$$

This can be written in terms of absolute time as

$$s_i(k+1) = \frac{M_i}{r_i} + \frac{1}{|U_i|} \sum_{j \in U_i} (s_j(k) + l_{ji}). \tag{3.10}$$

This can also be expressed in matrix form. Specifically, let H be the adjacency matrix that captures the network topology:

$$H_{ij} = \begin{cases} 1, & j \in U_i \\ 0, & \text{otherwise} \end{cases}$$

[2]This assumption is not necessary if we allow the stretchable adjustment period length to be as large as necessary to run the algorithm. That is, we need the adjustment period to be long enough to allow a node to set the next start time to the value dictated by the algorithm.

Using H, Equation (3.10) can rewritten as

$$s_i(k+1) = \frac{M_i}{r_i} + \frac{1}{\sum_j H_{ij}} \sum_j H_{ij}(s_j(k) + l_{ji}). \tag{3.11}$$

If we define the stochastic adjacency matrix G by

$$G_{ij} = \frac{H_{ij}}{\sum_j H_{ij}} = \frac{H_{ij}}{\text{degree of node } i}, \tag{3.12}$$

where the degree of node i includes the count of the self-loop, then Equation (3.11) becomes

$$s_i(k+1) = \frac{M_i}{r_i} + \sum_{j=1}^{N} G_{ij}(s_j(k) + l_{ji}). \tag{3.13}$$

where N denotes the number of nodes in the network. Let $S(k)$ denote the $N \times 1$ column matrix whose entries are the $s_i(k)$, M denote the $N \times 1$ column matrix whose entries are the M/r_i, and L denote the $N \times M$ matrix whose ij entry is l_{ij}. Finally, define diag(A) for an $N \times N$ matrix A to be the $N \times 1$ matrix whose ith entry is A_{ii}. Then Equation (3.13) can be rewritten as

$$S(k+1) = GS(k) + diag(GL) + M \tag{3.14}$$

A closed form solution for this Equation is

$$S(k) = G^k S(0) + \sum_{i=0}^{k-1} G^i (M + diag(GL)) \tag{3.15}$$

where k is any non-negative integer. Cycle lengths can be determined by looking at the differences of successive start times.

$$S(k+1) - S(k) = (G^{k+1} - G^k)S(0) + G^k(M + diag(GL)). \tag{3.16}$$

Now we show in Section 3.6 that under our network topology assumptions, the powers of G converge to a stochastic matrix Q, all of whose rows are identical. Thus,

$$\lim_{k \to \infty} (S(k+1) - S(k)) = Q(M + diag(GL)). \tag{3.17}$$

Since the limit is a column vector in which all entries are the same, cycle lengths converge to the same absolute length at each node. In addition, and perhaps more important, once the algorithm has reached steady state, start times remain locked relative to the start times of neighbor nodes. That is, there is no phase shift.

In the initiation phase of the algorithm, the value of each M_i is C, and in the primary phase $M_i = D_i$, so the above analysis shows that the algorithm converges in both phases, and that the rate of convergence is determined by the rate of convergence of the powers of G. In addition, the limiting cycle length depends in part on $\text{diag}(GL)$, which is a term that captures the effects of link latencies. Specifically, $(\text{diag}(GL)_i) = (GL)_{ii}$. Since the jth entry in the ith row of G is nonzero if and only if j is a neighbor of i, and ith row of G effectively lists the neighbors of i. The ith column of L on the other hand, lists the latencies from neighbors of i into i. Thus $(GL)_{ii}$ (and $(\text{diag}(GL)_i)$) is the average of link latencies on links toward i.

As mentioned earlier, our definition of D_i is designed to counteract this dependence on link latencies. To simplify the analysis in this case, assume that the set $\mathcal{K}$ in Equation (3.6) consists of the single value 0. Then D_i is defined by

$$D_i = C + s_i^i(0) - \frac{1}{|U_i|}\sum_{j\in U_i} s_j^i(0).$$

Substituting this into Equation (3.8) gives

$$s_i^i(k+1) = C + s_i^i(0) - \frac{1}{|U_i|}\sum_{j\in U_i} s_j^i(0) + \frac{1}{|U_i|}\sum_{j\in U_i} s_j^i(k).$$

In absolute terms, this becomes

$$s_i^i(k+1) = \frac{C}{r_i} + s_i(0) - \frac{1}{|U_i|}\sum_{j\in U_i}(s_j(0) + l_{ji}) + \frac{1}{|U_i|}\sum_{j\in U_i}(s_j(k) + l_{ji})$$

Moving to matrix notation, the analogue of Equation (3.14) is

$$\begin{aligned} S(k+1) &= C - G\,S(0) - diag(GL) + G\,S(k) + diag(GL) + S(0) \\ &= C + (I-G)S(0) + G\,S(k). \end{aligned}$$

The corresponding closed form solution is

$$S(k) = S(0) + \sum_{i=0}^{k-1} G^i C,$$

so

$$S(k+1) - S(k) = G^k C \to QC.$$

The limit in this case is a weighted sum of the C/r_i and is independent of the values of the link latencies.

If clock drift rates are required to satisfy $|1 - r_i| < \delta$ for some fixed δ, then the converged or limiting cycle length, C_L, satisfies

$$\frac{C}{1+\delta} < C_L < \frac{C}{1-\delta}. \tag{3.18}$$

A realistic value for δ is 0.0001, which corresponds to clocks that are accurate to 100 parts per million. For 125 μs cycle lengths, this guarantees a limiting cycle length between 124.9875 μs and 125.0126 μs, or less than one hundredth of a percent deviation from the desired length.

It is natural at this point to question the need for the initialization phase, since the previous analysis set D_i values according to the observed start time for the first cycle. In practice, start times for neighbor nodes may not be observable for several cycles. For example, with 125 μs cycles, a 10 ms delay corresponds to 80 cycles.

3.6 Convergence Behavior

Given that the rate at which the algorithm reaches steady state depends on the convergence properties of G^k, we need to determine the conditions under which G^k converges, the limit when it does converge (and the sense in which we mean "limit"), and the rate of convergence.

We begin with the issue of the conditions under which G^k converges. G is a finite stochastic matrix, and thus it must be the transition matrix for a finite Markov chain. Specifically, let $\mathcal{G}$ be the graph that represents the topology of our Cyclone network (including self-loops), and consider the Markov process corresponding to a "traveler" moving randomly (along edges of G) from node to node on $\mathcal{G}$, with the system in state S_i at a particular time epoch if the traveler is located at node i during that epoch. G is the transition matrix for this finite Markov chain. Because $\mathcal{G}$ is connected and contains self-loops, the Markov chain is ergodic (irreducibility follows from being connected, aperiodicity from the self-loops). Among other properties, ergodicity guarantees that the powers G^k approach a matrix Q, in the sense that each entry of G^k approaches the corresponding entry of Q [33, 23]. Moreover, each row of Q is the same positive probability vector W, where W is the unique probability vector such that $WG = W$. Equivalently, if $W = [w_1\ w_2\ \ldots\ w_N]$, then the w_i are uniquely determined through the system of equations

$$\sum_{i=1}^{N} w_i = 1; \qquad w_j = \sum_{i=1}^{N} w_i G_{ij}, \qquad j = 1, 2, \ldots, N.$$

Since w_i represents the "long term" probability that the system is in state i, it is fairly intuitive that each w_i should be given by

$$w_i = \frac{\text{degree of node } i}{\text{number of nonzero entries of } G}$$

where the degree of a node includes a count of the self-loop. A straightforward calculation verifies this result. To simplify notation, we will refer to the number of nonzero entries of G as the *degree* of G, denoted degree(G), and denote the degree of node i by degree(i), so that

$$w_i = \frac{\text{degree}(i)}{\text{degree}(G)}.$$

Given this value of the limiting matrix Q, the limiting cycle length L is given by

$$L = \sum_{i=1}^{N} w_i C_i(0) = \sum_{i=1}^{N} \frac{\text{degree}(i)}{\text{degree}(G)} C_i(0) \tag{3.19}$$

Since the initial cycle for node i has length (in absolute time) of C/r_i, we have

$$L = \sum_{i=1}^{N} \frac{\text{degree}(i)}{\text{degree}(G)} \frac{C}{r_i} \tag{3.20}$$

Thus L is, as stated earlier, a weighted average of initial cycle lengths.

Determining the rate of convergence is relatively straightforward in theory: the powers of the transition matrix of an ergodic Markov chain converge at a rate related to the moduli of the eigenvalues of the matrix. This can be observed by considering the *spectral representation* of G. Specifically, the (positive integral) powers of a diagonalizable stochastic matrix G are given by

$$G^k = Q + \lambda_1^k A_1 + \lambda_2^k A_2 + \cdots + \lambda_m^k A_m \tag{3.21}$$

where the λ_i are the non-one eigenvalues of G, Q is the limit of the powers of G, and the A_i are differential matrices (i.e. each row of the matrix sums to zero) satisfying the following:

1) $A_i A_j = A_j A_i = 0$ if $i \neq j$

2) $A_i^k = A_i, 1 \leq i \leq m, k = 1, 2, \ldots$.

3) $A_i Q = Q A_i = 0, 1 \leq i \leq N$.

4) $\|A_i\| \leq 1, i = 1, 2, \ldots, m$.

Viewed in this form, it is clear that G^k converges at the same rate as the largest of the moduli of the λ_i. We refer to an eigenvalue with the largest moduli as a "submaximal" eigenvalue. That is, an eigenvalue is submaximal if its absolute value is equal to the maximum of the moduli of the set of non-one eigenvalues (note that 1 is an eigenvalue of any stochastic matrix, and that Q would be the corresponding matrix in the spectral representation). Since G^k converges, it is clear that the moduli of the λ_i must be less than one. This also follows from one form of the Perron-Frobenius theorem which also asserts that the eigenvalue one has multiplicity one [18].

The matrix G corresponding to the networks under consideration here is diagonalizable. To see this, consider the non-normalized adjacency matrix H corresponding to G. Because all links in the underlying network are bidirectional, H is symmetric, and thus diagonalizable (and all of its eigenvalues are real). Normalizing (as defined by Equation (3.12)) amounts to multiplying H on the left by a diagonal matrix D with strictly positive diagonal entries. Such a D must be invertible and have an invertible square root. Thus $G = DH$ is similar to $D^{-\frac{1}{2}}(DH)D^{\frac{1}{2}} = D^{\frac{1}{2}}HD^{\frac{1}{2}}$. Since this last matrix is symmetric, G is similar to a symmetric matrix and thus it is diagonalizable and has only real eigenvalues.

Determining the eigenvalues of a matrix can be difficult. There are a few classes of topologies for which an explicit closed form solution for the eigenvalues can be found. One of these is a complete graph, in which the eigenvalues are easily seen to be 1 with multiplicity one (of course) and 0 with multiplicity $N-1$. The star topology is another whose eigenvalues are relatively easy to determine. We call a graph an N-star if the graph contains a total of N nodes: one "hub" node and $N-1$ leaf nodes. It can be shown that the non-one eigenvalues of the N-star are $\frac{1}{2}$ with multiplicity $N-2$ and $\frac{1}{N}-\frac{1}{2}$ with multiplicity one. Since $|\frac{1}{N}-\frac{1}{2}|<\frac{1}{2}$ for $N>1$, the submaximal eigenvalue is $\frac{1}{2}$, and G^k converges at the rate of $\mathcal{O}(\frac{1}{2^k})$.

Although we are more concerned with upper bounds on convergence rates of G^k, it is of interest to observe a situation in which a lower bound on the convergence rate can be computed. Specifically, if each node in $\mathcal{G}$ has degree m, then each nonzero entry of G is $\frac{1}{m+1}$, and trace$(G)=\frac{N}{m+1}$. Now, the trace of a matrix is equal to the sum of its eigenvalues, so since 1 is an eigenvalue of multiplicity one, the sum of the remaining eigenvalues is $\frac{N}{m+1}-1=\frac{N-m-1}{m+1}$. Let λ be a submaximal eigenvalue of G. In order for the sum of the non-one eigenvalues to equal the expression above, we must have

$$(N-1)|\lambda| \geq \frac{N-m-1}{m+1}$$

or equivalently

$$|\lambda| \geq \frac{N-m-1}{(m+1)(N-1)}.$$

Thus, the *fastest* that G^k can converge in this case is at the rate of

$$\left(\frac{N-m-1}{(m+1)(N-1)}\right)^k.$$

Finally, if we allow graphs with unidirectional links, then in terms of convergence, a worst case is a "one way" ring, for which the modulus of the base p in the exponential convergence rate p^k can be made arbitrarily close to 1. Although this shows that convergence rates can in theory be relatively slow, this is not a practical limitation. As our simulation results show (for a bidirectional ring—a situation for which computing closed form expressions for eigenvalues can be daunting), convergence rates will generally be slower than more favorable topologies, but still well within tolerable limits.

3.7 Summary

In this chapter, we formally describe the algorithm used by the Cyclone Network Synchronization scheme, and provide the intuition behind its working principle. An analysis on the converged cycle length shows that this length is dependent only on the clock drift rates, and not on the latency values. In addition, simulation results present in a subsequent chapter of this dissertation will also show that the algorithm converges in a very short amount of time for a variety of network settings.

Chapter 4

Enhancements

In this chapter, we discuss two enhancements to the basic algorithm that help facilitate the implementation of CNS on actual Cyclonode hardware in a Cyclone network. The first deals with minimizing the number of buffers needed at each Cyclonode. The second allows (new) nodes to join an existing Cyclone network or (current) nodes to leave an existing Cyclone network.

4.1 Minimize Buffering

In Equation (3.4), the value of k on both sides of the equation are only the same if the latency is less than one cycle length (at that given cycle). Otherwise, the k on the right-hand side is likely to be a few cycles behind the k on the left-hand side. For example, assuming the latency is equal to 3 cycle lengths, the start time of cycle N at the source node will not be visible at the destination node until cycle $N + 3$. In any case, a node simply "processes" the incoming start times in subsequent arrival order. If $s_i(k)$ makes use of $s_j(l)$ in its computation, then $s_i(k + 1)$ will make use of $s_j(l + 1)$, and so on.

Because of the different clock drift rates and the initialization phase (where cycle length at the nodes can vary), the FIFO processing of incoming start times

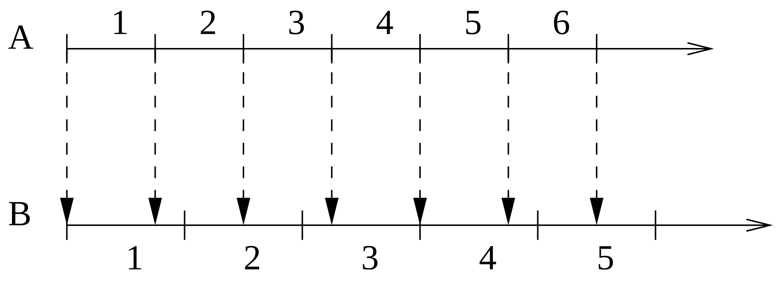

Figure 4.1: Buffering due to different clock drift rates.

can lead to cases where a node, when computing $s_i(k)$, will have to depend on s_j values that had arrived at s_i a few cycle earlier, e.g. at $k - N$ for some values of N (see Figure 4.1). This node will therefore have to allocate buffers to store these incoming values until they are processed. Since we do not know at this time which parameters affect or determine the number of buffers, we need to modify the basic algorithm so that this number is bounded, preferably by as small a value as possible (e.g. 1 or 2). This is the motivation for the "minimize buffering" modification.

Let C be the desired cycle length as described previously. When computing $s_i(k+1)$, we first look at the interval between $s_i(k) - C$ and $s_i(k) + C$, i.e. two "windows" of length C centered at $s_i(k)$ (see Figure 4.2). Let $s_j(l)$ be the next unprocessed incoming start time. If $s_j(l)$ is within some ϵ of either boundary, then we say that $s_j(l)$ has (potentially) "drifted" too much to the right or to the left, or faster or slower with respect to $s_i(k)$. In this case, instead of using $s_j(l)$ as in the original algorithm, we use either $s_j(l-1)$ if drifting to the left, or $s_j(l+1)$ if drifting to the right. At the next cycle, we will then use $s_j(l)$ or $s_j(l+2)$, respectively.

Even though we are no longer using $s_j(l)$, we cannot simply use the previous/next incoming start time value directly when computing the next start time

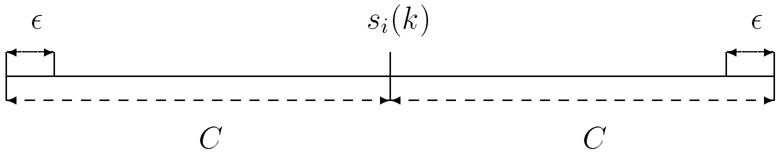

Figure 4.2: Two-windows checking scheme to minimize buffering.

since that would skew the average either too far to the past (since we have already used this value) or the future (since we are not supposed to use this value until the next cycle). To get around this problem, we add an "offset" value to each affected link, and either add to or remove from the averaging computation for the next and all subsequent cycles, such that they are unaffected by our jumping backward or ahead.

Formally, we modify the basic algorithm in the following way. We define an offset value for each link, $o_j (j \in U_i)$, which is initially set to 0. Equation (3.4) is then modified to take account the offset values:

$$s_i^i(k+1) = C + \frac{1}{|U_i|} \sum_{j \in U_i} (s_j^i(l) + o_j) \tag{4.1}$$

Let $s_j^i(m)$ be the earliest s_j^i that has arrived, but not yet used in the next cycle computation by node i. Let W be a window of size 2C centered $s_i(k)$. Let E be a small interval of size ϵ, say 1/10 the size of C, at both ends of W. If $s_j^i(m)$ does not fall into either E, i.e.

$$s_i(k) - C + \epsilon < s_j^i(m) < s_i(k) + C - \epsilon \tag{4.2}$$

then we let $s_j^i(l)$ be $s_j^i(m)$. Otherwise, if $s_j^i(m)$ falls into the left E window (i.e. $s_j^i(m) < s_i(k) - C + \epsilon$), then we set $s_j^i(l)$ to be $s_j^i(m+1)$, and update o_j as follow:

$$o_j = o_j - (s_j^i(m+1) - s_j^i(m)) \tag{4.3}$$

Similarly, if $s_j^i(m)$ falls into the right E window (i.e. $s_j^i(m) > s_i(k) + C - \epsilon$), then we set $s_j^i(l)$ to be $s_j^i(m-1)$, and update o_j as follow:

$$o_j = o_j + (s_j^i(m) - s_j^i(m-1)) \tag{4.4}$$

Finally, we modify Equation (3.8) as well to take into account the offset values:

$$s_i^i(k+1) = D_i + \frac{1}{|U_i|} \sum_{j \in U_i} (s_j^i(l) + o_j) \tag{4.5}$$

$s_j^i(l)$ and o_j are computed as described above.

4.2 Adding and Removing Nodes

We now look at how the basic CNS algorithm can be modified to accommodate topology changes, with new nodes joining or existing nodes leaving a Cyclone network. We assume that:

- The Cyclone network has reached "steady-state", where the cycle length are the same everywhere, as measured by the *global clock*.
- The Cyclone network is still "connected" after the addition or removal of a node.
- If a new node is being added, its data transmission period is less than the steady-state cycle length (both values as measured by its local clock).

- If a new node is being added, it can monitor the network for a period of time to determine the steady-state cycle length before beginning actual operation, i.e. sending actual data.

Let us first consider the addition of a new node, say node i. Based on our assumption, i will be able to determine the steady-state cycle length by listening on its coming links. If necessary, it can average this value over a number of cycles since the steady-state cycle lengths can still fluctuate a little due to factors such as clock drift and latency perturbations and computation round-offs. Once i is ready to join the network, it simply sets its cycle length to the average cycle length it observes on its incoming link,

$$s_i^i(k+1) = s_i^i(k) + \frac{1}{|U_i|}\sum_{j\in U_i}(s_j^i(k) - s_j^i(k-1)) \tag{4.6}$$

Let j be an existing node in the Cyclone network with an incoming link from i. Node j now has an extra incoming start time at each cycle. On one hand, it cannot simply use the incoming start time from i as is since this new addition will likely cause the average (start time) value in Equation (3.8) to change, and therefore changing the (steady-state) cycle length and subsequently throwing the whole network out of sync. On the other hand, j should not ignore i incoming start time completely because it should at least take into account any minor fluctuations in i's steady-state cycle length, in order to propagate this throughout the network so all nodes can make the proper adjustments. To satisfy both of these requirements, we make use of a "node-specific offset value" in a manner similar to the way link-specific offset values are used as described above in the *Minimize Buffering* section. When j detects i sending data for the first time, it factors the initial start time from i, say s_i, into its node-specific offset. Subsequent cycle length computations at j will then use this offset value to factor out only s_i, but still consider any fluctuations i may have caused.

Formally, we modify the algorithm again in the following way. We define an offset value for each node, p, which is initially set to 0. Equation (4.5) is then modified to take account the node-specific offset value:

$$s_i^i(k+1) = D_i + p_i + \frac{1}{|U_i|}\sum_{j\in U_i}(s_j^i(k) + o_j) \tag{4.7}$$

At any given cycle, when a node detects that one or more new links are becoming "active" for the first time, it updates its p offset value by computing the average of the incoming start times both with as well as without the start times from the new links. Let U_i be the set of neighbors including the new links and let V_i be the set without the new links (i.e. the previous U_i),

$$p_i = p_i + (\frac{1}{|V_i|}\sum_{j\in V_i}(s_j^i(k) + o_j)) - (\frac{1}{|U_i|}\sum_{j\in U_i}(s_j^i(k) + o_j)) \tag{4.8}$$

We handle the case when an existing node leaves the network in a similar manner. We factor the effect this node would have in the averaging value, such that subsequent cycle length computations will be carried out as if the node is still there. This is necessary so that the steady-state cycle length does not change. Let U_i be the set of neighbors without the removed links and let V_i be the set with the removed links still left in (i.e. the previous U_i), p_i is computed exactly as in Equation (4.8), as we simply swap the role of U_i and V_i when deleting a node.

Chapter 5

Simulation Results

In this chapter, we provide simulation results, using realistic values for parameters, to validate the analysis given in the previous chapter. Specifically, the results show that the converged cycle lengths conformed closely to the results of Equation (3.18). They also show how quickly, in terms of the *global clock* time, the CNS algorithm converges for various network configurations. And finally, results showing the behavior of the scheme in the presence of both latency as well as clock drift perturbations are presented.

5.1 Setup

We assume that the network operates at 10 GHz, with a 125 μs desired cycle length (or about 8000 cycles per second). We also assume that the granularity of the timestamp clock is accurate to a within a single clock tick, or 100 picoseconds. Rate of convergence is measured in terms of the number of cycles the network takes to reach steady-state (usually in the thousand of cycles when denotes by K). Converged cycle lengths (C_L), cycle length jitters, as well as start time offset jitters, are all measured in terms of the number of clock ticks. We represent the clock drift at each node by specifying the r value, the drift rate, and use Equation

(3.1) to convert a local time to the global time, or Equation (3.2) to convert the between the local times at two nodes. All arithmetic operations are carried out with finite precision.

Unless otherwise noted, our "baseline" simulation dataset will consist of the Cyclone network with the following parameters:

- The nominal cycle length of 1.25 million clock ticks, or 125 μs (this is the C value).
- 20 nodes organized in a "chain" topology (see Figure 5.1).
- Unidirectional links, with latencies being random values between 0 and 100 million clock ticks (or about 80 cycles).
- Clock drift rates between 0.9991 and 1.0009, or equivalent to clocks that are accurate to about 1000 PPM.
- The *alpha point* in the initialization phase set at the 2000^{th} cycle.
- $\mathcal{K}$ in Equation (3.7) set to 1000.

While we have chosen the above values for our default dataset, we note that the synchronization scheme itself is not dependent on any specific values, and we will in fact vary all of the parameters in our simulations.

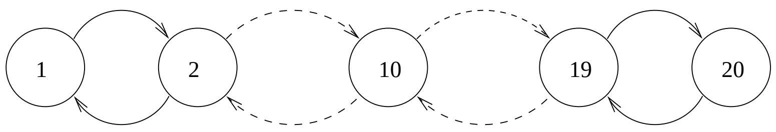

Figure 5.1: 20-node Chain Network Topology

The goal of the CNS algorithm is to enable the nodes in the network to reach "convergence", or "steady-state", whereby the limiting or converged cycle length satisfies Equation (3.18). Note that the converged cycle length, C_L, may fluctuate a little bit due to factors such as finite precision calculation roundings.

5.2 Convergence Criteria

In our simulations, we use the following criteria to determine if and when a network has reached convergence, or steady-state. Let $TOTAL$ be the number of cycles (per node) the algorithm will execute. To be more precise, since the nodes can have different clock drifts, not all of them will execute for exactly $TOTAL$ cycles. Instead, the first node that reaches this limit will terminate the simulation. Let the cycle denoting the end of the simulation be referred to as $Cycle_{TOTAL}$. As the simulation progresses, we keep track of two sets of statistic along the way. However, since we are mainly interested in the steady-state behavior, and not with the behavior of the network at the beginning while adjustments are being made, we collect these statistics only from the point where we believe convergence has been reached. Let $Cycle_{CONV}$ be the cycle at this point, and let $CONV$ be the number of cycles since the start of the simulation up until then. Statistics are therefore only kept from $Cycle_{CONV}$ until $Cycle_{TOTAL}$.

The first statistic we keep track of is the cycle length (as measured by the *global clock*) at each node during the simulation. This value changes for various reasons as the simulation progresses. We refer to this cycle length difference as the **cycle length jitter (CLJ)**. In addition, for the purpose of aiding in the calendar scheduling at a Cyclonode, we are also interested in keeping track of when an incoming cycle will arrive, relative to the start of the local cycle, on all the incoming links. If there is no fluctuation in the cycle lengths at all of the nodes once convergence is reached, then an incoming cycle will always arrive at exactly the same point relative to the start time at the local node. However, converged cycle lengths do fluctuate as mentioned above. Subsequently, the *time offset* between the start of an incoming cycle and the start of the corresponding cycle at the local node will fluctuate as well. We refer to this fluctuation as the **start time offset jitter (SOJ)**. Our goal is to make sure that both cycle length jitter as well as start time offset jitter are bounded.

Let $maxCycleLen_i$ ($minCycleLen_i$) be the maximum (minimum) cycle length seen at node i during $Interval_{CONV}$, the interval from $Cycle_{CONV}$ to $Cycle_{TOTAL}$. For an incoming link l to a node, let $maxStartOffset_l$ ($minStartOffset_l$) be the maximum (minimum) start time offset (between the start of the local cycle and the incoming start time on that given link) observed at this node during $Interval_{CONV}$.

We say that the network has reached convergence if the following two criteria hold true during $Interval_{CONV}$

$$\begin{aligned} maxCycleLen_i - minCycleLen_i &< \epsilon_1 \\ maxStartOffset_l - minStartOffset_l &< \epsilon_2 \end{aligned} \tag{5.1}$$

for some ϵ_1 and ϵ_2. We specify **both ϵ values to be 10 units (or clock ticks)** in our simulations. In other words, once convergence has been reached, the cycle length at each node should be "the same" (according to the *global clock*), subjected to some bounded fluctuations or jitters. Similarly, incoming cycles to a node should always arrive at the same time, relative to the start time of the corresponding local (outgoing) cycle. We then say that the network converges at $Cycle_{CONV}$, or that it takes $CONV$ cycles to converge.

We vary $Cycle_{CONV}$ accordingly to determine how fast the simulated network converges.

5.3 Baseline Dataset Simulation Output

MCL = minimum cycle length / CLJ = cycle length jitter

MSO = minimum start time offset / SOJ = start time offset jitter

Node	MCL	CLJ	MSO1	SOJ1	MSO2	SOJ2
1	1251125	1	-15014	1		
2	1251126	0	15002	1	-85232	2
3	1251125	1	85393	2	-1004033	3
4	1251125	1	1003129	2	81907	3
5	1251125	1	-81859	2	70431	2
6	1251125	1	-70439	2	13769	2
7	1251125	1	-13783	2	107699	2
8	1251125	1	-107614	2	55269	2
9	1251125	1	-55281	2	47212	1
10	1251126	1	-47217	2	39316	1

Table 5.1: Baseline dataset simulation output

Figure 5.1 shows the partial output of an actual simulation run on the baseline dataset (the full results can be found in Table A.1 in Appendix A). These numbers represent the values that were kept during the statistic gathering period (i.e. from $Cycle_{CONV}$ until the end of the simulation). For each node, we have the *minimum cycle length (MCL)* and its difference from the *maximum cycle length*, which is the *cycle length jitter (CLJ)*. Similarly, for each incoming link to a node, we have the *minimum start time offset (MSO)* and its difference from the *maximum start time offset*, which is the *start time offset jitter (SOJ)* (node 1 has a single incoming link from node 2). Of particular interest is the fact that both cycle length jitters as well as start time offset jitters are kept to only several clock ticks, even though our definition of convergence allows to values up to 10. In addition, the *Minimize Buffering* modification described in Section 4.1 ensures that *start time offsets* are never greater than the length of one cycle. Without this modification, simulations have shown that the start time offset values can be very large (up to 20-cycle length for some of the simulated topologies), requiring a significant number of buffers.

5.4 Cycle Length

We begin by simulating with cycle lengths of 125,000 and 12,500,000 clock ticks, respectively. Partial results are shown in Table 5.2 (the full results can be found in Tables A.2 and A.3 in Appendix A). We can see that in both cases, the converged cycle length (C_L) follows from Equation (3.18). In addition, the jitter values (both cycle length and start time offset) are of the same order of magnitude as in the case of the unmodified baseline dataset, where C is 1,250,000 clock ticks (Table 5.1). Moreover, all 3 datasets converge at the same rate of about 5 secs (40K cycles). These results confirm the fact that the cycle length value does not affect either the convergence rate or the jitter values (which represent the synchronization accuracy of the CNS scheme).

MCL = minimum cycle length / CLJ = cycle length jitter						
MSO = minimum start time offset / SOJ = start time offset jitter						
Node	MCL	CLJ	MSO1	SOJ1	MSO2	SOJ2
	C = 125,000					
1	125112	1	-1504	1		
2	125112	0	1502	1	-8497	1
3	125111	1	8513	1	-100379	3
4	125112	1	100287	2	8212	2
5	125111	2	-8208	2	7064	2
	C = 12,500,000					
1	12511260	1	-150138	1		
2	12511260	0	150017	1	-852606	2
3	12511259	1	854226	2	-10040563	2
4	12511259	1	10031534	3	818855	2
5	12511259	1	-818365	2	704078	2

Table 5.2: Simulation outputs for different cycle lengths

5.5 Network Topology

We now look at how the different network layouts or topologies affect the convergence rate. We arrange N nodes (e.g. 20, 50, 100) in a *chain* (see Figure 5.1), *bidirectional cycle* (see Figure 5.2), *star* (see Figure 5.3), and *random* layouts. The remaining parameters (e.g. clock drift rates, latencies, etc.) are the same as the *baseline dataset* mentioned above.

Here, we would expect the star network to converge the fastest since it has the smallest *diameter* (value of 2) of all the networks, allowing information such as

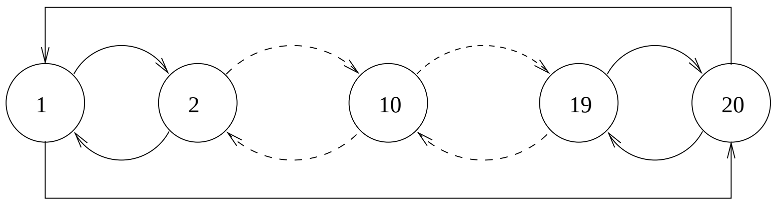

Figure 5.2: 20-node Bidirectional Cycle Topology

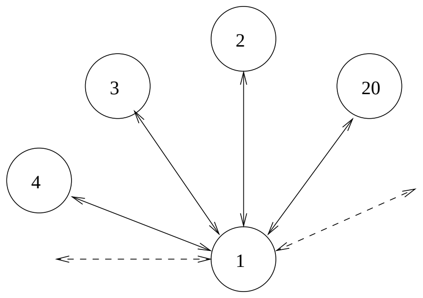

Figure 5.3: 20-node Star Network Topology

cycle lengths to propagate around the network in fewer cycles. The chain network should converge the slowest since it has the largest "longest path" between nodes 1 and 20 (value of 19). The bidirectional networks is simply the chain with a connection between nodes 1 and 20, and therefore should converge at the same rate or slightly faster than the chain.

We generate the *random* network using the pseudo-code shown in Figure 5.4. Since the resulting network is connected and has the minimum number of (bidirectional) edges, it is "minimally connected". A real-life network with the same number of nodes would probably be better connected (i.e. have more edges) than our random network, and thus should converge even faster. In addition to the 20-node, 50-node, and 100-node networks, we also simulate a 200-nodes, 500-nodes, and 1000-nodes networks (all random) to obtain an idea of how long such a large network would take to converge.

```
put the N nodes into the set **UNCONNECTED**
initialize the set **CONNECTED** to $\emptyset$
remove 2 random nodes, **X** and **Y**, from **UNCONNECTED**
connect **X** and **Y** bidirectionally
add **X** and **Y** to **CONNECTED**
while N is not empty
do
    pick a random node, **A**, in **CONNECTED**
    remove any node, B, from **UNCONNECTED**
    connect **A** and **B** bidirectionally
    put **B** into **CONNECTED**
end
```

Figure 5.4: Pseudo-code for random network generator.

Table 5.3 shows the convergence rate for the various networks. The values in the table indicates the time (in seconds) as well as the number of cycles needed to reach convergence. As expected, the star networks converge the fastest, followed by the random network, with the chain and bidirectional being the slowest. With the exception of the 20-node networks, the larger networks seem to converge at a rate that is independent of the total number of nodes (possibly because their diameters are similar). We note that even in the worse case, convergence is reached in about 30 secs. In addition, if we allow the ϵ value that is used to bound the jitters (both cycle length and start time offset) to be larger than 10 clock ticks (as is the case for all of the simulations above), the various networks could potentially converge even faster. Finally, as a way to confirm the long-term stability of the convergence behavior of CNS, several of the simulations were allowed to run for a period of about 108 hours of simulation-time, which corresponds to a $TOTAL$ value of about 3.1 billion cycles.

Convergence rates are in secs (# of cycles)				
# Nodes	**Star**	**Chain**	**Bidirectional**	**Random**
20	2 (15K)	5 (40K)	5 (40K)	4 (32K)
50	15 (120K)	62 (500K)	62 (500K)	31 (250K)
100	15 (120K)	62 (500K)	62 (500K)	31 (250K)
200	N/A	N/A	N/A	31 (250K)
500	N/A	N/A	N/A	31 (250K)
1000	N/A	N/A	N/A	31 (250K)

Table 5.3: Convergence rates for different network topologies.

5.6 Alpha Point

In this simulation, we vary the "alpha point" (this controls how far we allow the default drift rate values to initially propagate around the network) and see if its value affects the convergence rate. We use the 50-node star network from the Network Topology simulations, and specify various alpha points between 2K and 75K. Since the star network converges in about 120K cycles, it would not make sense for us to simulate with even larger alpha values since the network would need some non-zero number of cycles after the alpha point to stabilize. Table 5.4 shows the result of the simulations.

Alpha values are in # of cycles						
Alpha Value	2K	10K	20K	30K	50K	75K
Convergence	Yes	Yes	Yes	Yes	No	No

Table 5.4: Convergence results for 50-star network with different "alpha" values.

Convergence was not possible for alpha values of 50K and 75K since we did not change the $Cycle_{CONV}$ value (set at 120K cycles), and subsequently, the network did not have enough time to stabilize once the alpha point has been reached. Taking into account this fact, the results show that, for a given set of network parameters, the alpha point does not have an impact on the convergence rate. Thus the smallest possible alpha point should be selected, in order to allow the network to more quickly reach convergence.

5.7 Network Latency

Latency Value	**Convergence Rate (# of cycles)**
2 ms	18 secs (150K)
5 ms	18 secs (150K)
10 ms	18 secs (150K)
20 ms	31 secs (250K)
50 ms	50 secs (400K)
100 ms	50 secs (400K)
200 ms	87 secs (700K)
500 ms	112 secs (900K)
1 sec	225 secs (1800K)

Table 5.5: Convergence rate for different latency values.

In our baseline dataset, the latency is a randomly generated value between 0 and a maximum of 100 million clock ticks (or roughly 80 cycles). We now vary the maximum value from 20 million clock ticks (about 16 cycles, or 2 ms) all the way to 10 billion clock ticks (about 8000 cycles, or 1 sec). The network topology is a 20-node random network, with a diameter of 10.

Table 5.5 shows the results of the simulations. As expected, the network does take longer to converge for higher latency values, since the changes (or computations) at a node will take longer to propagate through the whole network. Even so, in the worst case, where the latency can be as high as 1 second on each link, the algorithm still converges in about 1.8 million cycles, or 225 seconds.

Finally, although not obvious from the above results, there is one value that is directly affected by the latency and should be adjusted accordingly. This is the $\mathcal{K}$ value in Equation (3.7), which corresponds to how many cycles each node should compute the D_i value once the "alpha point" has been reached. Intuitively, a node should keep on computing the D_i values until it has received the information from all of its neighbors. This delay is determined by the latencies on its incoming links. Thus $\mathcal{K}$ should be greater than the maximum incoming link latency, in term of the number of cycles. Since different nodes can have different incoming latencies, $\mathcal{K}$ can theoretically be different for each node. However, in practice, we use the same $\mathcal{K}$ value for all node. In the above simulations, $\mathcal{K}$ is set to 100,000 (cycles). Note that computing D_i longer than necessary does not affect the convergence result. It would simply result in unnecessary computation, and therefore $\mathcal{K}$ should be set to the lowest possible value in an actual implementation of the algorithm.

5.8 Clock Drift Rate

As shown in Equation (3.18), the limiting cycle length (or the converged cycle length) is a function of the clock drift values. We simulate different clock accuracy

Cycle lengths are in # of clock ticks					
δ	**PPM**	$C/(1+\delta)$	$C/(1-\delta)$	**Actual** (C_L)	**Deviation**
.01	10000	1237623	1262626	1261351	1%
.001	1000	1248751	1251251	1251124	.1%
.0001	100	1249875	1250125	1250111	.01%
.00001	10	1249987	1250012	1250009	.001%

Table 5.6: Converged cycle length for different clock drift values.

ranging from 10 PPM to 10,000 PPM. Table 5.6 shows the results of these simulations. The δ value, the bound on the clock drift rates, is taken from Equation (3.18), *PPM* is the accuracy of an equivalent clock, *Actual* is the actual limiting cycle length (C_L) obtained from the simulations, and *Deviation* is how much this cycle length deviates from the desired cycle length, or C. The results show that, assuming the hardware timestamp clock has a fine enough granularity, the amount of "padding" added by CNS to the cycle length to ensure that all nodes are synchronized is extremely small, even when using commodity clocks with accuracy with 1000 PPM or worse. Consequently, if we consider a system where the clocks are perfectly synchronized to have zero padding amount, then we can see that CNS adds very little overhead to a similar system with unsynchronized clocks. Finally, we note that all of the simulations converge at the same rate of about 5 secs (40K cycles), and therefore conclude that clock drift rates do not have an effect on the rate of convergence.

5.9 Topology Changes

MCL = minimum cycle length / CLJ = cycle length jitter

MSO = minimum start time offset / SOJ = start time offset jitter

Node	MCL	CLJ	MSO1	SOJ1	MSO2	SOJ2
	Addition of Node 1 at cycle 160,000					
1	1251124	1	-791127	3		
2	1251125	1	-247794	2	1080073	3
3	1251124	1	-587138	2	-56121	3
4	1251124	1	-639455	3	281728	2
5	1251124	1	-653170	3	728741	3
	Deletion of Node 1 at cycle 160,000					
1	1251124	1	-662749	2		
2	1251125	1	440774	2	541779	3
3	1251124	1	-467403	2	392	4
4	1251123	2	-261420	3	434263	4
5	1251124	1	93592	3	839603	3

Table 5.7: Effect of node addition/deletion on jitter values.

As described in Section 4.2, the CNS scheme supports simple modifications to the topology of a Cyclone network such as the addition or removal of a node. We simulate both cases by adding a node (Node 1) as well as removing one (Node 1) from the baseline dataset after it has reached convergence. Table 5.7 shows the partial results for these two simulations (the full results can be found in Tables A.4 and A.5 in Appendix A). The network reaches convergence in about 150K cycles in both cases, and we start keeping statistics from that point onward. For addition, we added node 1 to the network at around cycle 160K. Similarly, for deletion, we removed node 1 from the network at around the same cycle. The results show that both *cycle length jitters* as well as *start time offset jitters* are completely unaffected by changes to the topology in the form of a node addition or deletion, and therefore CNS is completely resilient to such simple network modifications.

5.10 Perturbations

So far, we have assumed in all previous simulations that latency and clock drift values are fixed. In other words, we assume that they are not affected by things such as temperature changes. In practice, however, perturbation does occur and is caused by a variety of factors. For example, changes in temperature can affect the fiber, causing small changes in the latency values. Similarly, temperature change can also cause the clock oscillator to drift at a slightly faster or slower rate than the nominal value. In both cases, however, the perturbation is usually bounded.

We now introduce latency and clock drift perturbations into the simulation to ensure that the algorithm would still converge even in the presence of such perturbations, and also to determine the affect that they have on the various values at convergence. As indicated by Equation (5.1), the two values we are interested in at convergence are the *cycle length jitter* and the *start time offset jitter*. Without any perturbation, we have shown that both of these values can be bounded to an ϵ value that is in the order of 10 units, or clock ticks. With perturbation, however, we expect the jitter values to increase. Rather than changing our definition of convergence by increasing the ϵ values, we first perform the simulation without any perturbation and obtain its convergence rate ($Cycle_{CONV}$). We then repeat the simulation using the obtained $Cycle_{CONG}$ value, but this time around with the perturbations. This allows us to accurately compare the behavior of the CNS scheme both with and without the presence of perturbations.

5.10.1 Latency Perturbation

In the first set of simulations, we look at how the size of the latency perturbation affects the convergence results. We assume that each link perturbs independently of one another. We also assume that the perturbations do not deviate from the baseline latency values by more than some fixed amount, i.e. that there is a lower and upper bounds. We express the perturbation bounds as a percentage of the desired cycle length, e.g. $\pm 1\%$ or $\pm .1\%$ of the cycle length. We fix the probability of a perturbation at each link at each cycle to 1/1000, and simulate a random walk with a total number of 20 steps between the lower and upper bounds (10 steps above the baseline and 10 steps below the baseline).

Table 5.8 shows how the magnitude of the perturbation affects the convergence values. .01% corresponds to the smallest simulated perturbation, and 1% corresponds to the largest one. To simplify the presentation, we only show the results for the first 10 nodes (see Table A.6 in Appendix A for the complete results). In addition, for each node, we show the *cycle length jitter (CLJ)* as well as the *start time offset jitter (SOJ)* only for the first incoming link to that node (*start time offset jitter* for all remaining incoming links, if any, are omitted for brevity). The results show that latency perturbations do not cause the network to go out of phase, and everything remains in sync. Only the cycle length jitters and

CLJ = cycle length jitter / SOJ = start time offset jitter						
	.01%		.1%		1%	
Node	CLJ	SOJ	CLJ	SOJ	CLJ	SOJ
1	0	284	0	6421	0	33931
2	9	232	98	7264	684	23868
3	10	366	86	6624	804	28305
4	10	383	99	5743	931	24519
5	9	512	104	4877	904	25528
6	10	385	91	4445	1032	21583
7	9	396	96	4398	908	30423
8	9	362	92	2613	897	25979
9	10	316	99	2412	1033	19673
10	10	257	91	2849	903	35991

Table 5.8: Effect of different latency perturbation sizes on jitter values (measured in # of clock ticks). The bounds of the various perturbation sizes are expressed as a percentage of the cycle length.

start time offset jitters are affected. Let's consider the values for the two columns associated with a .01% perturbation. Here, the perturbation is bounded by 125 clock ticks on both side of the base drift rate, and it changes by 12 clock ticks in every instance. Thus we can see that the magnitude of the *cycle length jitter* corresponds to the magnitude of the perturbation step. Similarly, the magnitude of the *start time offset jitter* corresponds to the magnitude of the perturbation bound. This behavior holds true for the remaining values in the table.

We now vary the rate of perturbation, from a probability of 1/1000 all the way

CLJ = cycle length jitter / SOJ = start time offset jitter								
	1/1000		1/100		1/10		1/1	
Node	CLJ	SOJ	CLJ	SOJ	CLJ	SOJ	CLJ	SOJ
1	0	234	0	807	0	1076	0	752
2	9	152	16	842	23	966	40	554
3	9	125	15	1088	24	879	46	583
4	9	214	14	871	25	845	52	578
5	9	130	15	785	25	992	50	558
6	9	146	12	650	23	770	49	591
7	9	211	16	581	24	641	49	583
8	10	186	14	511	23	652	51	618
9	9	273	14	572	25	683	51	576
10	10	239	16	489	28	552	51	588

Table 5.9: Effect of different latency perturbation frequencies, expressed as a probability of a perturbation occurring per link per cycle, on jitter values (measured in # of clock ticks).

up to 1/1 (i.e. a latency perturbation is possible at every cycle at every link), and see how this affects the convergence values. The size of the perturbation bound is set to .01% of the desired cycle length. The results are shown in Table 5.9. As above, only partial results are presented here (the complete results can be found in Table A.7 in Appendix A). We can see that as the frequency of perturbation increases, both the *cycle length jitters* as well as the *start time offset jitters* increase as well. However, even in the worst case scenario where there is potentially a perturbation at every link at every cycle (8000 times per sec), the network remains synchronized. For the particular network topology used in this sets of simulation, the *cycle length jitters* are in the order of .01% of the cycle time. Similarly, the *start time offset jitters* are in the order of .1%.

CLJ = cycle length jitter / SOJ = start time offset jitter								
	0K		1K		30K		55K	
Node	CLJ	SOJ	CLJ	SOJ	CLJ	SOJ	CLJ	SOJ
1	0	1173	0	722	0	47	0	5
2	21	1173	1	705	1	47	1	5
3	47	1175	1	670	1	47	1	5
4	139	1176	2	646	1	47	1	6
5	417	1265	2	588	1	46	1	6
6	139	520	2	485	1	46	1	7
7	47	430	2	401	1	43	1	6
8	21	364	2	337	2	41	2	6
9	9	315	2	289	2	38	2	5
10	5	288	2	263	2	37	2	5

Table 5.10: Rate of adjustment after a single latency perturbation. Each column group represents the # of cycles after the occurrence of the perturbation at cycle 275,000. Jitter values are measured in # of clock ticks.

Finally, since not every system will be subjected to continuous perturbations, i.e. a network in a relatively stable environment may experience such changes only occasionally, we simulate a single perturbation and look at how quickly the system recovers back to its previous converged state. As a reminder, convergence, in the absence of any perturbation, implies that both *cycle length jitters* as well as *start time offset jitters* must be less than some ϵ value, currently set at 10 clock tick units. We induce the 1250-ticks (.1% of C) perturbation at the edge from node 4 to node 5 in the network at cycle number 275K (as seen by the source node of that edge). Rather than keeping track of the various statistics (e.g. max/min cycle length, max/min start time offset) by setting the $Cycle_{CONV}$ value at the point where we believe the system initially converges, we move it to several points after cycle number 275K. By comparing the results we obtained from these different values of $Cycle_{CONV}$, we can determine the rate of adjustment made by the algorithm after a single latency perturbation has occurred.

Table 5.10 shows the partial results for $Cycle_{CONV}$ values set at the perturbation point, 1K cycles afterward, 30K cycles afterward, and 55K cycles afterward (the full results can be found in Table A.8 in Appendix A). At the point where the perturbation occurs (cycle 275K), both cycle length jitter as well as start time offset jitter values increase (CLJ to the hundred and SOJ to the thousand). However, it takes the algorithm only about .12 secs (1K cycles) to bring cycle length jitters back to their convergence values. Start time offset jitters, on the other hand, require about 7 additional secs (55K cycles) to recover.

5.10.2 Clock Drift Perturbation

We now look at how perturbations in the clock drift values affect the convergence results. We assume that the clock at each node perturbs independently of each other, and that the perturbations are bounded by some value (usually 10%) on either side of the baseline clock drift value.

We begin by simulating the same topology with different magnitude of perturbation. We fix the probability of a perturbation at each node at each cycle to be 1/1000, and simulate a random walk with a total number of 20 steps between the lower and upper bounds (10 steps above the baseline and 10 steps below the baseline). For example, given a clock with a baseline drift rate of 100 PPM, the upper bound will be 110 PPM and the lower bound will be 90 PPM, and the drift rate changes by 1 PPM at each step.

CLJ = cycle length jitter / SOJ = start time offset jitter						
	100 PPM		10 PPM		1 PPM	
Node	CLJ	SOJ	CLJ	SOJ	CLJ	SOJ
1	0	69267	0	15804	0	961
2	170	62552	16	15260	3	923
3	55	56970	12	14270	2	866
4	105	52809	11	13990	4	847
5	91	52991	19	13331	3	780
6	136	51989	9	12600	4	713
7	150	44993	21	12140	3	609
8	183	37669	15	10839	3	598
9	124	28596	15	9783	3	596
10	117	23417	18	9246	3	581

Table 5.11: Effect of different clock drift perturbation sizes on jitter values. Each column group represents the size of the perturbation bound. Jitter values are measured in # of clock ticks.

Partial results for bounds in the order of 100 PPM, 10 PPM, and 1 PPM are shown in Table 5.11 (full results can be found in Table A.9 in Appendix A). In the worst case scenario, where a bound of 100 PPM which corresponds to a clock drift accuracy of 1000 PPM, the sizes of the *cycle length jitters* and *start time offset jitters* are only about .01% and 6% of the cycle length, respectively.

Min/Max are deviations of the min/max cycle length						
	100 PPM		10 PPM		1 PPM	
Node	Min	Max	Min	Max	Min	Max
1	-.009%	-.009%	0%	0%	0%	0%
2	-.012%	.002%	-.001%	.001%	0%	0%
3	-.010%	-.006%	-.001%	0%	0%	0%
4	-.015%	-.007%	-.001%	0%	0%	0%
5	-.013%	-.006%	0%	.001%	0%	0%
6	-.012%	-.001%	-.001%	0%	0%	0%
7	-.011%	.001%	-.001%	.001%	0%	0%
8	-.010%	.004%	0%	.001%	0%	0%
9	-.011%	-.001%	-.001%	.001%	0%	0%
10	-.014%	-.004%	0%	.001%	0%	0%

Table 5.12: Effect of different clock drift perturbation sizes on the converged cycle lengths. The deviations are expressed as a percentage of the cycle length obtained without any perturbation.

While link latency perturbation will not ultimately affect the converged cycle length, clock drift rate perturbation could potentially do. This is because the converged cycle length is a function of the clock drift rate of all the nodes, according to Equation (3.18). In Table 5.11, we have only shown the *cycle length jitter values*, which is basically the difference between the maximum and minimum cycle lengths seen during the *statistic keeping period* (i.e. from $Cycle_{CONV}$ until the end of the simulation, $Cycle_{TOTAL}$). When there is no perturbation, the converged cycle length is 1251125 with a jitter value of 1 or 2 clock ticks. We show how far the maximum and minimum cycle lengths (with perturbation) deviate from this 1251125 converged cycle length, when expressed as a percentage of the latter. The partial results, corresponding to Table 5.11, are shown in Table 5.12 (full results can be found in Table A.10 in Appendix A). Even with perturbation bounds of 100 PPM, the cycle length changes by no more than $\pm.02\%$.

Next, we change the frequency of perturbation, from a probability of 1/1000 all the way up to 1/1 (i.e. a clock drift perturbation is possible at every cycle at every node), and see how this affects the convergence values. The bound on the perturbation size is set at 10 PPM. Partial results are shown in Table 5.13 (full results can be found Table A.11 in Appendix A). For this particular network configuration, both the *cycle length jitters* as well as the *start time offset jitters* increase as the probability of a perturbation increase from 1/1000 to 1/100. After that, whereas *cycle length jitters* continue to increase (although not by a significant amount) as the perturbation frequency increases to 1/10 and then eventually 1/1, *start time offset jitters* actually decrease. The network remains synchronized in all cases. The largest cycle length jitter values are in the order of .001% of the cycle length (for the 1/1 case), and those of start time offset jitters are in the order of 2% (for the 1/100 case).

Finally, to evaluate a stable network that is subjected to perturbations only on an infrequent basis, we look at how fast the system recovers from a single

CLJ = cycle length jitter / SOJ = start time offset jitter								
	1/1000		1/100		1/10		1/1	
Node	CLJ	SOJ	CLJ	SOJ	CLJ	SOJ	CLJ	SOJ
1	0	15804	0	20357	0	10447	0	6642
2	16	15260	45	19264	47	9234	56	5009
3	12	14270	48	16110	45	7341	50	4047
4	11	13990	35	15189	45	6859	66	4062
5	19	13331	32	13966	42	6065	45	3486
6	9	12600	28	13244	40	5763	55	3733
7	21	12140	36	11772	46	5726	61	3227
8	15	10839	35	10430	39	4478	58	2701
9	15	9783	35	9005	36	4383	55	2752
10	18	9246	31	7847	41	4951	62	2891

Table 5.13: Effect of different clock drift perturbation frequencies, expressed as a probability of a perturbation occurring per node per cycle, on jitter values (measured in # of clock ticks).

clock drift perturbation. For this simulation, we introduce a perturbation of size +.00001 (equivalent to changing the drift rate by +10 PPM) at node 10 at cycle 275K. We then move $Cycle_{CONV}$, which is used to keep track of jitter values, from cycle 275K forward until we obtain results that are the same as those seen at the initial convergence point (around cycle 260K). Table 5.14 shows the partial results for $Cycle_{CONV}$ values set at the perturbation point, 20K cycles afterward, 30K cycles afterward, and 40K cycles afterward (the full results can be found in Table A.12 in Appendix A). We can see that the perturbation caused a very

CLJ = cycle length jitter / SOJ = start time offset jitter								
	0K		20K		30K		40K	
Node	CLJ	SOJ	CLJ	SOJ	CLJ	SOJ	CLJ	SOJ
1	0	578	0	38	0	38	0	8
2	1	578	1	38	1	37	1	8
3	1	579	1	38	1	37	1	9
4	1	580	1	38	1	38	1	9
5	1	582	1	37	1	37	1	9
6	2	585	1	36	1	35	1	8
7	2	588	1	35	1	36	1	9
8	4	592	1	34	1	33	1	8
9	8	595	2	31	2	31	2	8
10	15	538	3	29	2	30	2	8

Table 5.14: Rate of adjustment after a single clock drift perturbation. Each column group represents the # of cycles after the occurrence of the perturbation at cycle 275,000. Jitter values are measured in # of clock ticks.

small increase in the *cycle length jitters*, and in fact those jitters returned to their convergence values quickly afterward (within less than 1K cycles–not shown in the table). The impact on *start time jitters* are more significant, and the network requires about 5 secs (or 40K cycles) to bring them back to their pre-perturbation values.

CLJ = cycle length jitter / SOJ = start time offset jitter				
	100 ticks / 100 PPM		10 ticks / 10 PPM	
Node	CLJ	SOJ	CLJ	SOJ
1	0	94829	0	6768
2	166	86933	23	6541
3	147	73981	16	5668
4	206	70117	32	5258
5	168	64554	18	4086
6	149	61668	12	3199
7	125	60267	16	3408
8	212	59083	14	3123
9	159	57853	15	3249
10	130	55260	16	3436

Table 5.15: Effect of both latency and clock drift perturbation on jitter values. Each column group represents the bound on the size of the perturbations ("ticks" for latency and "PPM" for the corresponding clock drift). Jitter values are measured in # of clock ticks.

5.10.3 Both latency and clock drift perturbations

Using the same network setup as above, we now introduce both latency and clock drift perturbations into the simulation at the same time. We assume that each perturbation, whether latency or clock drift, is independent of any other, and fix the probability at 1/1000. We set the size of a latency perturbation to be on the same order of magnitude as the size of the clock drift perturbation. For example, if the bound on the latency perturbation is 100 clock ticks, then that of clock drift perturbation would be 100 PPM. We simulate with latency perturbation bounds of 100 ticks and 10 ticks, with corresponding clock drift perturbation bounds of 100 PPM and 10 PPM, respectively. For both cases, each perturbation step is 1/10 of the perturbation bound.

The partial results for this simulation are shown in Table 5.15 (full results can be found in Table A.13 in Appendix A). If we compare the jitter values for the *100 ticks/PPM* columns with those found in the *100 PPM* columns in Table 5.11, we can see that they are of the same order of magnitude. The same holds true for the values found in the *10 ticks/PPM* columns and *10 PPM* columns in the two tables. Basically, the clock drift perturbations are having a much larger impact on the jitter values, especially the *start time offset jitter* ones, than the latency perturbations. The system remains in sync with very small *cycle length jitter values* in both cases.

5.11 Summary

The simulation results presented in this chapter show that the CNS scheme achieves convergence in all cases, for all parameter values. Simple node additions and deletions are handled in CNS with no susceptible changes to either cycle length jitter or start time offset jitter values. When perturbations are introduced into the simulation, the results show that while jitter values do increase as expected, they remain tolerable in the typical cases and, more importantly, do not cause the already synchronized network to go out of phase.

Chapter 6

Related Works

Coordination in a computer network, and distributed systems in general [39, 34, 15, 43, 26, 16, 46, 7], have been addressed in various contexts such as: mutual exclusion [13], consensus agreement [25], concurrency control [6, 5], deadlock detection [24], termination detection [14], and clock synchronization. Even the area of clock synchronization can be divided into physical clock synchronization, where we are concerned with the occurrence of an event on the basis of time [36, 44], and logical clock synchronization [35, 3], where only the logical ordering of events are important.

In this chapter, we look at some current approaches to physical clock synchronization since that particular area is most closely related to the work presented in this dissertation. Specifically, we describe Cristian Algorithm, Berkeley Algorithm, the Network Time Protocol (NTP), and the Precision Time Protocol (PTP). In addition, we also look at how synchronization is achieved in the DTM Gigabit network, as it is somewhat similar in nature to the Cyclone network architecture. We conclude by comparing these approaches with the CNS scheme.

6.1 Cristian Algorithm

This centralized clock synchronization algorithm [12] assumes the existence of an external reference clock (e.g. Cesium, GPS) that is connected to one of the nodes, referred to as the *master node*. The local clock on the master node is synchronized to this reference clock. The remaining nodes, the *clients*, then synchronize their local clocks to the *master node*'s local clock.

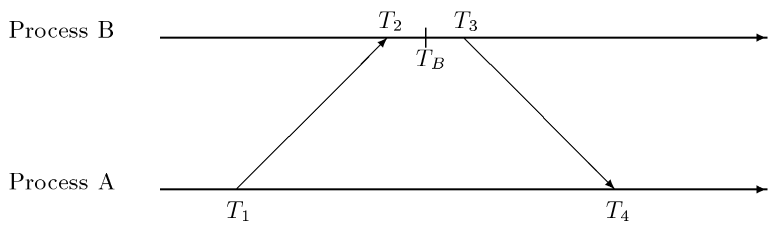

Figure 6.1: Cristian Algorithm.

The interaction between the client and server is shown in Figure 6.1. The client (process A) starts by sending a message at time T_1 to the master (process B). The master receives the message at time T_2. The master then looks up its current time, T_B, and sends this value back to the client at time T_3. The client receives the message containing T_B at time T_4. Note that T_1 and T_4 are measured according to the client's local clock. Similarly, T_2, T_B, and T_3 are measured according to the server's local clock.

$T_4 - T_1$ is the total round trip time as observed by the client. $T_3 - T_2$ is the time it takes for the master to respond to the request from the client, and may include queueing delay as well as processing time. The client computes the time offset, i.e. the difference between its clock and the master node's clock, by using

$$offset = \frac{(T_4 - T_1) - (T_3 - T_2)}{2} \tag{6.1}$$

and then adjusts its local clock by adding to this offset value. In the above equation, there is an assumption that the latency from client to server (forward path) is the same or very close to the latency from server to client (backward path), or *symmetric latency*. Similarly, if $T_3 - T_2$ is not known or provided by the server, then this value is assumed to be either (a) very small compare to the latency values, or (b) that T_B is in the middle of the interval.

To prevent the master from being a single point of failure, multiple masters can be configured. This, however, adds to the complexity and overhead of the algorithm. Experiments using this algorithm have shown that accuracy in the milliseconds range can be achieved.

6.2 Berkeley Algorithm

In Berkeley Algorithm [28], there is no external reference clock like in the case for Cristian Algorithm. Instead, the nodes synchronize their local clocks among themselves. A node is selected to be the *server*. Periodically, the server sends a message to each client to determine, using an algorithm similar to Cristian Algorithm, the clock offset between itself and that of the client (Figure 6.2(a)). The server then computes the average of all the clock offsets it has collected (Figure 6.2(b)), and sends to each client the difference between this average and the client's clock offset (which it had collected in the first step). Every node, including the master, then adjusts its local clock accordingly to achieve system wide synchrony (Figure 6.2(c)).

To prevent an errand or misbehaving clock from affecting the whole network, the averaging computation can be adjusted. For example, the smallest or largest offset values can be thrown out. Another approach is to consider only clock

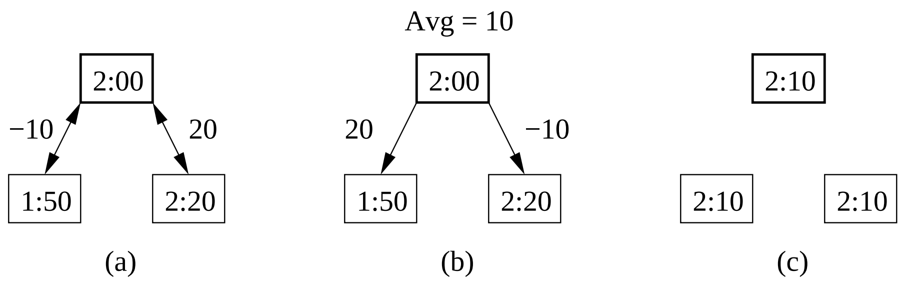

Figure 6.2: Berkeley Algorithm

offsets that do not differ from each other by some fixed amount. As in the case of Cristian Algorithm, there is an implicit assumption that the latencies of the forward and return path are the same or very close to each other. For fault tolerance purpose, if the master fails, another master can be selected from the remaining nodes [29]. Experiments using the algorithm have shown accuracy in the milliseconds range [30].

6.3 NTP

The Network Time Protocol (NTP) [41] is designed to maintain time synchrony in a wide area network, where messages may have to cross multiple gateways or routers, and network link latency can be unpredictable or even unreliable. The "time servers" in NTP are organized in a hierarchical subnet with the top level servers connected to external reference clocks (Figure 6.3(a)). These top level servers are considered to be at "stratum 1" level[1]. At one level down are servers designated to be at stratum 2, which synchronize themselves to those at stratum 1, and so on [42]. Each node may be connected to multiple nodes at a higher or same stratum level for fault tolerance purposes. The NTP subnet reconfigures

[1]The NTP definition of "stratum" does not correspond to the ITU definition.

itself in case of a node or link failure (Figure 6.3(b)).

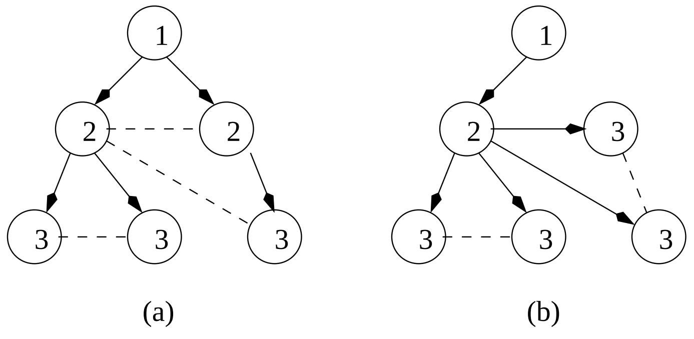

Figure 6.3: NTP subnet

An NTP node uses clock offsets values obtained from its subnet peers in order to synchronize its local clock. These offset values are first passed through a set of filters to reduce incidental timing noise. A peer-selection algorithm then determines the most accurate values based on criteria such as the distance between the server and the peer. Finally the resulting subset are combined on a weighted-average basis to create the actual adjustment value. Past results are a factor considered when computing future values.

There are actually three *modes* of operation in NTP. In *multicast mode*, where a high degree of accuracy is not required, a single server periodically broadcasts its timestamp values to a set of clients. The client computes the clock offset by assuming a link latency of a few milliseconds. If a multicast environment is not available, or if a better accuracy is needed, *procedure-call mode* is used. In this mode, a client requests and receives the timestamp values from the server, using a method that is similar to Cristian Algorithm (Figure 6.1). The third mode, *symmetric mode*, is used by a pair of servers to exchange timestamps between themselves, with each node acting as both a server and a client alternatively. This mode allows the two servers to maintain the highest synchronization accuracy between their clocks. In a general wide area network (i.e. the Internet), NTP can achieve an accuracy in the 10 milliseconds range. For a small local area network with low traffic, the accuracy in the 200 microseconds range can be achieved [40].

6.4 PTP

IEEE-1588 defines a Precision Timing Protocol (PTP) for used in a local area network such as ethernet by control and measurement systems [22], where high timing measurement accuracy is required. For protocols that send timestamps at a high level (e.g. at the software application level), there is a potential delay between the instance the timestamp is recorded and the time it is actually transmitted by the network interface controller (NIC). This delay is caused by the application making a system call and going through the kernel network stack, then to the NIC buffer, and finally onto the physical medium. A similar delay occurs at the receiving end. If these delays are not properly accounted for, and end up being treated as part of the network latency, then the clock offset computation will not be as accurate. PTP seeks to overcome this problem by requiring that the nodes contain hardware support that will perform the timestamping operation at the closest possible point to the network, with the ideal case being (i) just before the message is put onto, or (ii) just after the message is retrieved from, the physical medium [21].

Consider the message passing sequence shown in Figure 6.4, where Process B (the client) is trying to synchronize with the clock on Process A (the server). A starts out by sending a message containing the timestamp T_1 to B. Immediately afterward, it sends another message to B containing the actual time, $T_1^{'}$, just

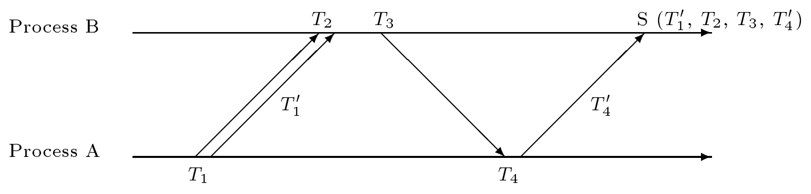

Figure 6.4: Precision Time Protocol

before the first message enter the wire (as measured by its special hardware). The purpose of these first two messages is to factor out the delay associated with the sending of a message from A. B now sends a message to A, and after receiving this message (at the application level), A sends back to B the actual time, T_4', just after the message enters the wire (again as measured by its special hardware). The purpose of these last two messages is to factor out the delay associated with the receiving of a message from A. At point S, B will then know about T_1', T_2, T_3, and T_4'. Using these 4 timestamp values, and by assuming that the forward and backward latencies are the same, B can compute the clock offset between it and the server (A), and adjusts its clock accordingly. In addition to defining the message passing protocol, PTP also defines a *Best Master Clock* (BMC) algorithm [1] that is used by the nodes to determine which one should be a server and which ones should be clients, as well the hierarchy among them. BMC is highly dependent on the network layer being a broadcast medium. With the proper hardware support, PTP can achieve clock synchronization accuracy in the sub-microsecond range.

6.5 DTM

Dynamic Synchronous Transfer Mode (DTM) is a fiber-optic network architecture designed to support a variety of applications, from voice to data, as well as those with quality of service (QOS) requirements [17]. Nodes in a DTM subnet are arranged in a dual buses topology, where each bus is a fiber carrying data in a particular direction. Multiple subnets can be connected together by using nodes that are attached to more than one dual buses (Figure 6.5). In DTM, a TDMA-based scheme is used, where the bandwidth is divided into consecutive *frames* of a fixed-size time interval (e.g. 125 microseconds). Each frame is divided into a number of *slots* of 64-bit each. The number of slots in a frame depends on the speed of the link. For example, in a 622 Mbit/s OC-192 network, there would be around 1200 slots per cycle. There are two types of slots: *static* and *dynamic* slots. The formers are used for control and the latters for application data.

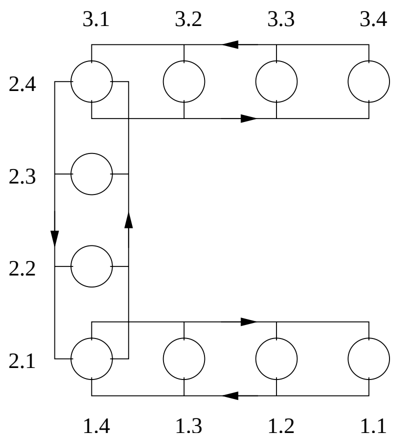

Figure 6.5: DTM network

DTM uses a scheme similar to Cyclone to synchronize the cycles across the multiple buses [9]. Each cycle consists of a *start* slot, follows by the data slots. In between cycles are one or more *fill* slots (these could be empty data slots). To ensure that the cycle lengths are the same on all the buses in the network, DTM synchronize the start time of the cycles across the different buses. The nodes are organized in a hierarchical manner with one *master* node and multiple *slave* nodes. The master is located at one end of a dual buses, and is assumed to be connected to an external reference clock. The master controls the cycle length on all of its outgoing buses by periodically sending out start slots. At the other end of each dual bus is a slave node. The slave listens to its incoming, or *trigger*, bus. When it sees a start slot, the slave then initiates a new cycle on all of its outgoing bus (there may be more than one) by sending out a start slot on these buses. Once the data for the current cycle has been transmitted, the slave then sends out one or more fill slots until it sees the next start slot on its trigger bus.

As an example, let node 1.1 be the master in Figure 6.5. It will control the cycle length on the right-to-left bus in subnet 1. At the other end of this subnet is node 1.4 (aka 2.1), which controls the cycle length for the left-to-right bus on subnet 1, as well as the bottom-to-top bus on subnet 2. Node 2.4 (aka 3.1) is a slave on subnet 2, and it controls the cycle length on the top-to-bottom bus on subnet 2, along with the left-to-right bus on subnet 3. Finally, node 3.4 is a slave on subnet 3, and it controls the right-to-left bus on this subnet. The accuracy of the cycle synchronization in DTM is dependent on that of the master external reference clock, as well as how closely its internal clocks are kept in sync with that clock.

6.6 Comparison to CNS

We now compare and contrast the various characteristics of the CNS scheme with the approaches described in the previous sections:

- In the message overhead passing category, CNS is extremely “lightweight” since no messages are being sent for the purpose of synchronization. DTM is similar in this respect. The other clock synchronization approaches have to exchange timestamps in order to compute the clock offset values. The downside to the CNS approach is that synchronization are only possible at cycle endpoints. In between those endpoints, events can still be coordinated, although the accuracy may no longer be as accurate compare to those achieved at the endpoints.

- CNS is a decentralized scheme, where there is no special distinction among the nodes and they all execute the exact same algorithm. Should a node failed in CNS, the remaining nodes will continue to function properly. The other approaches all have the concept of one or more master nodes that the remaining nodes will synchronize with (e.g. stratum-1 nodes in NTP, BMC in PTP, "master" node in Cristian, Berkeley, and DTM).

  We note that it is trivial to modify CNS so that there is conceptually a "master" node to which all other node will eventually sync up with. Basically, this master node simply computes the cycle length by using its local clock, and does not resort to using the CNS average algorithm like the remaining nodes do. Simulations have shown that the network will converge to the cycle length dictated by the local clock of the master node. This modification does not change the decentralized aspect of CNS, but it does allow for use of a highly accurate clock (e.g. Cesium) to synchronize the whole Cyclone network.

- While we do assume that links are point-to-point in Cyclone, CNS is not dependent on any particular network topology (e.g. bus, star, ring). PTP nodes depend on a broadcast medium such as Ethernet in order to carry out the BMC algorithm, and DTM assumes a dual buses layout. In addition, CNS does not require that the latency on the forward path (A to B) be equal to or be close to that of the backward path (B to A). This is unlike Cristian Algorithm, Berkeley Algorithm, or PTP, all of which assume a symmetric latency delay.

- The computations in CNS are relatively simple, and consisting of integer operations only. Its complexity should therefore be less than the full NTP implementation or even the BMC implementation in PTP.

- As shown in the simulation results, CNS can achieve a very high degree of synchronization precision, one that is not dependent on the clock drift values, but only on the clock drift jitter values. If we were to implement the other algorithms on top of Cyclone, then it is conceivable that they could achieve a similar degree of accuracy, due to the fact that the latency values are very predictable (even if there is no symmetric latency delay).

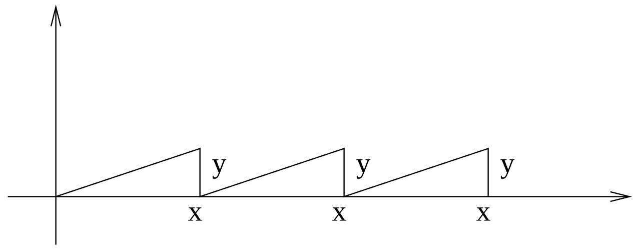

Figure 6.6: Sawtooth effect

However, one characteristic of synchronization approaches that adjust clocks on a regular interval is what we term the *sawtooth effect*. Figure 6.6 shows the difference between two clocks being synchronized with each other over time. Let the x's represent the synchronization points, so the intervals between them correspond to the clock synchronization intervals. At each x, the clocks are synchronized and therefore the time difference is zero. In between the x's, one clock will drift relative to the other, and the y's represent the maximum time difference. The value of y depends on the clock drift rates as well as the length of the interval. If synchronization events occur at x, then the accuracy can be very high. However, this accuracy will decrease if these events occur slightly to the left of x. With CNS, there is no sawtooth effect because it does not perform clock synchronization. Instead, the corresponding y values are very small, and depend only on the clock drift jitters as mentioned previously.

Chapter 7

Concluding Remarks

A new method for achieving global synchronicity in a distributed system is presented in this dissertation. This method, referred to as the Cyclone Network Synchronization (CNS) scheme, does not require that the local clocks on the various nodes be synchronized with each other or with a set of external clocks. CNS relies on the ability of each node to send data at a time of its choosing. Such data are sent at regular interval, with the next instance being determined based only on the local information available at the node. Once the scheme converges, the interval for all nodes becomes exactly the same, supporting a synchronous operation across the whole network. CNS takes into account the finite precision arithmetic and measurements it has to use, while still maintaining global synchrony with very small jitter values.

By using local clocks that are free-running and thus allowed to drift at their own rates, CNS does not suffer from some of the drawbacks commonly exhibited by approaches based on local clock synchronization. Specifically, the scheme does not require the use of highly accurate external clocks such as a Cesium or GPS clock, which can significantly add to the cost of the overall system. Unlike some clock synchronization methods, CNS does not also depend on any particular network characteristic such as a broadcast medium or bus structure, or the requirement that the latency on the sending path be equal to the latency on the receiving path of a message exchange (symmetric latency). The scheme is decentralized, with no special node nor the need to select one, and does not require reconfiguration in case of node failures, assuming the failure does not partition the network. It is also relatively simple, making use of only integer arithmetic operations, and therefore can be implemented in hardware if necessary. Finally, the synchronization accuracy that can be achieved in CNS is dependent only on the granularity of the timestamp counter, along with the perturbation caused by clock drift and latency jitters. However, this accuracy is not a function of the local clock drift rates, as is the case for all other clock synchronization methods.

CNS does require an initialization phase during which the network is not synchronized. The time taken by this delay is determined by the latency values as well as the network topology. For the typical LAN or WAN networks that are in used today, simulations have shown that the time taken to reach convergence is small and acceptable (in the order of 1 to 5 minutes). The scheme also incurs some overhead in terms of the additional padding of the cycle length. Again, simulations have shown that these extra gaps amount to only about .001% of the desired interval (cycle length). Finally, because CNS is not a clock synchronization scheme, local clock values cannot be used for off-line comparison of time instances in tasks such as logfile analysis. However, there is currently ongoing works to perform explicit clock synchronization by using CNS as a starting point. In this approach, a "common clock" is logically defined for the whole system, and each node then maintains a mapping that converts its local clock value to the corresponding common clock value, and vice versa. Logfile values can then be saved either as common clock values, or as local clock values along with the information needed to recreate the clock mapping function. The local clocks themselves are still free-running and unsynchronized just as they are in the CNS scheme.

Appendix A

Complete Simulation Results

In order to facilitate the presentation of the simulation results, some of the tables in Chapter 5 include only a partial set of values. The corresponding tables in this Appendix contain the complete results.

Partial results	**Complete results**
Table 5.1	Table A.1
Table 5.2	Table A.2
Table 5.2	Table A.3
Table 5.7	Table A.4
Table 5.7	Table A.5
Table 5.8	Table A.6
Table 5.9	Table A.7
Table 5.10	Table A.8
Table 5.11	Table A.9
Table 5.12	Table A.10
Table 5.13	Table A.11
Table 5.14	Table A.12
Table 5.15	Table A.13

MCL = minimum cycle length / CLJ = cycle length jitter
MSO = minimum start time offset / SOJ = start time offset jitter

Node	MCL	CLJ	MSO1	SOJ1	MSO2	SOJ2
1	1251125	1	-15014	1		
2	1251126	0	15002	1	-85232	2
3	1251125	1	85393	2	-1004033	3
4	1251125	1	1003129	2	81907	3
5	1251125	1	-81859	2	70431	2
6	1251125	1	-70439	2	13769	2
7	1251125	1	-13783	2	107699	2
8	1251125	1	-107614	2	55269	2
9	1251125	1	-55281	2	47212	1
10	1251126	1	-47217	2	39316	1
11	1251125	1	-39320	1	29637	1
12	1251125	1	-29640	1	33793	3
13	1251125	2	-33781	2	-15221	3
14	1251124	2	15228	3	-6875	3
15	1251125	1	6874	2	62986	2
16	1251124	2	-62912	3	-103062	2
17	1251125	1	103216	2	33003	1
18	1251125	1	-33004	1	10402	2
19	1251125	1	-10404	2	-3356	1
20	1251125	1	3356	1		

Table A.1: Simulation outputs for baseline dataset

MCL = minimum cycle length / CLJ = cycle length jitter

MSO = minimum start time offset / SOJ = start time offset jitter

Node	MCL	CLJ	MSO1	SOJ1	MSO2	SOJ2
1	125112	1	-1504	1		
2	125112	0	1502	1	-8497	1
3	125111	1	8513	1	-100379	3
4	125112	1	100287	2	8212	2
5	125111	2	-8208	2	7064	2
6	125112	1	-7066	2	1397	2
7	125112	1	-1399	2	10785	2
8	125111	2	-10778	2	5542	2
9	125112	1	-5544	2	4737	3
10	125112	1	-4739	2	3943	2
11	125111	2	-3944	2	2973	3
12	125111	2	-2975	3	3387	3
13	125112	1	-3387	2	-1516	2
14	125111	1	1516	2	-683	3
15	125112	1	681	3	6301	2
16	125111	2	-6295	2	-10300	2
17	125111	1	10315	2	3306	1
18	125111	1	-3307	1	1042	2
19	125112	1	-1043	2	-336	2
20	125111	1	335	2		

Table A.2: Simulation outputs for cycle length of 125,000 clock ticks

MCL = minimum cycle length / CLJ = cycle length jitter

MSO = minimum start time offset / SOJ = start time offset jitter

Node	MCL	CLJ	MSO1	SOJ1	MSO2	SOJ2
1	12511260	1	-150138	1		
2	12511260	0	150017	1	-852606	2
3	12511259	1	854226	2	-10040563	2
4	12511259	1	10031534	3	818855	2
5	12511259	1	-818365	2	704078	2
6	12511259	1	-704150	3	137480	2
7	12511259	2	-137605	3	1076806	2
8	12511259	1	-1075946	2	552544	2
9	12511260	1	-552656	2	471969	2
10	12511260	1	-472017	1	393039	2
11	12511259	1	-393079	2	296275	2
12	12511259	1	-296306	2	337863	3
13	12511259	2	-337730	3	-152259	3
14	12511259	1	152348	3	-68764	3
15	12511259	2	68769	3	629851	2
16	12511259	1	-629097	2	-1030671	2
17	12511259	1	1032217	2	329971	2
18	12511259	1	-329973	2	104001	3
19	12511259	1	-104013	2	-33561	2
20	12511259	1	33563	2		

Table A.3: Simulation outputs for cycle length of 12,500,000 clock ticks

MCL = minimum cycle length / CLJ = cycle length jitter

MSO = minimum start time offset / SOJ = start time offset jitter

Node	MCL	CLJ	MSO1	SOJ1	MSO2	SOJ2
1	1251124	1	-791127	3		
2	1251125	1	-247794	2	1080073	3
3	1251124	1	-587138	2	-56121	3
4	1251124	1	-639455	3	281728	2
5	1251124	1	-653170	3	728741	3
6	1251124	1	-286274	3	300741	2
7	1251124	1	-400746	3	268506	3
8	1251124	1	-703848	3	640803	3
9	1251124	2	-443342	2	270209	3
10	1251124	2	-100622	2	381972	2
11	1251124	1	-276456	3	722063	3
12	1251124	1	-344649	2	-31536	3
13	1251124	2	-563312	2	-113020	3
14	1251123	2	-660155	3	701730	3
15	1251124	2	-966572	3	-33648	3
16	1251123	2	-173708	3	348899	2
17	1251124	2	-275656	3	-258518	2
18	1251124	1	-402598	2	357539	2
19	1251124	1	-217862	1	-720939	2
20	1255124	1	-772762	1		

Table A.4: Effect of adding node 1 @ cycle 160,000 on jitter values.

MCL = minimum cycle length / CLJ = cycle length jitter
MSO = minimum start time offset / SOJ = start time offset jitter

Node	MCL	CLJ	MSO1	SOJ1	MSO2	SOJ2
1	1251124	1	-662749	2		
2	1251125	1	440774	2	541779	3
3	1251124	1	-467403	2	392	4
4	1251123	2	-261420	3	434263	4
5	1251124	1	93592	3	839603	3
6	1251124	1	-1099458	4	971865	3
7	1251124	1	-951103	3	118119	4
8	1251124	1	-418047	4	499417	4
9	1251124	2	-193610	4	320277	3
10	1251124	2	-457443	3	630655	3
11	1251123	2	-302169	3	1086810	3
12	1251124	1	-574009	3	-119678	4
13	1251124	2	-934783	3	85531	3
14	1251123	2	-1024571	3	839525	3
15	1251124	1	-117301	3	621671	3
16	1251123	2	-803828	3	112978	2
17	1251124	2	-97867	3	63478	2
18	1251124	2	-1046356	2	267418	3
19	1251123	2	-389922	3	475397	3
20	1251124	1	-688654	2		

Table A.5: Effect of removing node 1 @ cycle 160,000 on jitter values.

N = node, C = cycle length jitter, S1/S2 = start time offset jitter for links 1/2 Column groups = bound on size of latency perturbation as % of cycle length									
	.01%			.1%			1%		
N	C	S1	S2	C	S1	S2	C	S1	S2
1	0	284		0	6421		0	33931	
2	9	232	232	98	7264	7199	684	23868	23642
3	10	366	360	86	6624	6605	804	28305	27967
4	10	383	383	99	5743	5771	931	24519	24990
5	9	512	517	104	4877	4819	904	25528	25455
6	10	385	391	91	4445	4379	1032	21583	21007
7	9	396	395	96	4398	4394	908	30423	30083
8	9	362	361	92	2613	2611	897	25979	26052
9	10	316	316	99	2412	2457	1033	19673	20527
10	10	257	254	91	2849	2930	903	35991	35872
11	10	261	260	97	3997	3986	928	32919	33098
12	11	300	299	97	4297	4331	925	31272	30997
13	9	281	287	88	4267	4266	897	38213	38612
14	10	324	327	89	4325	4301	870	34292	33437
15	9	281	279	96	3793	3784	899	32437	33100
16	9	238	241	92	2962	2998	877	32434	32471
17	9	196	190	89	3632	3585	988	45478	45052
18	10	110	110	97	3041	3000	896	47522	47555
19	10	146	135	106	1636	1587	928	17777	17658
20	14	27		132	264		1311	2621	

Table A.6: Effect of different latency perturbation sizes on jitter values.

N = node, C = cycle length jitter, S1/S2 = start time offset jitter for links 1/2
Column groups = probability of a latency perturbation at each cycle

	1/1000			1/100			1/10			1/1		
N	C	S1	S2	C	S1	S2	C	S1	S2	C	S1	S2
1	0	234		0	807		0	1076		0	752	
2	9	152	151	16	842	832	23	966	965	40	554	529
3	9	125	128	15	1088	1093	24	879	861	46	583	556
4	9	214	211	14	871	865	25	845	844	52	578	584
5	9	130	136	15	785	781	25	992	976	50	558	539
6	9	146	153	12	650	650	23	770	771	49	591	601
7	9	211	203	16	581	583	24	641	643	49	583	592
8	10	186	183	14	511	517	23	652	644	51	618	622
9	9	273	266	14	572	569	25	683	683	51	576	567
10	10	239	238	16	489	486	28	552	560	51	588	594
11	10	144	135	15	584	579	23	674	680	48	560	576
12	10	204	201	17	565	563	24	611	607	55	600	578
13	9	170	170	16	726	720	24	639	637	50	652	656
14	9	202	198	16	695	689	24	684	683	46	566	580
15	11	213	210	18	653	665	25	605	598	48	578	566
16	10	300	301	15	636	631	24	595	592	50	584	584
17	10	300	301	16	570	569	24	673	676	55	706	701
18	10	192	193	14	540	530	24	769	781	55	719	706
19	10	209	212	17	490	485	28	514	518	58	583	605
20	14	27		17	34		34	67		64	128	

Table A.7: Effect of different latency perturbation frequencies on jitter values.

N = node, C = cycle length jitter, S1/S2 = start time offset jitter for links 1/2
Column groups = # of cycles after perturbation @ cycle 275,000

	0K			1K			30K			55K		
N	C	S1	S2	C	S1	S2	C	S1	S2	C	S1	S2
1	0	1173		0	722		0	47		0	5	
2	21	1173	1174	1	705	705	1	47	48	1	5	6
3	47	1175	1176	1	670	670	1	47	47	1	5	5
4	139	1176	1177	2	646	646	1	47	48	1	6	6
5	417	1265	797	2	588	586	1	46	46	1	6	6
6	139	520	518	2	485	482	1	46	44	1	7	6
7	47	430	428	2	401	399	1	43	44	1	6	6
8	21	364	362	2	337	335	2	41	41	2	6	5
9	9	315	312	2	289	286	2	38	38	2	5	5
10	5	288	286	2	263	262	2	37	37	2	5	5
11	2	252	250	1	230	229	1	33	33	1	5	5
12	1	219	218	1	199	199	1	30	30	1	4	4
13	1	190	189	1	172	171	1	27	27	1	4	4
14	2	153	153	2	138	138	1	23	23	1	4	4
15	2	134	133	2	121	121	1	20	20	1	4	3
16	2	104	103	2	96	94	1	17	15	1	4	2
17	2	71	70	2	52	50	1	12	12	1	3	3
18	2	54	53	2	52	50	2	9	9	2	3	2
19	2	29	27	2	27	26	2	6	5	2	3	2
20	1	3		1	3		1	3		1	2	

Table A.8: Rates of adjustment after a latency perturbation.

Cycle = cycle length jitter, Link1/Line2 = start time offset jitter for links 1/2

Column groups = bound on size of clock drift perturbation

Node	100 PPM			10 PPM			1 PPM		
	Cycle	Link1	Link2	Cycle	Link1	Link2	Cycle	Link1	Link2
1	0	69267		0	15804		0	961	
2	170	62552	62177	16	15260	15231	3	923	922
3	55	56970	52564	12	14270	14243	2	866	865
4	105	52809	52877	11	13990	13990	4	847	844
5	91	52991	53038	19	13331	13306	3	780	780
6	136	51989	51720	9	12600	12588	4	713	709
7	150	44993	44845	21	12140	12094	3	609	607
8	183	37669	37437	15	10839	10803	3	598	599
9	124	28596	28312	15	9783	9758	3	596	595
10	117	23417	23240	18	9246	9214	3	581	581
11	83	22470	22596	18	7917	7875	4	495	491
12	161	28012	28224	12	6889	6869	3	354	350
13	81	30178	30074	23	6240	6207	3	244	242
14	120	22098	21090	14	5149	5123	3	200	200
15	143	15935	15719	16	4580	4554	3	206	206
16	164	13079	13055	15	3660	3632	4	102	101
17	80	13917	14045	16	2626	2599	3	93	94
18	149	12264	12160	19	2093	2080	3	106	107
19	117	6150	5948	17	1097	1059	3	51	50
20	105	210		19	33		3	6	

Table A.9: Effect of different clock drift perturbation sizes on jitter values.

Min/Max = minimum/maximum cycle length

Column groups = bound on size of clock drift perturbation

	100 PPM		10 PPM		1 PPM	
Node	Min	Max	Min	Max	Min	Max
1	-.009%	-.009%	0%	0%	0%	0%
2	-.012%	.002%	-.001%	.001%	0%	0%
3	-.010%	-.006%	-.001%	0%	0%	0%
4	-.015%	-.007%	-.001%	0%	0%	0%
5	-.013%	-.006%	0%	.001%	0%	0%
6	-.012%	-.001%	-.001%	0%	0%	0%
7	-.011%	.001%	-.001%	.001%	0%	0%
8	-.010%	.004%	0%	.001%	0%	0%
9	-.011%	-.001%	-.001%	.001%	0%	0%
10	-.014%	-.004%	0%	.001%	0%	0%
11	-.015%	-.008%	0%	.001%	0%	0%
12	-.022%	-.009%	0%	.001%	0%	0%
13	-.011%	-.005%	-.001%	.001%	0%	0%
14	-.010%	0%	-.001%	0%	0%	0%
15	-.011%	.001%	0%	.001%	0%	0%
16	-.015%	-.002%	0%	.001%	0%	0%
17	-.013%	-.006%	0%	.001%	0%	0%
18	-.014%	-.002%	0%	.001%	0%	0%
19	-.011%	-.001%	0%	.001%	0%	0%
20	-.011%	-.003%	-.001%	.001%	0%	0%

Table A.10: Effect of different clock drift perturbation sizes on cycle lengths.

N = node, C = cycle length jitter, S1/S2 = start time offset jitter for links 1/2
Column groups = probability of a clock drift perturbation at each cycle

	1/1000			1/100			1/10			1/1		
N	C	S1	S2	C	S1	S2	C	S1	S2	C	S1	S2
1	0	15804		0	20357		0	10447		0	6642	
2	16	15260	15231	45	19264	19222	47	9234	9168	56	5009	4965
3	12	14270	14243	48	16110	16005	45	7341	7294	50	4047	4050
4	11	13990	13990	35	15189	15131	45	6859	6819	66	4062	4044
5	19	13331	13306	32	13966	13934	42	6065	6046	45	3486	3496
6	9	12600	12588	28	13244	13187	40	5763	5702	55	3733	3703
7	21	12140	12094	36	11772	11723	46	5726	5683	61	3227	3156
8	15	10839	10803	35	10430	10390	39	4478	4444	58	2701	2686
9	15	9783	9758	35	9005	8934	36	4383	4379	55	2752	2758
10	18	9246	9214	31	7847	7842	41	4951	4895	62	2891	2885
11	18	7917	7875	25	8644	8664	42	4485	4485	63	2729	2779
12	12	6889	6869	43	8499	8470	41	4828	4871	70	2935	2983
13	23	6240	6207	36	7461	7427	43	5203	5215	63	2950	2920
14	14	5149	5123	32	6401	6380	35	5085	5071	51	3014	3074
15	16	4580	4554	34	5130	5094	40	4616	4611	51	2853	2896
16	15	3660	3632	28	5502	5541	36	4543	4557	53	2845	2855
17	16	2626	2599	39	4324	4236	39	4002	3977	52	2773	2811
18	19	2093	2080	39	3300	3260	41	3497	3536	55	2869	2904
19	17	1097	1059	30	1774	1723	40	2114	2080	52	1901	1893
20	17	33		31	63		48	96		71	141	

Table A.11: Effect of different clock drift perturbation rates on jitter values.

N = node, C = cycle length jitter, S1/S2 = start time offset jitter for links 1/2

Column groups = # of cycles after perturbation @ cycle 275,000

	0K			20K			30K			40K		
N	C	S1	S2	C	S1	S2	C	S1	S2	C	S1	S2
1	0	578		0	38		0	38		0	8	
2	1	578	578	1	38	38	1	37	38	1	8	9
3	1	579	580	1	38	39	1	37	39	1	9	10
4	1	580	581	1	38	38	1	38	38	1	9	9
5	1	582	581	1	37	37	1	37	38	1	9	10
6	2	585	586	1	36	37	1	35	37	1	8	10
7	2	588	588	1	35	35	1	36	35	1	9	9
8	4	592	592	1	34	33	1	33	34	1	8	9
9	8	595	596	2	31	31	2	31	32	2	8	9
10	15	538	501	3	29	30	2	30	30	2	8	8
11	8	293	291	2	27	27	2	27	28	2	7	7
12	4	255	253	2	25	24	1	25	24	1	7	6
13	3	218	217	1	21	23	1	22	22	1	6	6
14	2	176	175	1	19	19	1	18	18	1	5	5
15	2	153	153	1	17	17	1	17	17	1	5	5
16	1	118	117	1	14	14	1	17	14	1	5	4
17	1	81	80	1	10	11	1	10	10	1	3	3
18	2	62	61	1	8	8	2	9	7	2	4	3
19	2	34	31	2	6	5	2	6	5	2	3	2
20	2	3		2	3		1	2		1	2	

Table A.12: Rate of adjustment after a clock drift perturbation.

Cycle = cycle length jitter, Link1/Line2 = start time offset jitter for links 1/2
Column groups = bound on size of latency and clock drift perturbation

	100 ticks / 100 PPM			10 ticks / 100 PPM		
Node	Cycle	Link1	Link2	Cycle	Link1	Link2
1	0	94829		0	6768	
2	166	86933	86531	23	6541	6531
3	147	73981	73696	16	5668	5640
4	206	70117	69673	32	5258	5217
5	168	64554	64415	18	4086	4049
6	149	61668	61576	12	3199	3197
7	125	60267	60047	16	3408	3388
8	212	59083	59249	14	3123	3120
9	159	57853	57599	15	3249	3257
10	130	55260	55081	16	3436	3449
11	167	45810	45457	22	3405	3390
12	154	39002	38924	10	3101	3094
13	160	36034	36062	21	2780	2798
14	146	33623	33610	21	2855	2851
15	277	28882	28866	17	2467	2446
16	170	26352	26266	17	2663	2687
17	148	23960	23992	15	2948	2942
18	87	24054	24158	26	2300	2252
19	236	13148	13119	19	977	934
20	283	565		11	23	

Table A.13: Effect of both latency and clock drift perturbation on jitter values.

BIBLIOGRAPHY

[1] *Std. 1588-2002 IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems.*

[2] American National Standards Institute. *Synchronization Interface Standard.* http://www.ansi.org.

[3] E. Anceaume and I. Puaut. A taxonomy of clock synchronization algorithms. Technical Report PI1103, IRISA, July 1997.

[4] H. Anton. *Elementary Linear Algebra.* John Wiley & Sons Inc, New Jersey, 2005.

[5] J. Bacon. *Concurrent Systems.* Addison-Wesley, Harlow, 1997.

[6] P.A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[7] U. Black. *Data Communications And Distributed Networks.* Prentice Hall, New Jersey, 1993.

[8] U. Black. *Emerging Communications Technologies.* Prentice Hall, New Jersey, 1997.

[9] C. Bohm. *The DTM protocol: Design and Implementation.* PhD thesis, Royal Institute of Technology, February 1994.

[10] S. Bregni. Clock stability characterization and measurement in telecommunications. *IEEE Transactions on Instrumentation and Measurement*, 46(6), December 1997.

[11] D. Comer. *Network Systems Design using Network Processors.* Pearson Prentice Hall, New Jersey, 2004.

[12] F. Cristian. A probabilistic approach to distributed clock synchronization. In *Proc. Ninth IEEE International Conference on Distributed Computing Systems*, volume SE-11, pages 288–296, June 1989.

[13] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[14] W. Dijkstra and C.S. Scholten. Termination Detection for Diffusing Computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[15] J. Eidson and K. Lee. Sharing a common sense of time. *IEEE Instrumentation & Measurement Magazine*, 6(1):26–32, March 2003.

[16] A. Tanenbaum et al. *Distributed Systems.* Prentice Hall, New Jersey, 2002.

[17] C. Bohm et al. The dtm gigabit network. *Journal of High Speed Networks*, 3(3):109–126, 1994.

[18] D. Isaacson et al. *Markov chains: theory and applications.* J. Wiley & Sons, 1976.

[19] G. Bernstein et al. *Optical Network Control.* Addison-Wesley, Harlow, 2004.

[20] G. Coulouris et al. *Distributed Systems.* Addison-Wesley, Harlow, 2001.

[21] J. Eidson et al. IEEE-1588 Standard for a precision clock synchronization protocol for networked measurement and control systems. In *Proceedings of the 34th Annual Precise Time and Time Interval (PTTI) Systems and Applications Meeting*, December 2002.

[22] J. Eidson et al. Synchronizing measurement and control systems. *Sensors Magazine*, 19(11), 2002.

[23] J.G. Kemeny et al. *Finite mathematical structures.* Prentice-Hall, 1958.

[24] K.M. Chandy et al. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):143–156, May 1983.

[25] L. Lamport et al. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[26] R. Chow et al. *Distributed Operating Systems & Algorithms*. Addison-Wesley, Harlow, 1997.

[27] W. Goralski. *SONET: A guide to synchronous optical network.* McGraw-Hill, New York, 1997.

[28] R. Gusella and S. Zatti. TEMPO - A network time controller for a distributed Berkeley UNIX system. *IEEE Distributed Processing Technical Committee Newsletter*, 6(NoSI-2):7–15, June 1984.

[29] R. Gusella and S. Zatti. An election algorithm for a distributed clock synchronization program. Technical Report UCB/CSD-86-275, University of California at Berkeley, June 1985.

[30] R. Gusella and S. Zatti. The accuracy of the clock synchronization achieved by TEMPO in Berkeley Unix 4.3 BSD. *IEEE Transactions on Software Engineering*, 15(7):847–853, July 1989.

[31] M. Horauer. Hardware support for clock synchronization in distributed system. In *Supplement of the 2001 International Conference on Dependable Systems and Networks*, pages 10–13, July 2001.

[32] International Telecommunications Union. *ITU-T G.707: Network Node Interface for the Synchronous Digital Hierarchy (SDH).* http://www.itu.int.

[33] L. Kleinrock. *Queueing systems*, volume 1. J. Wiley & Sons, New York, 1975.

[34] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transaction on Computers*, C-36(8):933–939, August 1987.

[35] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[36] L. Lamport and P.M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[37] S. Lee. *A study of Cyclone technology.* PhD thesis, University of Maryland at College Park, 1998.

[38] Barbara Liskov. Practical uses of synchronized clocks in distributed systems. In *PODC*, pages 1–9, 1991.

[39] K. Marzullo and S. Owicki. Maintaining the time in a distributed system. *ACM Operating Systems Review*, 19(3):44–54, July 1985.

[40] D.L. Mills. On the accuracy and stability of clocks synchronized by the Network Time Protocol in the Internet system. *ACM Computer Communication Review*, 20(1):65–75, January 1990.

[41] D.L. Mills. Internet time synchronization: the Network Time Protocol. *IEEE Transactions Communications*, 39(10):1482–1493, October 1991.

[42] Bell Communication Research. *Digital synchronization network plan.* Technical Advisory TA-NPL-000436, November 1986.

[43] A. Silbershatz. Communication and Synchronization in Distributed Programs. *IEEE Transactions on Software Engineering*, SE-5:542–546, November 1989.

[44] T.K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

[45] Symmetricom. *Synchronizing telecommunications networks: Basic concepts.* http://www.symmetricom.com.

[46] A. Tanenbaum. *Distributed Operating Systems.* Prentice Hall, New Jersey, 1995.

[47] A. Tanenbaum. *Computer Networks.* Prentice Hall, New Jersey, 2003.