A Synchronous Communication System for a Software-Based Byzantine Fault Tolerant Computer

by

Reuben Marbell Sterling

B.S. Computer Science and Electrical Engineering
Massachusetts Institute of Technology

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Masters of Engineering in Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology

September, 2006

Signature of Author:

Department of Electrical Engineering and Computer Science
August 21, 2006

Certified by:

Roger Racine
Charles Stark Draper Laboratory
Thesis Supervisor

Certified by:

Barbara H. Liskov
Ford Professor of Engineering
Thesis Advisor

Accepted by:

Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

THIS PAGE INTENTIONALLY LEFT BLANK

A Synchronous Communication System for a Software-Based Byzantine Fault Tolerant Computer

by

Reuben Marbell Sterling

Submitted to the
Department of Electrical Engineering and Computer Science

August 21, 2006

in Partial Fulfillment of the Requirements for the Degree of
Masters of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes the redesign of a Byzantine-resilient, quad-redundant computer to remove proprietary hardware components. The basic architecture consists of four Commercial Off-The-Shelf (COTS) processors in a completely-connected network of point-to-point ethernet connections. In particular, the focus of this thesis is an algorithm that combines clock synchronization and communications between fault containment regions by inferring relative clock skew from the arrival time of expected messages. Both a failsafe and a fault-tolerant algorithm are discussed, though the fault-tolerant algorithm is not fully analyzed. The performance of a prototype and the failsafe synchronization algorithm are discussed.

Technical Supervisor: Roger Racine
Title: Principle Member Technical Staff

Thesis Advisor: Professor Barbara H. Liskov
Title: Ford Professor of Engineering, MIT Computer Science and Artificial Intelligence Laboratory

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgment

August 21, 2006

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under Internal Company Sponsored Research Project 20317-001, Software Based Fault Tolerant Computer.

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions herein. It is published for the exchange and stimulation of ideas.

_____

(Reuben Marbell Sterling)

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

1 Introduction **13**

1.1 Thesis Description 14

1.2 Advantages of Software-Based Fault Tolerant Computers 15

1.3 Organization 16

2 Background **17**

2.1 Basic Voting Architectures 17

2.2 Advanced Voting Architectures 18

2.3 Introduction to Byzantine Resilience 19

2.4 Introduction to X-38 Fault Tolerant Computer 23

2.5 Clock Synchronization 26

3 System Overview **29**

3.1 Physical Description 30

3.2 Software Design 34

3.2.1 Abstraction Layers 35

3.2.2 Communication Stack 37

3.2.3 Code Structure 43

4 Synchronization Protocol **49**

4.1 Motivation and Description 50
4.2 Definitions and Assumptions 53
4.3 Failstop Algorithm 56
4.3.1 Characterization of Algorithm for Two Nodes 60
4.3.2 Characterization of Algorithm for Three or More Nodes 92
4.4 Byzantine Resilient Algorithm 114

5 Experimental Application 135

6 Results and Conclusions 141

6.1 Prototype and Experimental Results 141
6.1.1 Current SBFTC Limitations 141
6.1.2 Synchronization Algorithm Performance 142
6.1.3 SBFTC Performance 145
6.1.4 Design Alternatives 148
6.2 Discussion and Recommendations 149
6.3 Summary and Future Work 151
6.4 Acknowledgements 153

List of Figures

2-1 Basic Voter 18

2-2 Advanced Voter 19

2-3 Potential X-38 Configuration 24

2-4 Actual X-38 Configuration 25

3-1 Ideal SBFTC Physical Layout 31

3-2 Actual SBFTC Physical Layout 32

3-3 Message Delivery through a Switch 33

3-4 X-38 vs. SBFTC Layout 34

3-5 Software Abstraction Layers 36

3-6 Communication Layers 37

3-7 Class 1 Message Steps 42

3-8 Class 2 Message Steps 43

4-1 Synchronization Timing Diagram 58

4-2 Failstop Synchronization Algorithm 59

4-3 Byzantine Synchronization Algorithm - Part 1 116

4-4 Byzantine Synchronization Algorithm - Part 2 117

5-1 Inverted Pendulum 136

5-2 Physical Layout of Experimental Application 138

5-3 Experimental Application Screen Shot 138
5-4 Demonstration Timing Diagram 139

6-1 D_{max} vs. Min network latency and max network latency 144
6-2 D_{max} vs. Min network latency and Latency Spread 144

Chapter 1

Introduction

The advantage of fault tolerant computer systems in critical applications is clear. Computers are a powerful and ubiquitous resource, and their reliability can dictate the success or failure of a system. In particular, applications involving human safety require computer reliability far greater than what is typically needed. In such important applications, fault tolerant computer systems are absolutely necessary to ensure the safety of the humans involved.

In a naively engineered system, reliability of the whole system is only as good as the most unreliable piece. Unfortunately, building basic components of a system to the standards required by critical applications is usually infeasible due to time, cost, or technological constraints. As a result, systems must be engineered with the low reliability of individual components in mind.

The reliability of a computer system depends on both the correctness of the software written for the system and the robustness of the hardware against malfunction, particularly when the system operates in extreme environments like Earth orbit and beyond. Both hardware and software have their own unique challenges when implementing highly-reliable systems. This thesis is concerned primarily with techniques to increase the reliability of

the hardware aspects of critical systems by running identical code on redundant computers.

1.1 Thesis Description

This thesis project spans the first year of a two year effort to redesign an existing fault tolerant computer to remove specialized hardware. The original fault tolerant computer, built by Draper Laboratory for NASA's X-38 vehicle, features four redundant computers that communicate via special-purpose hardware. The system also provides a complete API for user applications to take advantage of the fault tolerance features. The result of the current thesis work is a prototype fault tolerant computer running entirely on generic hardware, with the specialized communication protocols implemented entirely in software. Basic abstraction layers were built to support a rudimentary API, and an experimental application was developed to demonstrate the control of a simple inverted pendulum.

The work focused on redesigning the low-level synchronization, communication and voting protocols that support the fault tolerant properties of the system, since this is the functionality implemented at the hardware level in the X-38 computer. While the algorithm used to vote the inputs and outputs of the redundant computers remains largely identical to that implemented in the X-38 system, the synchronization and communication protocols are substantially different.

In addition to the implementation of a prototype system, a major goal of this thesis was to propose and develop a Byzantine resilient synchronization algorithm targeted at the specific requirements for this system. The algorithm runs on each node and determines how far to adjust that node's clock forward by inferring other nodes' clocks from the arrival times of messages. At each round of execution, the algorithm selects the node that it determines to be the most ahead and adjusts the local clock to match it as closely as possible. Unfortunately, due to time constraints, the proposed algorithm was not developed

sufficiently. Instead, the partial work completed toward the development of the Byzantine resilient algorithm is given in this thesis, and the continued development and comparison is left for future work.

1.2 Advantages of Software-Based Fault Tolerant Computers

Traditionally, fault tolerant systems are built using specially designed hardware. This specialized hardware might act as interfaces between redundant computers, or it may even be processors designed for fault tolerance at the CPU level via redundant circuitry. Hardware-level fault tolerance is motivated by performance requirements typically not achievable at a software level, but the development and low-volume fabrication costs of special hardware makes fault tolerance typically quite expensive. Fortunately, as processor speeds grow and communication latencies shrink, the opportunity to lift fault tolerance functionality from hardware to software becomes more realistic. The migration of fault tolerance support to software allows the use of generic hardware. As a result, fault tolerant computers will become cheaper to produce.

In addition, implementing fault tolerance support in software allows an unprecedented level of flexibility. For instance, if a hardware-based fault tolerant computer loses some specialized hardware due to a fault, the functionality of that piece of hardware may be lost for good. A fault tolerant computer in which specialized functions are implemented in software could migrate that functionality from one generic piece of hardware to another.

1.3 Organization

Chapter 2 presents a discussion of background information on the subject of fault tolerant computers and clock synchronization. Chapter 3 follows with a description of the design of the prototype software-based fault tolerant computer (SBFTC) developed as part of this thesis work.

Chapter 4 describes the partially-developed Byzantine resilient synchronization algorithm. It begins by describing the motivation for the development of the algorithm and then introduces the definitions and assumptions used in the design and analysis. A simpler, failstop (non-Byzantine) version of the algorithm is presented and analyzed as a introduction and guide to reasoning about the Byzantine resilient algorithm. Finally, the Byzantine resilient algorithm is introduced. The intuition behind the algorithm is described, and the beginnings of mathematical analysis are presented, but a full analysis is not yet available.

Chapter 5 describes the experimental application developed to demonstrate and test the prototype SBFTC, and, finally, Chapter 6 presents the experimental results from this thesis work and concludes with suggestions for future development.

Chapter 2

Background

This chapter provides background information necessary to understand the thesis topic before the work is described in greater detail. An introduction to fault tolerance, particularly Byzantine fault tolerance, is presented, followed by an introduction to the Draper Laboratory X-38 fault tolerant computer. A brief survey of former research in clock synchronization is given, along with additional relevant research.

2.1 Basic Voting Architectures

Techniques to increase the reliability of the hardware of critical systems typically use redundant computers, each executing the same code simultaneously, operating on the same inputs, and (ideally) producing the same outputs.

Figure 2-1 presents a logical diagram of such a system. Fault detection is done via voting. When the redundant computers produce output, such as an actuator command or status message, the outputs from all the redundant computers are compared by a voting element for discrepancies. If a discrepancy is detected, various actions may be taken, depending on the number of redundant outputs available for comparison. Assuming only a single er-

ror, if three or more redundant outputs are available and two are in agreement, the system may continue to function normally. If only two redundant outputs are available and they disagree, typically the system cannot continue and merely reports an error and ceases execution. This system behavior stands in contrast to a non-redundant system, which would continue unchecke[illegible]rror.

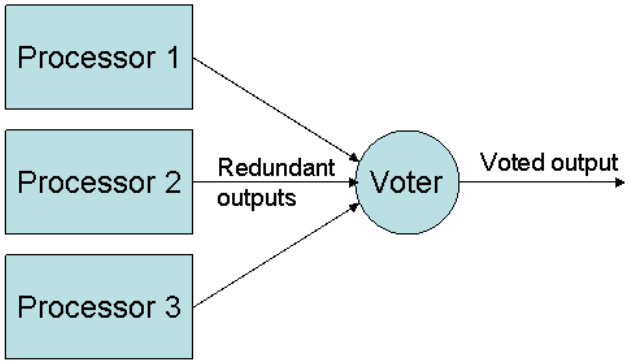

Figure 2-1: In the basic voter architecture, multiple computers execute the same code simultaneously. The redundant output created by the computers is voted to produce a single output masking any single error.

2.2 Advanced Voting Architectures

The voting system described above is not sufficient for many critical applications, particularly space operations, where radiation can corrupt any part of the system. Although faults that occur within the redundant computers would be handled correctly, consider the consequences of a failure of the voter element. While the voter may be considerably less complex than the redundant processors and less prone to error, the existence of a single point of failure excludes this design as a viable candidate for space applications.

Figure 2-2 shows an architecture that avoids single points of failure. Each node in the redundant system represents a fault containment region (FCR). Each FCR experiences

faults independently from every other. For example, an electrical short in one FCR is guaranteed not to affect any other FCR.

Every node is fully connected to every other via a bi-directional link. Messages can travel from one node to another without affecting communication in the opposite direction or between any other pair of nodes. While single failures may cause an FCR or a link to become unavailable, this architecture supports communication and voting protocols that are unaffected

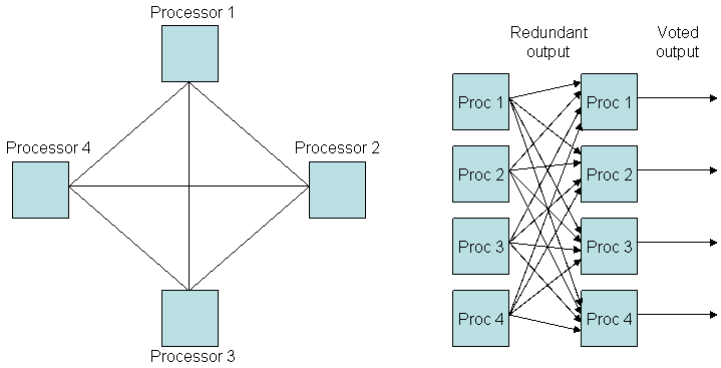

Figure 2-2: In the advanced voter architecture, processors broadcast their output to all other processors, which each vote the results.

2.3 Introduction to Byzantine Resilience

One important type of fault tolerance supported by the above architecture is known as Byzantine resilience. The term “Byzantine” is used as a synonym for “arbitrary.” It was originally coined in a paper by Lamport, et. al. [3], which described algorithms to reach consensus among cooperating parties – in their examples, Byzantine military generals – despite the existence of “traitors” that may fail to pass on messages, lie, or even conspire to break the consensus among the loyal generals.

The ability to tolerate arbitrary system faults, including lying and collaboration, might

seem like an excessive precaution. In many kinds of applications for which fault tolerant systems are required, like manned space exploration, the likelihood that the system may contain an active adversary is slight. Nonetheless, there are a few reasons why it might be preferable to design a system to be Byzantine resilient.

First, many types of failures that can occur naturally are similar to the types of attacks an active adversary might use to break the consistency of a system. For example, a node with failing hardware could easily transmit slightly different versions of a message to different recipients, introducing the same sort of confusion to the system an active adversary would want. Second, designing a system to handle *any* fault makes reasoning about the capabilities of the system much easier.

Lamport's paper proposes the following scenario. In a group of n generals, there exists a single commanding general and $n - 1$ lieutenant generals. The commanding general must send a message to his lieutenants such that the following requirement are satisfied:

1. All loyal generals obey the same order.
2. If the commander is loyal, every loyal lieutenant obeys the order he sends.
3. If the commander is not loyal (sends conflicting messages), every loyal lieutenant must agree on the same order (or agree to take no action).

In the architecture shown in figure 2-2, the incorrect execution of any single processor is considered a single fault. When a system is described as Byzantine resilient, it means that the system can tolerate some number of arbitrary faults and still function correctly. The number of tolerated failures depends on the number of nodes in the system, how completely the nodes are connected, whether the system is synchronous or asynchronous, and also on the types of messages being passed.

A synchronous system is one where the absence of a message from a particular node implies that the node is faulty. An asynchronous system makes no such assumption. The

Internet is an example of an asynchronous system. If a computer sends a message to another computer across the Internet, the communication medium offers no guarantees about the delivery time of that message or even that the message will arrive at all. A failed delivery does not implies that the sender is faulty. Systems with more predictable communication mediums, such as point-to-point connections, can support synchronous communication. If a node expects a message to arrive from another node by a particular time and it receives no message, the receiving node may assume the sender is faulty. Lamport et. al. assume synchronous communication.

Lamport et. al. describe two message types, “oral” and “written”. Intuitively, oral messages may be modified as they pass through nodes. Written messages, however, cannot be changed without detection and are always “signed” by the author. As a result, a recipient of a written message can know for certain who composed the message, or else he is guaranteed to notice a modification.

For synchronous systems using oral messages, Lamport et. al. show that in order to tolerate f faults, there must be at least $3f + 1$ total nodes in the system, i.e. $2f + 1$ loyal nodes. The system described in this thesis implements oral messages and a synchronous communication system.

A Simple Byzantine Resilient Protocol

Consider a system designed to handle f faults. There must be $n \geq 3f + 1$ nodes in the system. Each pair of nodes has a dedicated two-way link through which that pair can send messages between each other. Thus configuration allows a node to distinguish between messages from different nodes. The generalized algorithm requires f recursive iterations, but the specific case of $f = 1$, which only requires two communication rounds, is described here.

The steps of the algorithm are as follows:

1. The node with the message to transmit begins by broadcasting the message to all other nodes.

2. When a node receives the original message, the node stores it and re-broadcasts it to all other nodes.

3. When a node receives the re-broadcasted message, the node stores it.

4. After both communication rounds are complete, each node compares all the versions of the messages it received and accepts the majority version. If no majority is found, no message is accepted.

This algorithm satisfies the requirements listed above for Byzantine resilience. Consider requirement 2. Given that only one adversary may exist, if a commander sends the same message to all lieutenants, every node will receive the same message *at least* one more time. Since each lieutenant receives a maximum of three versions (from the commander and two other lieutenants), two matching messages comprise a majority, and the correct message is accepted.

Now consider requirement 3. A non-loyal commander is one who does not send the same original message to all three lieutenants. Fortunately, since the commander is dishonest, the three lieutenants are guaranteed to be honest. Let m_1, m_2 and m_3 be three possible messages the commander can send. Say the commander sends m_1 to two of the lieutenants and m_2 to the third. Then, in the second round, the first two lieutenants will receive another copy of m_1 from the other and the third will receive *two* copies of m_1 from the first two. Thus, all three lieutenants have received a majority of m_1 messages.

Suppose the disloyal commander sends m_1 to the first lieutenant, m_2 to the second, and m_3 to the third. This time, after round 2, none of the lieutenants will have received a

majority of *any* message, thus all the lieutenants will agree to not accept any message.

Requirement 1 follows from 2 and 3.

2.4 Introduction to X-38 Fault Tolerant Computer

The X-38 fault tolerant computer (FTC) is a Byzantine resilient, quad-redundant system on which this thesis is based [8]. Implemented at Charles Stark Draper Laboratory in 2002 for NASA's X-38 experimental aircraft, it provides a flexible architecture in which fault tolerant properties are provided by specialized hardware elements known as Network Elements (NE). In addition to the fault tolerant hardware architecture, the X-38 computer includes a large set of software libraries, which high-level software applications use to inherit the fault tolerant properties of the system. Since this thesis is primarily concerned with redesigning the specialized hardware as software, only the X-38's hardware architecture is described here.

Five NEs are linked in a fully connected network, and they perform the synchronization and I/O voting necessary for fault detection. The actual redundant processing is done by COTS processors that sit behind the NEs. Each redundant processor resides behind a different NE. A group of redundant processors is known as a virtual group. The quad-redundant virtual group requires processors residing behind four NEs, but the fifth NE still participates in communication/voting, allowing the system to tolerate two non-simultaneous faults. The first fault may be a Byzantine fault. After the system has recognized and recovered from the first fault, it is prepared to handle a second. Under certain conditions, the second fault may also be Byzantine, but under others, the system may only handle a subset of possible second faults.

The flexibility in the design is reflected by the ability to add an arbitrary number of COTS processors behind each NE, with the processors and the associated NE connected

through a VME backplane. As a result, multiple redundant computers can all exist simultaneously in the X-38 architecture, with each NE handling the communication for processors belonging to one or more redundant computers. A diagram of a potential hardware layout is given in figure 2

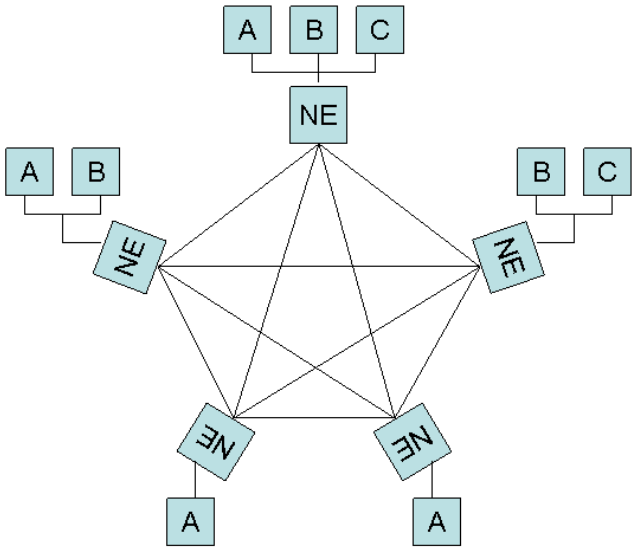

Figure 2-3: The X-38 is capable of supporting several redundant computers. Each processor of a redundant computer resides behind a different NE. In this example, processors marked by A are part of a quad-redundant computer, processors marked by B are part of a tri-redundant computer, and processors marked by C are part of a duplex computer. Simplex computers (not shown) are also supported.

The actual X-38 hardware layout contains only one quad-redundant computer and five simplex (non-redundant) computers. A diagram depicting the actual hardware layout of the X-38 computer is given in figure 2-4. The simplexes perform the roles of actuator and sensor control and communicate with the quad-redundant computer through the fault tolerant communication services provided by the NEs. Each quad-redundant processor was referred to as a flight-critical processor (FCP), and each simplex was known as an I/O Control Processor (ICP).

Communication between the NEs occurs at regular intervals and follows what is called a

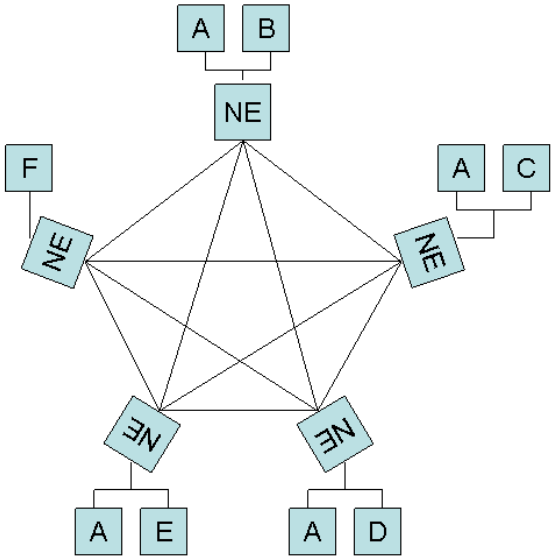

Figure 2-4: The actual X-38 configuration consists of a single quad-redundant computer (A) and five simplex computers (B, C, D, E, and F), which control actuators and sensors.

System Exchange Request Protocol (SERP). The basic feature of the SERP is an agreement step that takes place before each message exchange. During the agreement step, each pair of boards determines whether there is a message to send, whether the receiving board has space in its buffers to accept the message, and what class of message is to be sent. When an agreement is reached, the actual message is sent between the NEs. Once the appropriate voting steps among the NEs have been completed, the NEs deliver the message to the appropriate processor that resides behind them.

Since the NEs must know when to perform the SERP, a clock must be shared between NEs so they may communicate with the other NEs at the correct times. If no clock were shared, the various clocks used across the system would drift apart, eventually making the communication system useless. Unfortunately, NEs cannot directly share clock signals, since this would violate the integrity of the fault containment regions. Instead, clock synchronization is established through messages passed over the fault tolerant network.

2.5 Clock Synchronization

Much prior research has been published on the topic of fault-tolerant clock synchronization in distributed systems. One example of a distributed clock synchronization system is the Network Time Protocol (NTP),which is heavily in use throughout the modern Internet [7]. NTP is an extension of the distribute time service originally proposed by [6]. NTP is designed for a highly unpredictable environment, i.e. the Internet, where communication between computers can be delayed large and unpredictable lengths of time. NTP uses a large number of sources to determine the correct time, potentially down to milliseconds. The algorithm first determines a range of possible correct times based on a sampling of readings from a single source, with each sample taking a variable length trip across the network. Based on the intersection of a number of such ranges, the algorithm produces a good approximation of the true time. While NTP is very effective for Internet synchronization, the amount of overhead required makes it less appealing for embedded applications, where network latency can be much more predictable. Also, NTP is a "one-direction" algorithm where there is eventually a single, ultimate source of time, such as an atomic clock.

Algorithms for distributed, embedded systems commonly do not enjoy a connection to a wide area network, and thus cannot update their time from oracle sources. Instead, the nodes must cooperatively keep their time synchronized. Fortunately, such applications have much more predictable networks, which allows clocks to be kept quite synchronized, and many fault-tolerant algorithms have been developed for this purpose. The properties of these algorithms guarantee that if enough nodes are operating correctly, that all correct nodes in the system will adjust their clocks toward a consistent point in time and maintain synchronized clocks.

A general solution to the problem is given by [2]. This algorithm does not assume a completely connected network, but instead assumes that all correct nodes are fully con-

nected through some non-faulty path. The latency between any two correct nodes along the fault-free path is assumed to be bounded by some constant. The solution also requires the use of signatures. Like the algorithm presented in this thesis, neighboring clock times are inferred from the arrival times of messages, but since this solution does not assume a completely connected network, the achievable bound is not as favorable.

Another, more related solution is presented in [5]. This algorithm is very similar to the solution proposed here. Messages are sent from each node at a standard time, as measured by each node's logical clock, and every node waits long enough to receive the messages from every other correct node and the arrival time is recorded. Afterwards, an averaging function is applied to all the times at which the messages were received and the local clock is adjusted to the calculated average. The achievable synchronization is a function of the variation in network latency, i.e. minimum and maximum latencies. However, this algorithm potentially sets some clocks backwards which is not acceptable for the SBFTC.

The above algorithms strive to be optimal in terms of clock agreement between nodes. The algorithm in [9] provides synchronized clocks that are optimal in reference to accuracy, i.e. the departure from real time. The drift of the synchronized clocks from real time in this algorithm are only as much as the underlying physical clocks' maximum drift rate. However, the actual synchronization between nodes is not optimal. The maximum synchronization is a function of the maximum network latency, not of the variation in network latency. This algorithm follows the requirement that clocks may never be adjusted backwards.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

System Overview

The aim of this thesis work is to redesign the previously existing X-38 fault tolerant computer architecture to remove proprietary hardware components. Since the X-38 proprietary hardware's role is fault-tolerant communication between fault containment regions (FCR), i.e. a Network Element (NE) and its associated processors, the principal contribution of this thesis is to implement the fault-tolerant communication protocols as software running on COTS hardware. This style of computer can be referred to as a software-based fault tolerant computer (SBFTC).

An SBFTC offers some advantages over an FTC containing proprietary hardware. One clear advantage is cost. Producing a system requiring proprietary hardware is a costly endeavor, requiring greater development resources and low-volume fabrication of the specialized hardware. Of course, the high development and manufacturing costs imply a less competitive price on the market. A software-based solution promises faster development and lower-cost hardware, since it can be purchased from a third-party vendor, who presumably manufactures the product in higher quantities and can sell it cheaply.

Another advantage is flexibility. As more of a system's functionality is raised to the software level, the less dependent on any one piece of hardware the system becomes. If a

node fails due to a physical malfunction, a software process might be moved from executing on the failed hardware to another spare.

Traditionally, SBFTC's have been infeasible due to technology constraints, particularly CPU speed. Even with the advent of modern high-speed processors, radiation-hardened technology required for space operation tends to lag in performance. For example, while current high-end processor technology approaches clock speeds of several gigahertz, radiation-hardened computer clock speed remains in the range of hundreds of megahertz.

The SBFTC developed as part of the thesis work is a prototype intended to test the feasibility of a such a system with current technology. Since the SBFTC is intended to replace the original X-38 FTC, the design strives to achieve the specifications of the original system: two non-simultaneous faults, arbitrary numbers of virtual groups, and similar computation power and I/O throughput.

The system currently only handles a single fault, since it has only four nodes and implements only oral messages. For work on how to augment the SBFTC with written messages to support two faults, see [1].

3.1 Physical Description

This section describes two architectures. The first is the ideal fault-tolerant architecture toward which the work described in this thesis is targeted. The second is the actual physical architecture used in the experimental application of this thesis work. As will be seen, the actual physical architecture does not provide the properties of the ideal architecture and, therefore, is not well-suited for the application. Fortunately, the software written for the application hides the ill-suited architecture via an abstraction layer, so the high-level software described in this section and the synchronization algorithm described in Chapter 4 are written as if running on the ideal architecture. Unfortunately, the performance of the

system is adversely affected.

The physical layout of the ideal SBFTC consists of five boards, each with its own CPU. Four of the boards run both an NE and a flight-critical processor (FCP). The remaining board runs only an I/O Control Processor (ICP). The four NE/FCPs have independent, point-to-point connections. Messages sent on one link will not cause contention with messages on any other. The fifth ICP board only shares a connection with a single NE. If the ICP wishes to send a message to any other board, it must send the message first to its associated NE, which will forward it to the other nodes. Figure 3-1 depicts the ideal architecture of the SBF

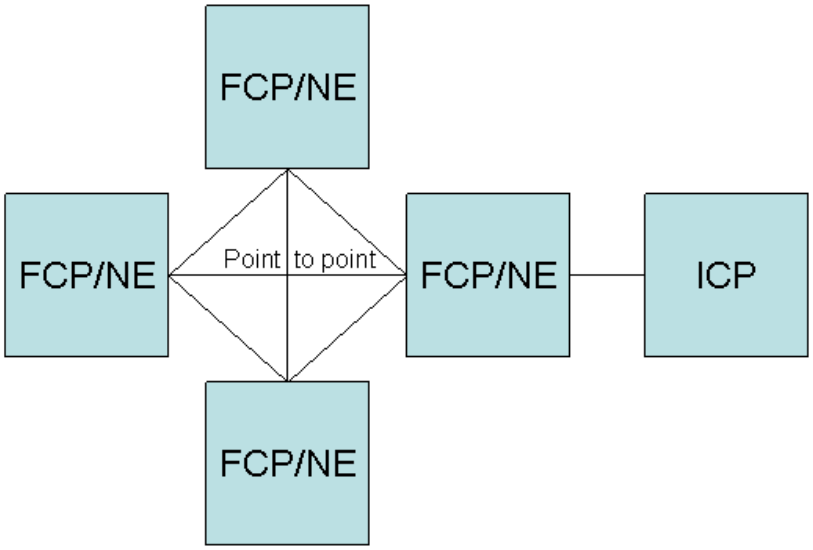

Figure 3-1: The ideal SBFTC would be composed of four nodes connected over a point-to-point communication medium. The fifth node, an ICP, may only communicate with its associated NE.

The actual physical layout of the SBFTC also consists of five boards, an ICP and four NE/FCPs, but all five boards are connected via a common ethernet. Each board, an Embedded Planet EP405, features a 300 MHz IBM PowerPC 4xx processor and a 100BaseTX ethernet adapter. Figure 3-2 depicts the actual physical layout of the SBFTC.

There are a few noteworthy departures from the ideal design. First, there is a common

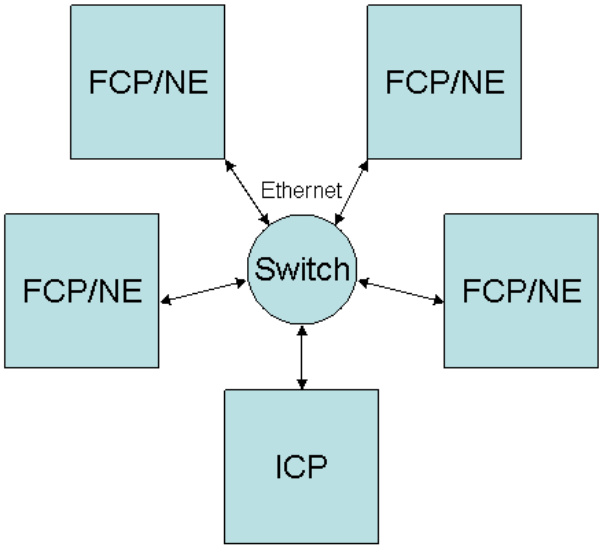

Figure 3-2: The actual SBFTC is composed of five boards with ethernet adapters and one ethernet switch, which provides a common network between all the boards.

ethernet switch connecting all the boards instead of a dedicated channel between each pair of boards. Clearly, a common switch for all inter-board communication introduces a single point-of-failure and should not exist in an ideal system.

Additionally, ethernet network contention raises the issue of increased, and even indefinite, network delays. The switch alleviates some of the problem. Typically, every board connected to an ethernet competes to send its message across a shared network. If two boards try to send a message at the same time, the boards detect the conflict, and each board waits a random amount of time before trying again. With so many nodes trying to send messages at the same time, the potential for large delays is high. In fact, theoretically, an ethernet does not guarantee that a message will ever reach its destination, since there is an exponentially small probability that nodes may continue to attempt sending at the same time forever. The switch solves this problem by accepting messages from multiple nodes simultaneously and storing the competing messages in a buffer, thus preventing the boards from ever conflicting and waiting. However, messages with the same destination

still need to be sent serially across the same wire. As a result, messages sent at the same time may take a long time to reach their destination. Fortunately, if there is a known upper bound on the number of messages in-flight at any time, there is a known upper bound on the maximum latency. Figure 3-3 illustrates how the switch handles messages.

Due to resource constraints, the common switch is the easiest way to simulate a completely connected point-to-point network. Fortunately, as long as the network can guarantee a maximum bound on latency, there is no theoretical limitation to the system capability other than decreas

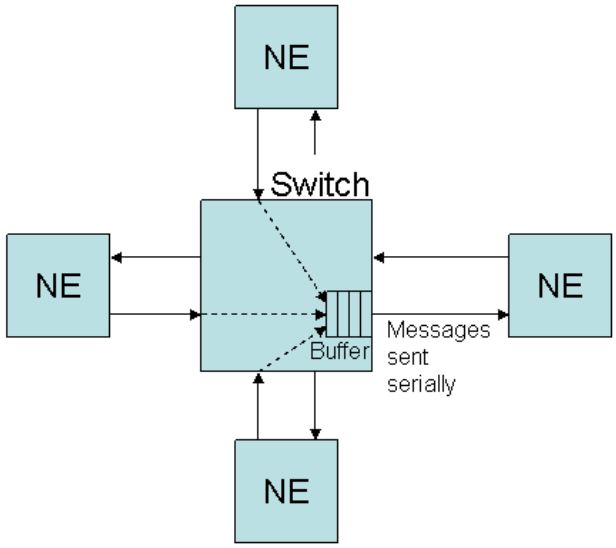

Figure 3-3: The common switch allows all boards to send at the same time and buffers the messages internally. Messages with a common destination must still be sent serially after they are buffered.

Another curious feature is that not all the nodes connected to the common network are NEs. Four of the nodes are NEs and the last is an ICP. Recall that in the X-38 FTC, ICPs and FCPs both reside behind NEs. In the SBFTC, this single ICP should also exist behind an NE and not have direct access to the common network, but, again, lack of access to the required hardware necessitated the given topology. Logically, the ICP functions as if it were located behind a single NE via software-layer abstraction.. It does not participate in any

communication protocols between NEs and only communicates with a single, designated NE, but it uses the common network to do so.

3.2 Software Design

Figure 3-4 depicts the hardware and software components of a single FCR of the X-38 FTC and the new SBFTC and demonstrates a few major differences between the two designs. First, and most importantly, the network element now exists as a software module running on a commercial off-the-shelf (COTS) board. Also, while the NE hardware in the X-38 FTC is dedicated entirely to performing NE functions, the COTS board in the SBFTC runs both t

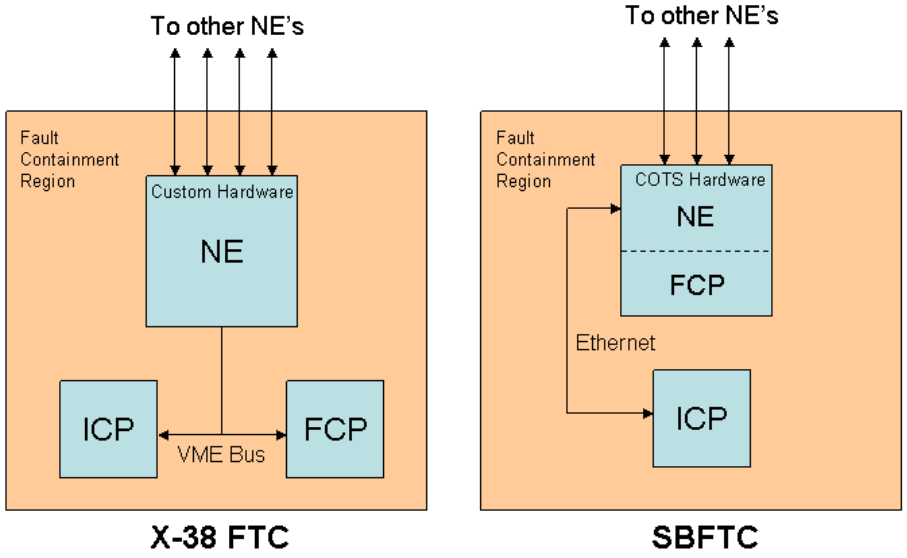

Figure 3-4: The physical layout of a single FCR in the X-38 FTC and the SBFTC. In a shift from the X-38 FTC design, the SBFTC combines the NE and the FCP as logical processors executing on a single board. The ICP still runs on a separate board.

Both FTCs use a real-time operating system (RTOS) to help ensure that critical tasks receive sufficient CPU resources at the appropriate times. The RTOS used for this project

is Green Hills Integrity. Integrity is marketed as a “time and space partitioned” RTOS. Its time partitioning helps support “hard” real-time constraints. A system with tasks that *must* be completed by given deadlines (or else the system fails) is said to have hard real-time constraints. In contrast, a system with more relaxed deadlines is said to have soft real-time constraints. Integrity’s time partitioning guarantees that software modules receive the amount of CPU time intended by the system designer regardless of the behavior of other modules running on the system. Its space partitioning guarantees that a software module cannot interact (or interfere) with other software modules running on the same board unless given explicit permission before runtime. As a result, despite accidental or malicious behavior of foreign software, the kernel and other software processes remain unaffected.

A time and space partitioned RTOS is particularly useful in the software design of the SBFTC, since it allows the NE to reliably coexist on-board with the FCP. Not only must the NE be protected from the FCP software, which contains application-dependent, foreign code, both the NE and the FCP must perform time-critical operations despite sharing the CPU with one another. A time-partitioned RTOS helps guarantee that the CPU is shared correctly and efficiently.

3.2.1 Abstraction Layers

Abstraction layers in the SBFTC hide low-level implementation details of the system from higher-level software components. This allows cleaner, clearer code, and it also will allow the non-ideal physical layout of the system to be changed in the future without modifying the higher-level code. The SBFTC implementation defines a few layers of abstraction, depicted by figure 3-5.

The lowest layer of abstraction in the SBFTC is the board layer. A board is the basic

unit of hardware, and there are always a fixed number of boards in the system. Code written at this layer is aware of the physical layout of the system.

The virtual processor (VP) layer sits above the board layer. The VP layer defines three processor types: NEs, FCPs and ICPs. One or more VPs may exist on a single board. The VP layer keeps track of which processors are running on which physical elements. In the current prototype, there are a fixed number of NEs, FCPs and ICPs in the system, and the particular boards where they exist are set before run-time and remain static throughout the lifetime of the system.

The virtual group (VG) layer is built on top of the processor layer. A VG is a set of one or more VPs, each sitting behind a different NE, that act as a single, logical computer. Each member of a VG executes the exact same software, operates on exactly the same inputs, and produces, under error free conditions, exactly the same output. Several different VG's may exist in the SBFTC at the same time, and different VG's may contain different numbers of members. This software layer maintains knowledge of which processors comprise which groups. In the prototype, VGs and their members are set before run-time and remain static throughout the lifetime of the system. Ideally, the system should be capable of assembling VGs upon startup according to available resources and reconfiguring as resources change during execution. Such an extension is left for future development.

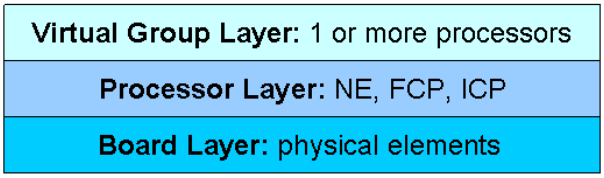

Figure 3-5: The SBFTC software implements several abstraction layers, the hierarchy of which is depicted here. The board layer recognizes individual, fixed physical elements is the system. The virtual processor layer recognizes processors running on top of the physical elements. The virtual group layer combines multiple processors into redundant, fault tolerant computers.

3.2.2 Communication Stack

Each abstraction layer described in Section 3.2 has an associated communication layer. The communication stack defined for the SBFTC sits atop the layers defined by the Open Source Interconnect (OSI) model (physical, link, network, and transport). Each communication layer provides an API to the layer above, which uses the API to route messages to the corresponding layer on a destination node. The following sections describe the relationships b[illegible]re 3-6.

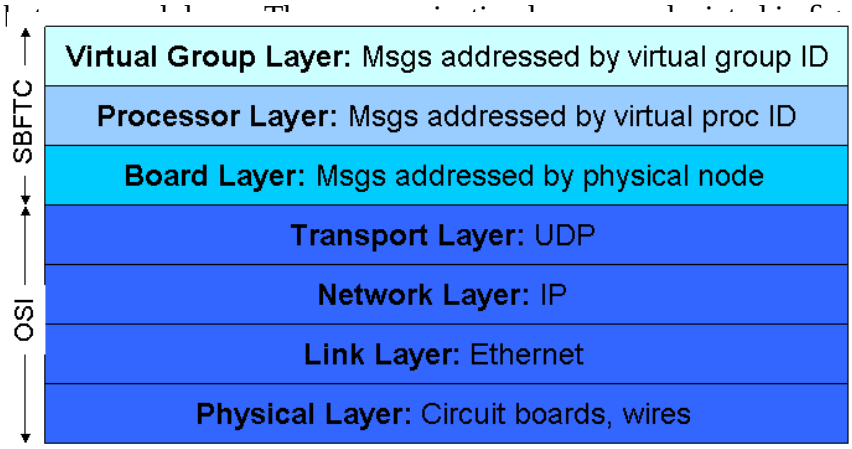

Figure 3-6: Each SBFTC abstraction layers defines an associated communication medium, the hierarchy of which is depicted here. The SBFTC layers are built above the Transport OSI layer. Messages between virtual groups are addressed by virtual group ID. Virtual group messages are turned into processor layer messages and passed to the processor layer. The processor layer maintains a list of which virtual processors are running on which physical nodes, and passes appropriately addressed packets to the board layer. Each layer may require some message processing at each hop to determine if the destination has been reached yet.

Board Communication

Boards are the lowest abstraction layer defined by the SBFTC. Messages between boards are passed via UDP packets, which are sent over a switched 100 Mbps ethernet network. Each board contains a standard ethernet adapter with a unique ethernet address and unique

IP address.

Ideally, each NE would have a dedicated point-to-point connection to every other node. Such a design is simulated by connecting each NE through a shared ethernet switch. The switch eliminates wire contention between the ethernet adapters at the multiple nodes, thereby reducing the average transport latency across the network.

In an ideal system, dedicated point-to-point connections would be achieved with multiple communication ports on each board (not necessarily ethernet), with each port connected directly to that of another board. If this were the case, it would be the responsibility of the board communication layer to route messages destined for a particular board through the correct communication port.

In the interests of prototype performance, it should be noted that circumventing the transport and network layers entirely and using the ethernet link layer directly may be desirable, however such optimizations are beyond the scope of the thesis.

Virtual Processor Communication

NEs, FCPs and ICPs pass messages via a communication layer that sits above the board communication layer. If the board communication layer is analogous to the OSI model's physical layer, the VP communication layer is analogous to the OSI link layer. It provides a protocol to send messages across the board layer to an "adjacent" VP.

FCPs and ICPs are considered to be adjacent to their associated NEs, and NEs are adjacent to each other. This communication layer cannot be used to pass messages between FCPs and ICPs. Such communication is required to be fault tolerant, and it is handled via virtual groups at the next communication layer.

Each NE is assigned an ID that is unique to the system, and each FCP or ICP is assigned an ID that is unique to its associated NE. The IDs are assigned before run-time and remain constant throughout the lifetime of the system. The processor layer on each NE stores a

table mapping VP IDs to board IDs. When a message is sent using this layer, the board ID is fetched from the table, and the message is passed down the communication stack to the board layer, which sends the message over UDP to the correct board.

A message originating from an FCP or an ICP need not include an address since it may only send messages to its NE. A message originating from an NE may be addressed to one or more NEs or to a single VP. NEs may address messages to themselves.

In the current prototype, messages must be a fixed length, although this limitation was primarily imposed for ease of implementation and analysis. Although there is no fundamental minimum limit to the length, the nature of the communication network requires that a maximum message length be imposed for timing reasons.

Since multiple messages may need to be sent from one NE to another during the same interval, part of the NE/NE communication routine's job is to merge these messages into a single buffer when transmitting and to split the messages when receiving.

NE/NE messages are exchanged regularly at fixed intervals, and all NEs send their messages at as close to the same time as possible. Small differences in send times are due to discrepancies in each board's internal clock and fundamental limits of synchronization.

The strategy of sending all messages at the same time is advantageous in a few ways. First, it eases analysis of the system and allows an easy bound to be determined for the latency of messages across the network. Also, it provides the opportunity for clock synchronization based on inferring clock differences between boards via message arrival times. The clock synchronization algorithm used in the SBFTC is presented in Section 4.

FCP/ICP to NE messages can be sent asynchronously. NE to FCP/ICP messages are sent by the NE at regular intervals following NE/NE message exchanges in order to deliver incoming messages.

Virtual Group Communication

The virtual group (VG) communication layer is analogous to the OSI network layer, which sits above the link layer and provides communication between non-adjacent nodes. The VG communication layer provides communication between non-adjacent VPs.

Each VG is associated with a system-wide unique ID. In the prototype SBFTC, VG's and their members are hard-coded into the system. Ideally, the system should be capable of assembling VG's upon startup and reconfiguring during runtime according to available resources. Such an extension is left for future development.

VGs provide the basic level of Byzantine fault-tolerance in the SBFTC. As discussed in Section 3.2, a VG is one or more VPs all running the exact same code simultaneously, operating on the same inputs and producing the same outputs. This communication layer enforces that communication occur between entire VGs instead of between single VPs. Any message sent from one VG to another will be delivered reliably, provided that the system currently is suffering at most one fault.

The guarantee of reliable delivery requires:

1. The output of all members of a VG are correctly voted to mask errors in any one of the members of the VG.

2. The same input is delivered to all members of the recipient VG.

To achieve these requirements, the SBFTC uses a broadcast and reflect algorithm similar to that discussed in Section 2.3. There are two kinds of messages that are sent over the VG communication layer, known as class 1 and class 2 messages. Class 1 messages are also known as single-source messages and are used when a single-member VG (an ICP) sends a message to a multi-member VG (an FCP). An example of such a message is an input sensor sending data to a set of FCPs. Class 2 messages are sent when a multi-member

VG sends a message to another VG of any size, though typically an ICP. An example of such a message is a set of redundant FCPs belonging to the same VG sending a control command to an ICP controlling a flap or an engine valve. ICPs may not send messages to other ICPs, so no message class is defined to handle such cases.

Class 1 Messages

Figure 3-7 shows the path of a class 1 message. A class 1 message makes four hops in its journey from an ICP to multiple FCPs.

1. The message travels from the ICP to its associated NE.
2. The NE recognizes that the message originated from an ICP, initiates a class 1 message exchange by sending a copy of the message to all other NEs in the system.
3. Each receiving NE responds by reflecting its copy of the class 1 message to all other NEs.
4. Each NE determines the correct message by comparing its multiple copies. Each NE then checks whether it is responsible for a member of the destination VG indicated is the message. If so, the NE forwards the message to the appropriate FCP.

At the end of the exchange, each FCP receives the exact same message at approximately the same time.

Note that all NEs vote their multiple message copies regardless of whether they are responsible for an FCP on not. This way, in the event of an error, each NE reaches a decision on who the guilty party might be.

Class 2 Messages

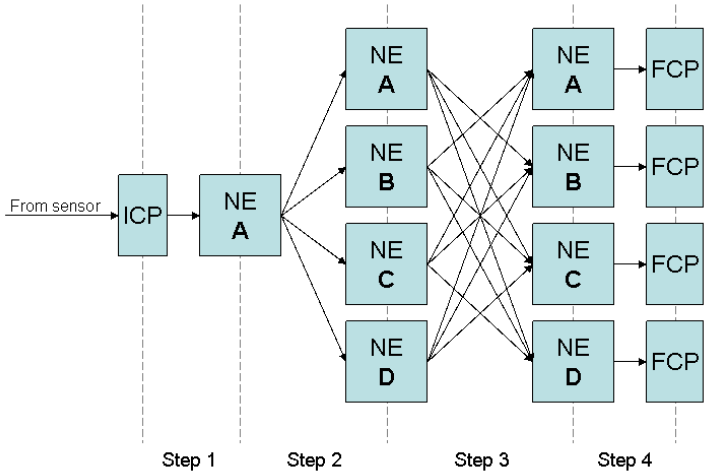

Figure 3-7: A class 1 message requires four steps to travel from an ICP to an FCP. The message is first sent from the ICP to the adjacent NE. Next, the NE initiates a two-step, Byzantine fault tolerant message exchange for the message to reach the other NEs. When the multiple copies of the message have been received by the NEs, each NE votes the message, then forwards it to its adjacent FCP.

Figure 3-8 shows the path of a class 2 message from a set of FCPs to an ICP. A class 2 message requires three hops en route to its destination.

1. The message travels from the FCP to its associated NE.

2. The NE recognizes that the message originated from an FCP, then initiates a class 2 message exchange by sending a copy of the message to all other NEs in the system.

3. Each receiving NE now has a version of the FCP output from all NEs and performs two steps in parallel

   (a) Each NE checks whether it is responsible for a member of the destination VG indicated is the message. If so, the NE determines the correct message by comparing its multiple copies, then forwards the message to the appropriate ICP.

   (b) Each NE also reflects the multiple versions of the FCP output to all other NEs so that, it the event of an inconsistency, all NEs can determine the guilty node.

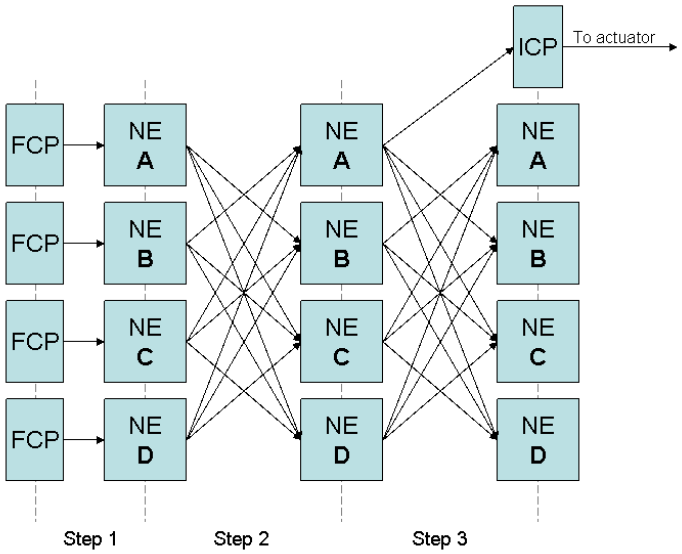

Figure 3-8: A class 2 message requires three steps to travel from an FCP to an ICP. The message is first sent from the FCP to the adjacent NE. Next, the NE sends a copy of the message to all other NEs. Step three involves two parallel operations. First, each NE checks to see if it is responsible for the destination ICP. If it is, it forwards the voted message to that ICP. At the same time, each NE reflects the messages it received in step two to all other NEs. This way, if any of the FCPs send inconsistent output, all the NEs may agree on the guilty party.

3.2.3 Code Structure

The source code is divided into major components: communication libraries, initialization routine, and communication loop. These three parts rely heavily on two third-party libraries, Integrity's OS API and Interpeak's IPCOM/IPLITE lightweight network stack. The Integrity API is particularly important for running multiple tasks and scheduling the execution of those tasks via timers and "alarms." The Interpeak network stack provides a BSD-compliant API for IP/UDP functionality designed for embedded applications.

The communication libraries define the board and virtual process abstraction layers and associated APIs. They allow an NE to send and receive messages to its associated FCP and ICP and to other NEs without having specific knowledge on which physical boards those processors are executing. Currently, the mapping between VPs and physical nodes is hand-

coded into the libraries, but further development would allow those tables to be initialized at startup.

The libraries provide a send/receive mechanism based on fixed-size packets, and allow applications to wait on the arrival of packets using a Unix select-like interface. A user defines a set of NEs and/or FCP/ICPs to wait for a specified (or indeterminate) length of time. The operation will block until a message arrives for any of the defined processors or until the specified time elapses. If the wait operation indicates that a particular processor has messages waiting, a read operation for that processor is guaranteed not to block.

The initialization routine is primarily an implementation of the non-authenticated "Optimal Clock Synchronization" algorithm described in [3]. Its job is to ensure the clocks of the nodes are sufficiently synchronized before the synchronization algorithm described in this thesis takes over maintenance. The implementation consists of two tasks. One task's responsibility is to keep track of time and broadcast "init" packets at the start of each synchronization round. The other task waits for incoming packets, which may arrive before and after what the node believes is the start of the synchronization round. When a packet arrives, the task takes appropriate action, i.e., recording the packet's arrival, sending out an echo packet when appropriate, or adjusting the system clock forward.

Once the initialization routine has completed, control is passed to the communication loop, which manages both the VP and VG communication layers. The communication loop is the heart of the NE, which functions as a fault-tolerant bridge between FCPs and ICPs, maintains synchronization, and controls message sending, receiving and voting. The loop runs as a single thread, performing each responsibility in sequence and repeating every fixed amount of time.

The communication loop repeats the following steps:

1. Prepare and send messages

2. Wait to receive all messages

3. Process and vote appropriate messages

4. Send and receive messages from FCP/ICP

At the beginning of a communication round, the communication loop wakes up, examines its active message table to determine which messages must be sent to other NEs, prepares a buffer containing the messages specific for each NE, and sends them all. The contents of the buffers to each NE are not necessarily the same. For instance, a NE does not need to reflect a message back to the node from which it received the message originally.

After sending, each node waits a pre-specified amount of time to allow messages from trailing nodes to arrive. During this time, the processor is freed for other work to be performed by the CPU, such as executing FCP code.

When the waiting time has elapsed, the NE then checks all the incoming messages for header corruption and adds the new message information to the appropriate locations in the active message table. For example, if the NE receives multiple messages from different NEs containing different versions of the same class 1 message, those versions will be grouped together in the message table. When enough message versions have been received from the other NEs, the versions are compared for inconsistencies. If one version contains errors, they are out-voted by the consistent messages, and thus the errors will not be propagated to the destination FCP or ICP.

The final task of the NE's communication loop is to handle messages to and from its associated FCP/ICPs. If any voted messages are intended for an FCP or ICP for which the NE is responsible, the NE then forwards the message. If any messages are available from the FCP/ICP to be received by the NE, the NE reads them and adds them to the message table in preparation for the next communication cycle. The NE then sleeps until the next cycle, allowing other processes on the node to receive CPU time.

Message Table

The description of the communication loop mentioned the active message table. Each NE maintains a table of active messages is the system. An active message is one that has been sent by an FCP or ICP, but has not yet been delivered to the recipient FCP or ICP. A message is created in the NE's table when it is received from an FCP/ICP or when it is received from another NE during a first-round broadcast for a class 1 or class 2 message. The message is deleted when it either delivered to a recipient FCP/ICP or when the NE recognizes it is not responsible for the destination processor. The table may be thought of as part of the Virtual Group layer, since the table keeps track of related class 1 and class 2 messages for voting purposes.

Other Capabilities

In addition to the SBFTC functionality, the system also includes meta-capabilities to assist in development, debugging, and demonstration. First, the system provides a logging capability that allows debugging text to be stored to a memory buffer rather than a debugging console. This allows debugging and status information to be collected without significantly impacting the timing of the code, particularly the communication loop, and then replayed at the developer's convenience. Text printed directly to the console significantly delays the execution of code, which causes unpredictable behavior between nodes expecting messages within time windows.

In order to support delayed log printing and other on-demand services, an administration interface provides the ability for a user to send UDP packets to specific nodes that instruct them to perform actions like dumping the debug log. If a task wishes to accept external commands, it may register a callback function and a command header such as

“LOG”. The administrator interface then listens on a dedicated port for incoming packets. If a packet arrives in which contents follow the pattern “LOG *”, the specified callback function is invoked with the packet’s contents as an argument.

The administration interface is also used to allow a user to inject faults into the system at will for testing and demonstration purposes.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

Synchronization Protocol

Each of the Network Elements (NE) in the SBFTC must maintain its internal clock at approximately the same time to ensure proper functioning of the system. Periodic adjustments of the local clocks are necessary, since all clocks drift relative to one another, and eventually the skew will prevent the boards from behaving in a consistent, predictable manner. The primary reason for keeping the clocks on each board synchronized is to ensure predictability of the communication system and to ease the implementation of the fault tolerant network.

In many multi-node applications, it is often possible and convenient to directly wire all the nodes with the same electrical clock signal. However, in a highly available system, using a common clock signal is impossible because it introduces a single point of failure. If the clock faults, the entire system goes down.

Fault tolerant solutions using custom hardware are able to solve the problem of synchronization by comparing redundant clock signals emanating from each node and adjusting them via specialized hardware. An SBFTC running on COTS hardware cannot rely on such a solution, since low-cost hardware does not include mechanisms for reliable electrical clock synchronization. Instead, each node must maintain a synchronized logical clock

that should keep approximately the same time as the logical clocks on all other nodes. A node maintains its logical clock through two values, an absolute time and a delta. The absolute time is incremented according to the hardware clock. The delta may be adjusted programmatically. The logical clock time is calculated by summing the absolute value and the delta.

Operating systems provide such a clock. However, the challenge exists in adjusting the delta component on all nodes correctly. Since no hardware support exists to synchronize logical clocks, the software is responsible for determining when clock adjustments are necessary by passing messages over a standard communication medium.

The following sections describe a logical clock synchronization algorithm developed for the SBFTC as part of this thesis work. The first section introduces the concepts and intuition behind the algorithm. The next section describes the notation used in this section as well as the assumptions the algorithm depends on. The third section presents a failstop (non-Byzantine resilient) algorithm. The fourth section extends the algorithm for Byzantine resilience. The Byzantine resilient algorithm is introduced, but a complete mathematical analysis has not been completed.

4.1 Motivation and Description

The fault-tolerant network is a bottleneck resource in the SBFTC. In order to make behavior of the system predictable and useful, the SBFTC must guarantee a user application that any data sent across the network will have a *maximum* latency across the network. If that guarantee is violated, time-critical operations may cease to function correctly.

The smallest maximum latency the SBFTC can promise to user applications is the latency of a standard message across the network *plus* the maximum time that message may have to wait for other, higher priority operations to finish using the network. If the sys-

tem must reserve the network periodically for system-level operations like clock synchronization, applications can never be guaranteed latencies less than the time that operation consumes.

Given the importance of reducing required network overhead, the intent of this algorithm is to allow the nodes of the system to maintain a synchronized logical clock while reserving the network for the smallest time intervals possible. While the Byzantine resilient algorithm actually requires a large amount of data to be exchanged between nodes, it allows individual communication rounds to be separated by time, allowing critical messages to be sent in between.

Another motivation in the design of these algorithms is the requirement that clocks never be set backward. In real-time systems, where tasks are scheduled via the system clock and messages may be timestamped, adjusting a clock backwards can result in inconsistent operation unless great care is taken. These algorithms avoid that pitfall by always setting clocks forward or not at all.

Defining Clock Synchronization

Synchronization does not imply that the logical clock is exactly the same on every node at all times. Such perfect synchronization is not possible at the software level. This is not to say that the logical clocks cannot be perfectly synchronized at some *instant*, but maintaining constant perfect synchronization over a time interval is not possible without the high granularity of control possible at the hardware level.

Synchronization of a set of logical clocks is instead defined in a more relaxed way. The logical clocks of a set of k correctly functioning nodes $N = \{n_1, n_2, ..., n_k\}$ are considered

synchronized over a time interval from t_0 to t_1 if

$$\exists\, D_{max} \text{ s.t. } \forall t \in [t_0, t_1],\ \forall n_i, n_j \in N,\ \left|C_i(t) - C_j(t)\right| \leq D_{max}\ ,$$

where $C_i(t)$ is the value of node i's logical clock at real time t and D_{max} is a constant.

In other words, the logical clocks of the nodes are considered synchronized if the difference between any two logical clocks is at most D_{max}. This relaxed definition of synchronization is necessary because it is impossible to achieve perfect synchronization.

Equally important, the relaxed constraint allows time to pass and clocks to drift between synchronization rounds. To describe this notion more precisely, begin with the following definitions:

1. Let A_{sync} be a synchronization algorithm that runs periodically.
2. Let the drift rate between the physical clocks of two nodes be bounded by a constant dr.
3. Let D_{max} be the minimum synchronization (i.e. a maximum difference between logical clocks) required for an application.
4. Let $D_{sync} \leq D_{max}$ be the minimum synchronization achieved by A_{sync}.

Then, the amount of time $dt_{between}$ allowed between points of maximum synchronizations is bounded by

$$dt_{between} \leq \frac{D_{max} - D_{sync}}{dr} - \epsilon,$$

where ϵ is a constant. The role of ϵ is to leave opportunity for mid-execution clock adjustments that may temporarily increase clock skew more than natural drift would.

The algorithms presented below run at well-defined intervals as described above. At the appropriate time according to its logical clock, each node sends a message to every other

node, and each receiving node takes note of the time that each message arrives (according to its own logical clock). Then, given that

- All nodes send their messages at what they believe is the correct time,
- The network has a known minimum and maximum latency,

each node can infer the local logical clock value at each of the other nodes and update its clock appropriately.

As will be shown in the following sections, the achievable D_{sync} value of the synchronization algorithms depends on properties of the network and the drift rate between nodes.

4.2 Definitions and Assumptions

This section presents definitions and assumptions used in the characterization of the synchronization algorithms presented in the following sections.

Node: An independent processing unit with an on-board timekeeper and a method of communication with all other nodes.

Wall clock: A timekeeper that maintains an absolute reference time. This time is not available to the nodes, but as an observer, one is aware of it.

Local clock: A timekeeper belonging to each node. Each local clock keeps time independently from the others. Different local clocks may read different times at the same wall clock time.

Clock drift: The tendency for local clocks on different nodes to run at slightly different rates.

Local clock drift from wall time is assumed to be bounded by a known constant $\boldsymbol{\rho} > 0$. Let $\boldsymbol{C_i(t)}$ be the local clock reading of node i at time t. Then,

$$(1+\rho)^{-1}(t_2 - t_1) \leq C_i(t_2) - C_i(t_1) \leq (1+\rho)(t_2 - t_1). \tag{4.1}$$

The above may be interpreted as, given two times, t_1 and t_2, the amount of logical clock time that could elapse on a node during that period is bounded by linear envelope.

The above may also be rewritten as

$$(1+\rho)^{-1}\left(C_i(t_2) - C_i(t_1)\right) \leq t_2 - t_1 \leq (1+\rho)\left(C_i(t_2) - C_i(t_1)\right). \tag{4.2}$$

Similarly, this may be read as, given two logical clock times, $C_i(t_1)$ and $C_i(t_2)$, the amount of real time that could elapse during that period is bounded by a linear envelope.

Note that the local clock drift rate between any two nodes is bounded by

$$dr = (1+\rho) - (1+\rho)^{-1} = \frac{\rho(2+\rho)}{1+\rho}.$$

It should be noted that, intuitively, the lower drift bounds might more correctly be $(1-\rho)(t_2 - t_1) \leq C_i(t_2) - C_i(t_1)$. However, the type of bound given above is more convenient, and is consistent with other literature, including [9].

Let wall times be denoted by a lowercase t and local clock times be denoted by an uppercase T.

Let $\boldsymbol{C_i(t)}$ be the value of the local clock of node i at time t.

Let $\boldsymbol{D_i(t)}$ be the difference between the local clock of node i and time t at time t. In other words,

$$C_i(t) = t + D_i(t).$$

Let $\boldsymbol{D_{ij}(t)}$ be the difference between the local clocks of nodes i and j at time t. This can be expressed as

$$D_{ij}(t) = D_i(t) - D_j(t) = C_i(t) - C_j(t).$$

Let $\boldsymbol{D_{max}}$ denote the maximum value of $D_{ij}(t)$ acceptable for an application. More precisely,

$$\forall \text{ nodes } i, j, \forall \text{ times } t, \left|D_{ij}(t)\right| \leq D_{max}.$$

Let $\boldsymbol{\left[D_{ij}\right]_i}$ be the difference between the local clocks on nodes i and j, *as calculated by node i.* Note that $\left[D_{ij}\right]_i$ is not a function of t. It represents a single value calculated by node i.

Let $\boldsymbol{T_{send}}$ be the local time at which all nodes are scheduled to send a message to every other node.

Let $\boldsymbol{t_{send_i}}$ be the time at which node i actually sends its messages to the other nodes. Note that T_{send} may correspond to different t_{send_i}'s for different nodes. Note that

$$C_i(t_{send_i}) = T_{send}.$$

Let $\boldsymbol{m_{ij}}$ represent a message sent from node i to node j.

Let $\boldsymbol{t_{m_{ij}}}$ be the time at which m_{ij} is received by node j.

Let $\boldsymbol{T_{m_{ij}}}$ be the local time on node j at which m_{ij} was received by node j.

Let $\boldsymbol{\ell_{m_{ij}}}$ be the time it takes for m_{ij} to travel from node i to node j. Note the relationship

$$t_{m_{ij}} = t_{send_i} + \ell_{m_{ij}}.$$

,

The communication network is assumed to deliver a message in a bounded window of time. Thus,

Let $\boldsymbol{\ell_{max}}$ be the maximum possible value of any $\ell_{m_{ij}}$.

Let $\boldsymbol{\ell_{min}}$ be the minimum possible value of any $\ell_{m_{ij}}$. In other words,

$$\forall \text{ nodes } i, j, \quad \ell_{min} \leq \ell_{m_{ij}} \leq \ell_{max}.$$

Let $\boldsymbol{\ell_{assume}}$ represent a constant such that $\ell_{min} \leq \ell_{assume} \leq \ell_{max}$. It may be interpreted as a best guess for $\ell_{m_{ij}}$. Its optimal value will be calculated as part of the analysis.

Finally, correct functioning of this algorithm requires that all local clocks are initialized such that they are synchronized to within some known value. No initialization algorithm based on the discussed algorithm is provided in this paper. It is left for future work.

4.3 Failstop Algorithm

This section presents the characterization of the non-Byzantine resilient synchronization algorithm. First, the algorithm is analyzed for two nodes. Then the analysis is extended to include three nodes.

The algorithm runs symmetrically on all participating nodes. Algorithm 4.3.1 defines the steps of a single round of the failstop algorithm. The algorithm may be broken into two processes:

1. **Receive:** The node maintains a process at all times that timestamps packets when they arrive.

2. **Send, Wait, and Adjust:** Upon a predetermined time, T_{send}, the node sends a message to all other nodes. The node then spends a predetermined length of time τ_{wait} waiting for the arrival of messages from other, slower nodes. From the arrival times of each message, the node estimates the largest difference between its local clock and other local clocks. If the node estimates that its own local clock is ahead of all others, it does not adjust its local clock. If it estimates that its local clock is behind the clocks of at least one other node, it adjusts its own clock to match that of the node estimated to be most ahead.

In the next round of synchronization, T_{send} takes on another value agreed upon by all participating nodes. Typically, the value of T_{send} at round i, $T_{send}^{(i)}$, is defined by $T_{send}^{(i)} = T_{send}^{(0)} + iP$, where P is some constant.

Figure 4-1 depicts a timing diagram of the interaction between two nodes executing algorithm 4.3.1 between which clocks are initially skewed by an amount Δ. Note that although each node sends a message at what it believes is T_{send}, the messages are actually sent at different times.

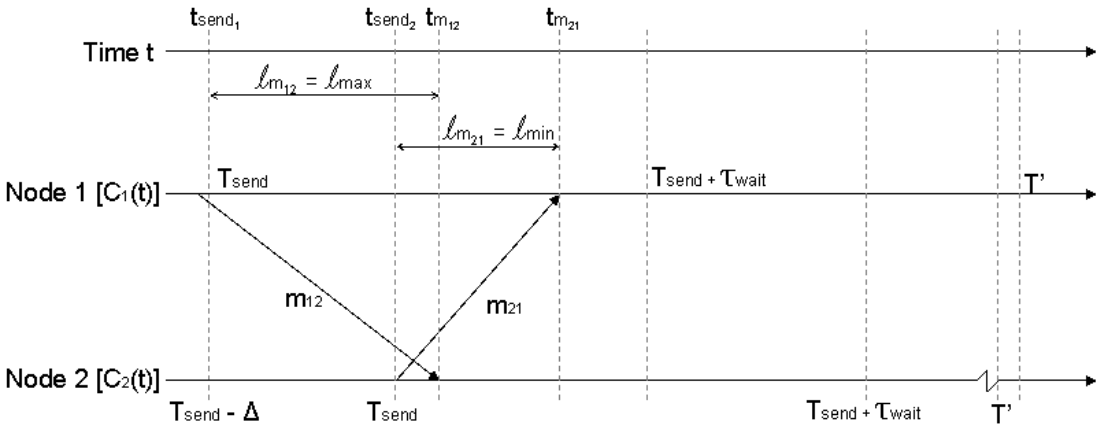

Figure 4-1: An example timing diagram of the interaction between two nodes. Not to scale. The top horizontal line represents wall time. The lower two represent the progression of local clock time for nodes **1** and **2**. The two nodes send a message at what each believes to be Tsend. Note that at time t_{send_1}, node **1**'s clock reads T_{send} and node **2**'s clock reads $T_{send} - \Delta$, indicating that $D_{\mathbf{12}}(t_{send_1}) = \Delta$. Each message may take a different amount of time to reach its destination. In this case, $m_{\mathbf{12}}$ takes the longest time possible and $m_{\mathbf{21}}$ takes the shortest time possible. Each node counts out τ_{wait} after sending its message, then checks to see if it should adjust its clock forward. Each node infers the other's relative clock skew based on the arrival time of the received message. In this case, node **1** realizes it is first and does not adjust its clock. Node **2** adjusts its clock forward to match **1**'s as closely as possible, in this case a little too much, since node **2** reaches T' ahead of **1**.

Algorithm 4.3.1: FAILSTOP-SYNC()

Let $N = \{n_1 ... n_k\}$ be a set of k participating nodes.
Let $i \in \{1...k\}$ be the number of the node executing the instance of the algorithm.
τ_{wait} will be derived during later analysis

global $timestamps[k] \leftarrow \{null, ..., null\}$

process RECEIVE-MESSAGES()
while $C_i(t) < T_{send} + \tau_{wait}$
do { **if** m_{ji} received
then $timestamps[j]$ = CURRENT-TIME()

process SEND-AND-ADJUST()
local $deltas[k] \leftarrow \{0, ...0, \}$

if $C_i(t) = T_{send}$
then {
for $j \leftarrow 1$ **to** k
do { **if** $j \neq i$
then Send message m_{ij} to n_j.

while $C_i(t) < T_{send} + \tau_{wait}$
do { Nothing

for $j \leftarrow 1$ **to** k
do { **if** $j \neq i$
then $deltas[j] \leftarrow$ CALCULATE-DELTA($timestamps[j]$)

SET-TIME(CURRENT-TIME() − MIN($deltas$))

procedure CALCULATE-DELTA($timestamp$)
$delta \leftarrow timestamp - (T_{send} + \ell_{assume})$
return ($delta$)

Figure 4-2: Failstop synchronization algorithm. Process Receive-Messages and process Send-And-Adjust run separately. The first process timestamps messages as they arrive from other nodes. The second sends messages, waits for all messages to arrive, then adjusts the clock forward if necessary.

4.3.1 Characterization of Algorithm for Two Nodes

This section provides the mathematical characterization of the failstop synchronization algorithm and demonstrates that it is correct.

The analysis is organized as follows:

1. First, bounds on the nodes' estimation of the other's clock skew are determined in terms the initial relative clock skew between the nodes.

2. Next, the estimation bounds from the previous result are used to determine the clock adjustment behavior under all possible initial relative clock skews.

3. After understanding how nodes adjust their clocks, bounds are derived for the maximum clock skew possible following the execution of the algorithm.

4. Next, an expression is derived for the actual maximum skew possible at any point during the algorithm in term of the elapsed time between synchronizations.

5. Finally, lower bounds are derived on how frequently the algorithm may be executed, thereby allowing the actual maximum synchronization to be calculated using the results from step 4.

Determining each node's perception of local clock difference

To begin, consider two nodes, **1** and **2**, performing algorithm 4.3.1. The analysis of the two nodes will not generalize to the case of three or more nodes, but it is described first because it is the most straightforward to analyze.

Without loss of generality, the local clock of node **1** is defined to be equal or ahead of **2**'s at the start of the algorithm. In other words,

$$D_{\mathbf{12}}(t_{start}) \geq 0.$$

The first step of analysis is to determine the bounds of **1** and **2**'s perception of the other nodes' local clocks in terms of the relative skew between them at the start of the algorithm.

The start of the algorithm is defined to be the time at which the first node sends a message. Since node **1**'s clock is defined to be ahead of **2**'s at the start of the algorithm, by definition,

$$D_{\mathbf{12}}(t_{start}) = D_{\mathbf{12}}(t_{send_{\mathbf{1}}}).$$

First consider how node **2** perceives the local clock of node **1**. Begin by recalling the formula that the nodes use to estimate the difference between their local clocks based on message arrival time,

$$[D_{ij}]_i = C_i(t_{m_{ji}}) - (T_{send} + \ell_{assume}).$$

Therefore, **2** predicts the difference between **1**'s and its own local clocks by

$$[D_{\mathbf{21}}]_{\mathbf{2}} = C_{\mathbf{2}}(t_{m_{\mathbf{12}}}) - (T_{send} + \ell_{assume}),$$

which may be also expressed as

$$[D_{\mathbf{21}}]_{\mathbf{2}} = t_{m_{\mathbf{12}}} + D_{\mathbf{2}}(t_{m_{\mathbf{12}}}) - (T_{send} + \ell_{assume}). \tag{4.3}$$

Next, consider amount of time that will elapse on **2**'s local clock from when **1** sends the message $m_{\mathbf{12}}$ at time $t_{send_{\mathbf{1}}}$ to when **2** receives the message at time $t_{m_{\mathbf{12}}}$. Equation (4.1) provides bounds on the elapsed time:

$$\begin{aligned} C_2(t_{m_{12}}) - C_2(t_{send_1}) &\leq (1+\rho)(t_{m_{12}} - t_{send_1}) \\ C_2(t_{m_{12}}) - C_2(t_{send_1}) &\geq (1+\rho)^{-1}(t_{m_{12}} - t_{send_1}). \end{aligned}$$

Since $C_i(t) = t + D_i(t)$, the above can be rewritten as

$$\begin{aligned} t_{m_{12}} + D_2(t_{m_{12}}) - t_{send_1} - D_2(t_{send_1}) &\leq (1+\rho)(t_{m_{12}} - t_{send_1}) \\ t_{m_{12}} + D_2(t_{m_{12}}) - t_{send_1} - D_2(t_{send_1}) &\geq (1+\rho)^{-1}(t_{m_{12}} - t_{send_1}). \end{aligned}$$

Isolating $D_2(t_{m_{12}})$ yields

$$\begin{aligned} D_2(t_{m_{12}}) &\leq (1+\rho)(t_{m_{12}} - t_{send_1}) - (t_{m_{12}} - t_{send_1}) + D_2(t_{send_1}) \\ D_2(t_{m_{12}}) &\geq (1+\rho)^{-1}(t_{m_{12}} - t_{send_1}) - (t_{m_{12}} - t_{send_1}) + D_2(t_{send_1}) \end{aligned}$$

$$\begin{aligned} \Rightarrow \quad D_2(t_{m_{12}}) &\leq \rho(t_{m_{12}} - t_{send_1}) + D_2(t_{send_1}) \\ D_2(t_{m_{12}}) &\geq -\left(\frac{\rho}{1+\rho}\right)(t_{m_{12}} - t_{send_1}) + D_2(t_{send_1}). \end{aligned}$$

Recognizing that $t_{m_{12}} - t_{send_1} = \ell_{m_{12}}$ and that $\ell_{min} \leq \ell_{m_{12}} \leq \ell_{max}$ gives

$$\begin{aligned} D_2(t_{m_{12}}) &\leq (\rho)\ell_{max} + D_2(t_{send_1}) \\ D_2(t_{m_{12}}) &\geq -\left(\frac{\rho}{1+\rho}\right)\ell_{max} + D_2(t_{send_1}). \end{aligned}$$

From these inequalities, bounds can now be found for $[D_{21}]_2$ using equation (4.3):

$$
\begin{aligned}
[D_{21}]_2 &\leq t_{m_{12}} + (\rho)\ell_{max} + D_2(t_{send_1}) - (T_{send} + \ell_{assume}) \\
[D_{21}]_2 &\geq t_{m_{12}} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} + D_2(t_{send_1}) - (T_{send} + \ell_{assume}).
\end{aligned}
$$

Finally, by recognizing that $T_{send} = C_1(t_{send_1}) = t_{send_1} + D_1(t_{send_1})$, the following bounds are achieved:

$$
\begin{aligned}
[D_{21}]_2 &\leq t_{m_{12}} + (\rho)\ell_{max} + D_2(t_{send_1}) - (t_{send_1} + D_1(t_{send_1}) + \ell_{assume}) \\
[D_{21}]_2 &\geq t_{m_{12}} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} + D_2(t_{send_1}) - (t_{send_1} + D_1(t_{send_1}) + \ell_{assume})
\end{aligned}
$$

$$
\begin{aligned}
\Rightarrow \quad [D_{21}]_2 &\leq (t_{m_{12}} - t_{send_1}) + (\rho)\ell_{max} + (D_2(t_{send_1}) - D_1(t_{send_1})) - \ell_{assume} \\
[D_{21}]_2 &\geq (t_{m_{12}} - t_{send_1}) - \left(\frac{\rho}{1+\rho}\right)\ell_{max} + (D_2(t_{send_1}) - D_1(t_{send_1})) - \ell_{assume}
\end{aligned}
$$

$$
\begin{aligned}
\Rightarrow \quad [D_{21}]_2 &\leq (\ell_{m_{12}}) + (\rho)\ell_{max} + (D_{21}(t_{send_1})) - \ell_{assume} \\
[D_{21}]_2 &\geq (\ell_{m_{12}}) - \left(\frac{\rho}{1+\rho}\right)\ell_{max} + (D_{21}(t_{send_1})) - \ell_{assume}
\end{aligned}
$$

$$
\begin{aligned}
\Rightarrow \quad [D_{21}]_2 &\leq \ell_{max} + (\rho)\ell_{max} - D_{12}(t_{send_1}) - \ell_{assume} \\
[D_{21}]_2 &\geq \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - D_{12}(t_{send_1}) - \ell_{assume},
\end{aligned}
$$

which finally simplifies to

$$
\begin{aligned}
[D_{21}]_2 &\leq (1+\rho)\ell_{max} - D_{12}(t_{send_1}) - \ell_{assume} \\
[D_{21}]_2 &\geq \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - D_{12}(t_{send_1}) - \ell_{assume}.
\end{aligned} \tag{4.4}
$$

In the next step, the bounds of **1**'s perception of **2**'s local clock are determined. Begin by recognizing that

$$[D_{12}]_1 = C_1(t_{m_{21}}) - (T_{send} + \ell_{assume}). \tag{4.5}$$

The challenge here is to find $C_1(t_{m_{21}})$ in terms of the local clock difference at the start of the algorithm, namely $D_{12}(t_{send_1})$. Begin by determining the bounds of the elapsed time on **1**'s local clock in terms of real time. Equation (4.1) gives the following:

$$\begin{aligned} C_1(t_{m_{21}}) - C_1(t_{send_1}) &\leq (1+\rho)(t_{m_{21}} - t_{send_1}) \\ C_1(t_{m_{21}}) - C_1(t_{send_1}) &\geq (1+\rho)^{-1}(t_{m_{21}} - t_{send_1}). \end{aligned} \tag{4.6}$$

Since $t_{m_{21}} = t_{send_2} + \ell_{m_{21}}$ and $\ell_{min} \leq \ell_{m_{21}} \leq \ell_{max}$, bounds on $t_{m_{21}}$ are as follows

$$\begin{aligned} t_{m_{21}} &\leq t_{send_2} + \ell_{max} \\ t_{m_{21}} &\geq t_{send_2} + \ell_{min}. \end{aligned} \tag{4.7}$$

Bounds on t_{send_2} can be determined using equation (4.2):

$$\begin{aligned} t_{send_2} - t_{send_1} &\leq (1+\rho)(C_2(t_{send_2}) - C_2(t_{send_1})) \\ t_{send_2} - t_{send_1} &\geq (1+\rho)^{-1}(C_2(t_{send_2}) - C_2(t_{send_1})). \end{aligned}$$

Recognizing that $C_2(t_{send_2}) = T_{send} = C_1(t_{send_1})$ gives

$$
\begin{aligned}
t_{send_2} - t_{send_1} &\leq (1+\rho)(C_1(t_{send_1}) - C_2(t_{send_1})) \\
t_{send_2} - t_{send_1} &\geq (1+\rho)^{-1}(C_1(t_{send_1}) - C_2(t_{send_1}))
\end{aligned}
$$

$$
\begin{aligned}
\Rightarrow \quad t_{send_2} - t_{send_1} &\leq (1+\rho)D_{12}(t_{send_1}) \\
t_{send_2} - t_{send_1} &\geq (1+\rho)^{-1}D_{12}(t_{send_1})
\end{aligned}
$$

$$
\begin{aligned}
\Rightarrow \quad t_{send_2} &\leq (1+\rho)D_{12}(t_{send_1}) + t_{send_1} \\
t_{send_2} &\geq (1+\rho)^{-1}D_{12}(t_{send_1}) + t_{send_1}.
\end{aligned}
$$

Substituting for t_{send_2} in equation (4.7) gives

$$
\begin{aligned}
t_{m_{21}} &\leq (1+\rho)D_{12}(t_{send_1}) + t_{send_1} + \ell_{max} \\
t_{m_{21}} &\geq (1+\rho)^{-1}D_{12}(t_{send_1}) + t_{send_1} + \ell_{min}.
\end{aligned}
$$

Next, referring back to the bounds on $C_1(t_{m_{21}}) - C_1(t_{send_1})$ given by equation (4.6) and substituting for $t_{m_{21}}$ yields

$$
\begin{aligned}
C_1(t_{m_{21}}) - C_1(t_{send_1}) &\leq (1+\rho)((1+\rho)D_{12}(t_{send_1}) + t_{send_1} + \ell_{max} - t_{send_1}) \\
C_1(t_{m_{21}}) - C_1(t_{send_1}) &\geq (1+\rho)^{-1}((1+\rho)^{-1}D_{12}(t_{send_1}) + t_{send_1} + \ell_{min} - t_{send_1})
\end{aligned}
$$

$$
\begin{aligned}
\Rightarrow \quad C_1(t_{m_{21}}) - C_1(t_{send_1}) &\leq (1+\rho)^2 D_{12}(t_{send_1}) + (1+\rho)\ell_{max} \\
C_1(t_{m_{21}}) - C_1(t_{send_1}) &\geq (1+\rho)^{-2}D_{12}(t_{send_1}) + (1+\rho)^{-1}\ell_{min}.
\end{aligned}
$$

Finally, since $C_1(t_{send_1}) = T_{send}$, the above bounds can be directly substituted into equa-

tion (4.5), determining the bounds on $[D_{\mathbf{12}}]_{\mathbf{1}}$ in terms of $D_{\mathbf{12}}(t_{send_1})$:

$$\begin{aligned} [D_{\mathbf{12}}]_{\mathbf{1}} &\leq (1+\rho)^2 D_{\mathbf{12}}(t_{send_1}) + (1+\rho)\ell_{max} - \ell_{assume} \\ [D_{\mathbf{12}}]_{\mathbf{1}} &\geq (1+\rho)^{-2} D_{\mathbf{12}}(t_{send_1}) + (1+\rho)^{-1}\ell_{min} - \ell_{assume}. \end{aligned} \tag{4.8}$$

Determining limits of behavior

Now that the bounds for $[D_{\mathbf{12}}]_{\mathbf{1}}$ and $[D_{\mathbf{21}}]_{\mathbf{2}}$ have been determined in terms of the initial clock difference, it is possible to evaluate how nodes will adjust their local clocks based on their approximation of their distance from the other node.

The amount that a node i adjusts its clock is a function of $[D_{ij}]_i$, its approximation of its own clock's distance from j's clock, but the function is not smooth over all values of $[D_{ij}]_i$, but rather a piecewise function of the form

$$adjust([D_{ij}]_i) = \begin{cases} -[D_{ij}]_i & , [D_{ij}]_i < 0 \\ 0 & , [D_{ij}]_i \geq 0 \end{cases}.$$

The piecewise nature of the adjustment function requires the behavior analysis of the synchronization to be applied in a piecewise manner as well, with discontinuities existing when either $[D_{ij}]_i = 0$ or $[D_{ji}]_j = 0$.

In other words, the behavior of the system depends on whether one, both, or neither node adjusts its local clock during the algorithm. Different equations will bound the achievable synchronization for each case.

Returning to nodes **1** and **2**, the following questions are considered:

1. Under what conditions is $[D_{\mathbf{12}}]_{\mathbf{1}} \geq 0$ possible?

2. Under what conditions is $[D_{12}]_1 < 0$ possible?

3. Under what conditions is $[D_{21}]_2 \geq 0$ possible?

4. Under what conditions is $[D_{21}]_2 < 0$ possible?

The conditions for all cases are expressed in terms of $D_{12}(t_{send_i})$:

$[\mathbf{D_{12}}]_1 \geq 0$ possible: $[D_{12}]_1 \geq 0$ is possible when the upper bound of $[D_{12}]_1$ is positive. Equation (4.8) gives

$$\begin{aligned} (1+\rho)^2 D_{12}(t_{send_1}) + (1+\rho)\ell_{max} - \ell_{assume} &\geq 0 \\ \Rightarrow \qquad D_{12}(t_{send_1}) &\geq (1+\rho)^{-2}\ell_{assume} - (1+\rho)^{-1}\ell_{max}. \end{aligned}$$

Note that, by definition, $\rho > 0$, $\ell_{assume} \leq \ell_{max}$, and $D_{12}(t_{send_1}) \geq 0$, thus the above inequality is always true. Therefore, it is always possible that node **1** estimates its local clock to be ahead of **2**'s. This makes sense, since **1**'s clock is defined to be the same or ahead of **2**'s at the start of the algorithm.

$[\mathbf{D_{12}}]_1 < 0$ possible: $[D_{12}]_1 < 0$ is possible when the lower bound of $[D_{12}]_1$ is negative. Equation (4.8) gives

$$\begin{aligned} (1+\rho)^{-2} D_{12}(t_{send_1}) + (1+\rho)^{-1}\ell_{min} - \ell_{assume} &< 0 \\ \Rightarrow \qquad D_{12}(t_{send_1}) &< (1+\rho)^2\ell_{assume} - (1+\rho)\ell_{min}. \end{aligned}$$

The above condition can be true or false depending on the values of ρ, ℓ_{max} and ℓ_{assume}.

$[\mathbf{D_{21}}]_2 \geq 0$ possible: Equation (4.4) gives

$$(1+\rho)\ell_{max} - D_{12}(t_{send_1}) - \ell_{assume} \geq 0$$

$$\Rightarrow \qquad D_{\mathbf{12}}(t_{send_1}) \;\leq\; (1+\rho)\ell_{max} - \ell_{assume}.$$

$[\mathbf{D_{21}}]_2 < \mathbf{0}$ possible: Equation (4.4) gives

$$\begin{aligned} \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - D_{\mathbf{12}}(t_{send_1}) - \ell_{assume} &< 0 \\ \Rightarrow \qquad D_{\mathbf{12}}(t_{send_1}) &> \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - \ell_{assume}. \end{aligned}$$

Since $\ell_{assume} > \ell_{min}$, this inequality is always true.

With these limits established, limits can now be found for each of the behavior cases by looking at the intersection of the limits.

$[\mathbf{D_{12}}]_1 \geq \mathbf{0} \wedge [\mathbf{D_{21}}]_2 < \mathbf{0}$ possible: The case of only node **2** adjusting its clock is always possible, i.e.,

$$\begin{aligned} D_{\mathbf{12}}(t_{send_1}) &\geq (1+\rho)^{-2}\ell_{assume} - (1+\rho)^{-1}\ell_{max} \\ D_{\mathbf{12}}(t_{send_1}) &> \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - \ell_{assume} \end{aligned}$$

$$\Rightarrow \quad D_{\mathbf{12}}(t_{send_i}) > 0.$$

$[\mathbf{D_{12}}]_1 < \mathbf{0} \wedge [\mathbf{D_{21}}]_2 < \mathbf{0}$ possible: The case of both nodes **1** and **2** adjusting their clocks is possible when

$$\begin{aligned} D_{\mathbf{12}}(t_{send_1}) &< (1+\rho)^2\ell_{assume} - (1+\rho)\ell_{min} \\ D_{\mathbf{12}}(t_{send_1}) &> \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - \ell_{assume} \end{aligned}$$

$$\begin{aligned} \Rightarrow \quad D_{\mathbf{12}}(t_{send_1}) &< (1+\rho)^2\ell_{assume} - (1+\rho)\ell_{min} \\ D_{\mathbf{12}}(t_{send_1}) &> 0 \end{aligned}$$

$[\mathbf{D_{12}}]_1 < \mathbf{0} \wedge [\mathbf{D_{21}}]_2 \geq \mathbf{0}$ **possible:** The case of only node **1** adjusting its clock is possible when

$$\begin{aligned} D_{\mathbf{12}}(t_{send_1}) &< (1+\rho)^2\ell_{assume} - (1+\rho)\ell_{min} \\ D_{\mathbf{12}}(t_{send_1}) &\leq (1+\rho)\ell_{max} - \ell_{assume} \end{aligned}$$

$[\mathbf{D_{12}}]_1 \geq \mathbf{0} \wedge [\mathbf{D_{21}}]_2 \geq \mathbf{0}$ **possible:** Finally, the case of neither node adjusting its clock is possible when

$$\begin{aligned} D_{\mathbf{12}}(t_{send_1}) &\geq (1+\rho)^{-2}\ell_{assume} - (1+\rho)^{-1}\ell_{max} \\ D_{\mathbf{12}}(t_{send_1}) &\leq (1+\rho)\ell_{max} - \ell_{assume} \end{aligned}$$

$$\begin{aligned} \Rightarrow \quad D_{\mathbf{12}}(t_{send_1}) &\geq 0 \\ D_{\mathbf{12}}(t_{send_1}) &\leq (1+\rho)\ell_{max} - \ell_{assume} \end{aligned}$$

From these results, the range of $D_{\mathbf{12}}(t_{send_1})$ can be broken up into regions where different clock adjustment behavior is known to occur. Analysis can then be performed for each region independently, then compared. The regions depend on the value of ℓ_{assume}. Let ℓ_{mid} be defined as the value of ℓ_{assume} when $(1+\rho)\ell_{max} - \ell_{assume} = (1+\rho)^2\ell_{assume} - (1+\rho)\ell_{min}$. Then,

$$\ell_{mid} = ((1+\rho)\ell_{max} + (1+\rho)\ell_{min})\left(1 + (1+\rho)^2\right)^{-1}.$$

If

$$\ell_{assume} > \ell_{mid},$$

then

$$(1+\rho)^2\ell_{assume} - (1+\rho)\ell_{min} > (1+\rho)\ell_{max} - \ell_{assume},$$

and the possible node behavior in different regions are as follows:

$$\begin{aligned} D_{\mathbf{12}}(t_{send_1}) &\in [0, (1+\rho)\ell_{max} - \ell_{assume}] : \mathbf{1} \text{ sets, } \mathbf{2} \text{ sets, both set, neither sets} \\ &\in [(1+\rho)\ell_{max} - \ell_{assume}, (1+\rho)^2\ell_{assume} - (1+\rho)\ell_{min}] : \mathbf{2} \text{ sets, both set} \\ &\in [(1+\rho)^2\ell_{assume} - (1+\rho)\ell_{min}, \infty) : \mathbf{2} \text{ sets.} \end{aligned}$$

Alternatively, if

$$\ell_{assume} < \ell_{mid},$$

then

$$(1+\rho)^2\ell_{assume} - (1+\rho)\ell_{min} < (1+\rho)\ell_{max} - \ell_{assume},$$

and the possible node behavior in different regions are instead

$$\begin{aligned} D_{\mathbf{12}}(t_{send_1}) &\in [0, (1+\rho)^2\ell_{assume} - (1+\rho)\ell_{min}] : \mathbf{1} \text{ sets, } \mathbf{2} \text{ sets, both set, neither sets} \\ &\in [(1+\rho)^2\ell_{assume} - (1+\rho)\ell_{min}, (1+\rho)\ell_{max} - \ell_{assume}] : \mathbf{2} \text{ sets, neither sets} \\ &\in [(1+\rho)\ell_{max} - \ell_{assume}, \infty) : \mathbf{2} \text{ sets.} \end{aligned}$$

Determining limits of synchronization

The maximum synchronization achievable through the algorithm depends not only on environment variables like ℓ_{max}, ℓ_{min}, and ρ, but also on how frequently the synchronization

algorithm can run. Given the synchronization of the nodes $D_{\mathbf{12}}(t_{start})$ at the start of a round and the maximum time $dt_{between}$ between rounds, the synchronization of the nodes at the start of the next round, $D_{\mathbf{12}}(t^{(2)}_{start})$, is bounded by

$$\begin{aligned} D_{\mathbf{12}}(t^{(2)}_{start}) &\leq D_{\mathbf{12}}(t_{start}) + \Delta_{sync_{max}} + dr \cdot dt_{between} \\ D_{\mathbf{12}}(t^{(2)}_{start}) &\geq D_{\mathbf{12}}(t_{start}) + \Delta_{sync_{min}} - dr \cdot dt_{between}, \end{aligned} \tag{4.9}$$

where $\Delta_{sync_{min}}$ and $\Delta_{sync_{max}}$ are the minimum and maximum change in clock skew due to adjustment in a single round, and $dr \cdot dt_{between}$ is the maximum drift between two local clocks possible between the start of two rounds. Note $\Delta_{sync_{min}}$ and $\Delta_{sync_{max}}$ may be broken down further into

$$\begin{aligned} \Delta_{sync_{min}} &= \Delta_{sync_{min_{\mathbf{1}}}} + \Delta_{sync_{min_{\mathbf{2}}}} \\ \Delta_{sync_{max}} &= \Delta_{sync_{max_{\mathbf{1}}}} + \Delta_{sync_{max_{\mathbf{2}}}}, \end{aligned}$$

where $\Delta_{sync_{min_{\mathbf{1}}}}$ is the change contributed by node **1** and $\Delta_{sync_{max_{\mathbf{2}}}}$ is the change contributed by node **2**. Note that a positive value of $\Delta_{sync_{min}}$ or $\Delta_{sync_{max}}$ indicates that node **1** adjusted its clock forward (generally undesirable) while a negative value means that node **2** adjusted its clock forward (generally desirable).

The previous section determined the boundary cases for how nodes adjust their clocks, so it is now possible to calculate $\Delta_{sync_{min}}$ and $\Delta_{sync_{max}}$ for each of the different adjustment cases in each of the different regions.

Case $\ell_{assume} > \ell_{mid}$**,** $D_{\mathbf{12}}(t_{send_{\mathbf{1}}}) \in [0, (1+\rho)\ell_{max} - \ell_{assume}]$**:** In this case the smallest contribution of node **1** is

$$\begin{aligned}\Delta_{sync_{min_{\mathbf{1}}}} &= -\min\{0, \max\{[D_{\mathbf{12}}]_{\mathbf{1}}\}\} \\ &= -\min\left\{0, (1+\rho)^2 D_{\mathbf{12}}(t_{send_{\mathbf{1}}}) + (1+\rho)\ell_{max} - \ell_{assume}\right\} \\ &= 0.\end{aligned}$$

The largest contribution of node **1** is

$$\begin{aligned}\Delta_{sync_{max_{\mathbf{1}}}} &= -\min\{0, \min\{[D_{\mathbf{12}}]_{\mathbf{1}}\}\} \\ &= -\min\left\{0, (1+\rho)^{-2} D_{\mathbf{12}}(t_{send_{\mathbf{1}}}) + (1+\rho)^{-1}\ell_{min} - \ell_{assume}\right\} \\ &= -(1+\rho)^{-2} D_{\mathbf{12}}(t_{send_{\mathbf{1}}}) - (1+\rho)^{-1}\ell_{min} + \ell_{assume}.\end{aligned}$$

The smallest contribution of node **2** is

$$\begin{aligned}\Delta_{sync_{min_{\mathbf{2}}}} &= \min\{0, \min\{[D_{\mathbf{21}}]_{\mathbf{2}}\}\} \\ &= \min\left\{0, \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - D_{\mathbf{12}}(t_{send_{\mathbf{1}}}) - \ell_{assume}\right\} \\ &= \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - D_{\mathbf{12}}(t_{send_{\mathbf{1}}}) - \ell_{assume}.\end{aligned}$$

The largest contribution of node **2** is

$$\begin{aligned}\Delta_{sync_{max_{\mathbf{2}}}} &= \min\{0, \max\{[D_{\mathbf{21}}]_{\mathbf{2}}\}\} \\ &= \min\{0, (1+\rho)\ell_{max} - D_{\mathbf{12}}(t_{send_{\mathbf{1}}}) - \ell_{assume}\} \\ &= 0.\end{aligned}$$

Therefore, for the case $\ell_{assume} > \ell_{mid}$, $D_{\mathbf{12}}(t_{send_1}) \in [0, (1+\rho)\ell_{max} - \ell_{assume}]$,

$$\begin{aligned} \Delta_{sync_{min}} &= 0 + \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - D_{\mathbf{12}}(t_{send_1}) - \ell_{assume} \\ \Delta_{sync_{max}} &= -(1+\rho)^{-2} D_{\mathbf{12}}(t_{send_1}) - (1+\rho)^{-1}\ell_{min} + \ell_{assume} + 0 \end{aligned}$$

$$\begin{aligned} \Rightarrow \quad \Delta_{sync_{min}} &= \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - D_{\mathbf{12}}(t_{send_1}) - \ell_{assume} \\ \Delta_{sync_{max}} &= -(1+\rho)^{-2} D_{\mathbf{12}}(t_{send_1}) - (1+\rho)^{-1}\ell_{min} + \ell_{assume}. \end{aligned}$$

Substituting these results into equation (4.9) yields

$$\begin{aligned} D_{\mathbf{12}}(t^{(2)}_{start}) &\leq D_{\mathbf{12}}(t_{send_1}) - (1+\rho)^{-2} D_{\mathbf{12}}(t_{send_1}) - (1+\rho)^{-1}\ell_{min} + \ell_{assume} + dr \cdot dt_{between} \\ D_{\mathbf{12}}(t^{(2)}_{start}) &\geq D_{\mathbf{12}}(t_{send_1}) + \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - D_{\mathbf{12}}(t_{send_1}) - \ell_{assume} - dr \cdot dt_{between} \end{aligned}$$

$$\begin{aligned} \Rightarrow \quad D_{\mathbf{12}}(t^{(2)}_{start}) &\leq \left(1 - (1+\rho)^{-2}\right) D_{\mathbf{12}}(t_{send_1}) - (1+\rho)^{-1}\ell_{min} + \ell_{assume} + dr \cdot dt_{between} \\ D_{\mathbf{12}}(t^{(2)}_{start}) &\geq \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - \ell_{assume} - dr \cdot dt_{between}. \end{aligned}$$

Since $D_{\mathbf{12}}(t_{send_1}) \leq (1+\rho)\ell_{max} - \ell_{assume}$, substituting for $D_{\mathbf{12}}(t_{send_1})$ gives

$$\begin{aligned} D_{\mathbf{12}}(t^{(2)}_{start}) &\leq \left(1 - (1+\rho)^{-2}\right)((1+\rho)\ell_{max} - \ell_{assume}) - (1+\rho)^{-1}\ell_{min} + \ell_{assume} + dr \cdot dt_{between} \\ D_{\mathbf{12}}(t^{(2)}_{start}) &\geq \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - \ell_{assume} - dr \cdot dt_{between} \end{aligned}$$

$$\begin{aligned} \Rightarrow \quad D_{\mathbf{12}}(t^{(2)}_{start}) &\leq \left((1+\rho) - (1+\rho)^{-1}\right)\ell_{max} + (1+\rho)^{-2}\ell_{assume} - (1+\rho)^{-1}\ell_{min} + dr \cdot dt_{between} \\ D_{\mathbf{12}}(t^{(2)}_{start}) &\geq \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - \ell_{assume} - dr \cdot dt_{between}. \end{aligned}$$

Case $\ell_{assume} > \ell_{mid}$, $D_{\mathbf{12}}(t_{send_1}) \in [(1+\rho)\ell_{max} - \ell_{assume}, (1+\rho)^2\ell_{assume} - (1+\rho)\ell_{min}]$: In this case, the only possible behaviors are that only node **2** adjusts its clock or that both nodes **1**

and **2** adjust their clocks.

The smallest contribution of node **1** is

$$\begin{aligned}\Delta_{sync_{min_{\mathbf{1}}}} &= -\min\{0, \max\{[D_{\mathbf{12}}]_{\mathbf{1}}\}\} \\ &= 0.\end{aligned}$$

The largest contribution of node **1** is

$$\begin{aligned}\Delta_{sync_{max_{\mathbf{1}}}} &= -\min\{0, \min\{[D_{\mathbf{12}}]_{\mathbf{1}}\}\} \\ &= -\min\left\{0, (1+\rho)^{-2} D_{\mathbf{12}}(t_{send_{\mathbf{1}}}) + (1+\rho)^{-1}\ell_{min} - \ell_{assume}\right\}.\end{aligned}$$

The smallest contribution of node **2** is

$$\begin{aligned}\Delta_{sync_{min_{\mathbf{2}}}} &= \min\{0, \min\{[D_{\mathbf{21}}]_{\mathbf{2}}\}\} \\ &= \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - D_{\mathbf{12}}(t_{send_{\mathbf{1}}}) - \ell_{assume}.\end{aligned}$$

The largest contribution of node **2** is

$$\begin{aligned}\Delta_{sync_{max_{\mathbf{2}}}} &= \min\{0, \max\{[D_{\mathbf{21}}]_{\mathbf{2}}\}\} \\ &= (1+\rho)\ell_{max} - D_{\mathbf{12}}(t_{send_{\mathbf{1}}}) - \ell_{assume}.\end{aligned}$$

Therefore, for the case $\ell_{assume} > \ell_{mid}$, $D_{\mathbf{12}}(t_{send_{\mathbf{1}}}) \in [(1+\rho)\ell_{max} - \ell_{assume}, (1+\rho)^2\ell_{assume} -$

$(1+\rho)\ell_{min}]$,

$$\begin{aligned}
\Delta_{sync_{min}} &= \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - D_{\mathbf{12}}(t_{send_{\mathbf{1}}}) - \ell_{assume} \\
\Delta_{sync_{max}} &= -\min\left\{0, (1+\rho)^{-2} D_{\mathbf{12}}(t_{send_{\mathbf{1}}}) + (1+\rho)^{-1}\ell_{min} - \ell_{assume}\right\} \\
&\quad +(1+\rho)\ell_{max} - D_{\mathbf{12}}(t_{send_{\mathbf{1}}}) - \ell_{assume}.
\end{aligned}$$

Substituting these results into equation (4.9) yields

$$\begin{aligned}
D_{\mathbf{12}}(t^{(2)}_{start}) &\leq -\min\left\{0, (1+\rho)^{-2} D_{\mathbf{12}}(t_{send_{\mathbf{1}}}) + (1+\rho)^{-1}\ell_{min} - \ell_{assume}\right\} \\
&\quad +(1+\rho)\ell_{max} - \ell_{assume} + dr \cdot dt_{between} \\
D_{\mathbf{12}}(t^{(2)}_{start}) &\geq \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - \ell_{assume} - dr \cdot dt_{between}.
\end{aligned}$$

The smallest possible value for $D_{\mathbf{12}}(t_{send_{\mathbf{1}}})$ in this case is chosen as a substitution, giving

$$\begin{aligned}
D_{\mathbf{12}}(t^{(2)}_{start}) &\leq -\min\left\{0, (1+\rho)^{-2}\left((1+\rho)\ell_{max} - \ell_{assume}\right) + (1+\rho)^{-1}\ell_{min} - \ell_{assume}\right\} \\
&\quad +(1+\rho)\ell_{max} - \ell_{assume} + dr \cdot dt_{between} \\
D_{\mathbf{12}}(t^{(2)}_{start}) &\geq \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - \ell_{assume} - dr \cdot dt_{between}
\end{aligned}$$

$$\begin{aligned}
\Rightarrow \quad D_{\mathbf{12}}(t^{(2)}_{start}) &\leq -\left((1+\rho)^{-2}\left((1+\rho)\ell_{max} - \ell_{assume}\right) + (1+\rho)^{-1}\ell_{min} - \ell_{assume}\right) \\
&\quad +(1+\rho)\ell_{max} - \ell_{assume} + dr \cdot dt_{between} \\
D_{\mathbf{12}}(t^{(2)}_{start}) &\geq \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - \ell_{assume} - dr \cdot dt_{between},
\end{aligned}$$

which simplifies to

$$
\begin{aligned}
D_{\mathbf{12}}(t^{(2)}_{start}) &\leq \left((1+\rho)-(1+\rho)^{-1}\right)\ell_{max} - (1+\rho)^{-1}\ell_{min} + (1+\rho)^{-2}\ell_{assume} + dr \cdot dt_{between} \\
D_{\mathbf{12}}(t^{(2)}_{start}) &\geq \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - \ell_{assume} - dr \cdot dt_{between}.
\end{aligned}
$$

Case $\ell_{assume} > \ell_{mid}$**,** $D_{\mathbf{12}}(t_{send_1}) \in [(1+\rho)^2\ell_{assume} - (1+\rho)\ell_{min}], \infty$**:** In this case, the only possible behavior is that only node **2** adjusts its clock. The limits of adjustment are

$$
\begin{aligned}
\Delta_{sync_{min_{\mathbf{1}}}} &= 0 \\
\Delta_{sync_{max_{\mathbf{1}}}} &= 0 \\
\Delta_{sync_{min_{\mathbf{2}}}} &= \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - D_{\mathbf{12}}(t_{send_1}) - \ell_{assume} \\
\Delta_{sync_{max_{\mathbf{2}}}} &= (1+\rho)\ell_{max} - D_{\mathbf{12}}(t_{send_1}) - \ell_{assume},
\end{aligned}
$$

giving

$$
\begin{aligned}
\Delta_{sync_{min}} &= \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - D_{\mathbf{12}}(t_{send_1}) - \ell_{assume} \\
\Delta_{sync_{max}} &= (1+\rho)\ell_{max} - D_{\mathbf{12}}(t_{send_1}) - \ell_{assume}
\end{aligned}
$$

and

$$
\begin{aligned}
D_{\mathbf{12}}(t^{(2)}_{start}) &\leq (1+\rho)\ell_{max} - \ell_{assume} + dr \cdot dt_{between} \\
D_{\mathbf{12}}(t^{(2)}_{start}) &\geq \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - \ell_{assume} - dr \cdot dt_{between}.
\end{aligned}
$$

The same analysis is now performed for the cases where $\ell_{assume} < \ell_{mid}$.

Case $\ell_{assume} < \ell_{mid}$, $D_{\mathbf{12}}(t_{send_1}) \in [0, (1+\rho)^2\ell_{assume} - (1+\rho)\ell_{min}]$:

$$\begin{aligned}
\Delta_{sync_{min_{\mathbf{1}}}} &= -\min\{0, \max\{[D_{\mathbf{12}}]_{\mathbf{1}}\}\} \\
&= -\min\left\{0, (1+\rho)^2 D_{\mathbf{12}}(t_{send_1}) + (1+\rho)\ell_{max} - \ell_{assume}\right\} \\
&= 0
\end{aligned}$$

$$\begin{aligned}
\Delta_{sync_{max_{\mathbf{1}}}} &= -\min\{0, \min\{[D_{\mathbf{12}}]_{\mathbf{1}}\}\} \\
&= -\min\left\{0, (1+\rho)^{-2} D_{\mathbf{12}}(t_{send_1}) + (1+\rho)^{-1}\ell_{min} - \ell_{assume}\right\} \\
&= -(1+\rho)^{-2} D_{\mathbf{12}}(t_{send_1}) - (1+\rho)^{-1}\ell_{min} + \ell_{assume}
\end{aligned}$$

$$\begin{aligned}
\Delta_{sync_{min_{\mathbf{2}}}} &= \min\{0, \min\{[D_{\mathbf{21}}]_{\mathbf{2}}\}\} \\
&= \min\left\{0, \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - D_{\mathbf{12}}(t_{send_1}) - \ell_{assume}\right\} \\
&= \ell_{min} - \left(\frac{\rho}{1+\rho}\right)\ell_{max} - D_{\mathbf{12}}(t_{send_1}) - \ell_{assume}
\end{aligned}$$

$$\begin{aligned}
\Delta_{sync_{max_{\mathbf{2}}}} &= \min\{0, \max\{[D_{\mathbf{21}}]_{\mathbf{2}}\}\} \\
&= \min\{0, (1+\rho)\ell_{max} - D_{\mathbf{12}}(t_{send_1}) - \ell_{assume}\},
\end{aligned}$$

which gives