An Architecture Study of a Byzantine-Resilient Processor Using Authentication

by

Anne L. Clark

Submitted to the

Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements
for the degree of

Master of Science

at the

Massachusetts Institute of Technology

June 1994

Signature of Author____________________________
Anne L. Clark
Department of Electrical Engineering and Computer Science, May 6, 1994

Certified by____________________________
Stephen A. Ward
Thesis Supervisor, Massachusetts Institute of Technology

Approved by____________________________
Richard E. Harper
Thesis Supervisor, Charles Stark Draper Laboratory

Accepted by____________________________
F. R. Morgenthaler
Chair, Department Committee on Graduate Students

Eng.

An Architecture Study of a Byzantine-Resilient Processor Using Authentication

by

Anne L. Clark

Submitted to the Department of Electrical Engineering and Computer Science on May 6, 1994 in partial fulfillment of the requirements for the Degree of Master of Science in Electrical Engineering and Computer Science.

ABSTRACT

This architecture study provides the ground work for implementing a new generation of Byzantine resilient processors using authentication. The use of authentication allows a significant reduction in the theoretical requirements necessary for providing Byzantine resilience, or the ability to continue correct operation in the presence of arbitrary or even malicious faults. This decrease in requirements led to a goal of providing a system which combines the stringent standards embodied by Byzantine resilience with the lower costs necessary to make the system viable for more markets than previous Byzantine resilient processors.

A layering scheme is proposed which can be placed between the user and hardware. These layers consist of protocols which provide the basic building blocks of the architecture. The proposed authentication protocol which provides the digital signatures used to verify the origin and contents of messages is a public-key protocol using 32-bit Cyclic Redundancy Codes (CRC's) to encode the message with 32-bit modular inverse key pairs to sign and authenticate the CRC. An interactive consistency protocol responsible for correctly distributing single-source data between processors is built using the *SM(m)* algorithm from [LSP82] with improvements suggested in [Dol83]. A voting protocol responsible for generating a group consensus value guaranteed to be the same on all nonfaulty processors suggests exchanging unsigned messages and then using a full-set majority vote *choice()* function to calculate the group consensus value. Finally, the proposed synchronization protocol needed to provide synchronized virtual clocks on all nonfaulty processors is placed on top of a full message exchange (FME) known as a *From_all* exchange to read the clocks on other processors. A time adjustment is then calculated using a technique suggested in [LM84].

Thesis Supervisor: Stephen A. Ward

Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I thank Dr. Richard Harper for his support and guidance throughout the writing of this thesis. A true Southern gentleman, Rick's clear sightedness and sense of humor kept everything moving in the correct direction and smoothed all of the rough spots. The last two years of working for him have been a true pleasure.

I thank Dr. Stephen Ward for his contributions. His supervision guaranteed that this thesis would be up to the high standards of MIT.

Many thanks to everyone else in the Fault-Tolerant Computer Group at CSDL: Jay Lala (even if your tie always found all of the problems), Carol (for the Saturday conversations which made being at work on the weekend all right), Gail (for your patience in teaching me C and all of the Emacs short cuts), and Florie (for the scissors, pens, stapler, typewriter, etc., and most importantly, the smile every morning).

Last but definitely not least, thanks to all of the people who made MIT such a fun experience: Ashok (the cutest and cuddliest), Chris (I would have gone crazy without the company and card games), Maria (did we really spend Friday nights doing the recycling?!!), Rob (you still need to relax), Jeff and the rest of the Air Force crowd, and all of my friends in Edgerton Hall.

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under Internal Research and Development Project No. 521: Digital Signatures for Fault Tolerant Computing.

To Mom, Dad, David, and Steven - For the love and support.

LIST OF FIGURES

LIST OF TABLES

1. Introduction

1.1. Problem Statement

Mission- and life-critical computing systems are demanding increasing levels of reliability. Current allowable failure probabilities range from 10^{-4} to 10^{-6} per hour for mission-critical functions and 10^{-6} to 10^{-10} per hour for vehicle-critical and crew-safety functions [HL91]. As a result, the traditional method of designing fault-tolerant processors using a failure modes and effects analysis (FMEA)-based approach has become extremely costly and time-consuming. To guarantee a certain reliability, these systems not only have to show that the probability of a modeled fault occurring is within parameters, but that the chances of an unpredicted fault is also within bounds. An alternative to this technique avoids making a priori assumptions by allowing faults to act in any manner, up to and including malicious and intelligent behavior. Such a system is called "Byzantine resilient," or capable of withstanding Byzantine faults.

Making an architecture Byzantine resilient provides the ability to mask, or continue operation in the presence of, a specific number of faults. The requirements for such fault-masking involve lower bounds based on the number of faults that the designer wishes to protect against. These bounds determine how many processors must be in the system, what the connectivity between the processors must be, and how many times information must be exchanged between processors. An additional requirement that the individual processors be synchronized to within a known skew prevents one processor from deadlocking the system.

Theoretically, a f-Byzantine resilient processor which uses an authenticated protocol (i.e., a protocol where messages are signed with digital signatures to allow the detection of certain faults) has definite advantages over those using unauthenticated protocols. These advantages include a decrease by a factor of f in the number of processors, the ability to eliminate of voting of entire messages, and a reduction in required message passing and connectivity. The problem is that these advantages do not necessarily translate into a faster, more efficient architecture when implemented. For example, the increased message length due to signatures could seriously affect latency. Before a processor providing Byzantine resilience using authentication can be designed and implemented, a study is required to investigate the fault-tolerance issues involved in such a computer architecture, to identify and propose solutions for the main functional blocks needed for implementation, and then to implement sections of the architecture to make decisions based on performance issues and to pinpoint areas needing optimization.

1.2. Objective

There are certain practical issues relating to an architecture's Byzantine resilience and multiprocessing which must be resolved: authentication, interactive consistency, voting, fault-tolerant clock synchronization, and the performance of the proposed architecture. The proposed architecture is centered around a method of signing messages that provides unforgeable signatures which (1) allow detection of any alteration of the message's contents and (2) can be authenticated by any processor in the system. A cryptographic scheme must be chosen to balance the needs of data integrity against performance. Interactive consistency (a process where all nonfaulty processors agree on a vector, y) between the processors must be provided through a fault-tolerant distributed decision algorithm. A voting protocol is needed to check for faults by exchanging congruent information, that is, data which are the same on all nonfaulty processors. Fault tolerance issues require that the clocks on all nonfaulty processors be synchronized to within some known skew in order to guarantee termination of tasks [FLP83]. Finally, performance issues involved in all of the proposed solutions (i.e., added latency due to message signatures) must be taken into account. The objective of this thesis is to identify and evaluate solutions to these problems.

1.3. General Approach

The first step in any computer design is to examine the motivations behind developing the architecture. These motivations heavily influence any decisions which must be made and therefore need to be thoroughly understood. Their relative importance to each other must be clearly stated in order to settle any tradeoffs which appear during the design process. Most importantly, any conflicts must be resolved before any other work is done.

Once the motivations behind designing the system are understood, the architecture needs to be split into its functionality blocks (i.e., interactive consistency, synchronization, etc.). These functionality blocks provide abstract layers which are the interface between the user and the system's hardware. The requirements for each block must be outlined and then a thorough study of the theoretical work in each specific area done. Once a solid foundation has been achieved, a selection process is performed in order to choose algorithms that not only fit the problem being studied, but are also realistically implementable. The tradeoffs involved in actually implementing the protocols must then be examined in the context of the rest of the system.

Once the individual blocks are designed, an implementation needs to be done to get a more accurate picture of how the architecture performs. Any design tradeoffs that remain within the architecture can be tested to show the performance advantages and disadvantages of each side of the issue This information can be combined with the individual capabilities of each choice to make a final decision. These measurements can then be compared to other Byzantine resilient processors to give a clear idea of what will be gained and what will be lost if the architecture is later consummated.

2. Motivation

2.1. Overview

The Charles Stark Draper Laboratory has been heavily involved in fault-tolerant computing since digital computers first began to become a vital part of guidance, navigation, and control systems. Starting with the Fault-Tolerant Multi-Processor (FTMP), a project sponsored by NASA in parallel with the software-implemented fault-tolerance (SIFT) program, Draper Laboratory began investigating the issues associated with designing fault-tolerant computers to be used for controlling aircraft [HLS87]. A number of generations of processors have followed the FTMP in providing highly survivable systems. These systems were specially developed to provide extremely reliable, real-time embedded capability for critical operations. The architecture proposed in this thesis encompasses the next step in this progression.

This processor plans to provide the same high level of fault-tolerance and to support the same type of applications as its predecessors. The main difference comes from the use of authentication and the corresponding decrease in the requirements necessary to provide the required reliability. These savings have been extended into an attempt to cut the costs involved in such a specialized piece of equipment without seriously impacting performance.

2.2. Fault-Tolerance Requirements

The traditional method of designing fault-tolerant systems is to use a failure-mode and effects analysis (FMEA)-based approach. This technique analyzes likely failure modes of the system, predicts the probability of their occurring, and then designs the system to protect against those that are found to be likely to appear. The reliability of such a system is based on the probability of an unanticipated fault occurring. With reliability requirements of life- and mission-critical systems falling to less than 10^{-9} per hour, this approach has become both extremely expensive and unrealistic. A way of mathematically proving that a system will continue to operate in the presence of faults is needed.

The need for fault-masking protocols was reaffirmed when certain strange and totally unanticipated failure modes were observed. At least one in-flight failure of a triplex digital computer system was traced to an apparently Byzantine fault and the lack of architectural safeguards against such faults [MG78]. Also in circuit-switched network studies at Draper, a failure mode was observed in which a faulty node responded to commands addressed to any node [HL91]. Finally, a failed processor sending different information to other processors was observed in the SIFT computer [Pal87]. Such faults occurring in a system which is not designed to withstand them would be catastrophic.

The realization that designers could never protect against all possible failure modes resulted in what has become known as the Byzantine General Problem. The terminology and the theoretical foundation for work in this area comes from a paper by Lamport, Shostak, and Pease [LSP82] where they state:

> Reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system. This situation can be expressed abstractly in terms of a group of generals of the Byzantine army camped with their troops around an enemy city. Communicating only by messenger , the generals must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others. The problem is to find an algorithm to ensure that the loyal generals will reach agreement.

The generals correspond to processors while the messengers represent interprocessor links. [LSP82] and papers that followed [Dol83, DS83, DDS84] provide a set of requirements (lower bounds and protocols) which if obeyed make a system *f*-Byzantine resilient, guaranteeing correct operation in the presence of *f* arbitrary faults.

These requirements are necessary to guarantee the correct dissemination of single-source information, vital for the operation of the overall system. Internal tasks such as synchronization depend on the assumption that a certain set of the collected information is accurate. Information taken from external interfaces comes from redundant sources to prevent a single point of failure and must be utilized by the system. Figure 2.1 illustrates a common example of information processing used for controlling an aircraft. Redundant sensors *A*, *B*, and *C* deliver data such as wind speed and altitude to each of the processors in the system. The processors then use a fault-tolerant exchange protocol to distribute the input information so that each nonfaulty processor now has a correct set of the sensor readings. Each processor performs any required filtering and computations before performing a second fault-tolerant exchange to decide which command to deliver to the redundant system effectors. If the guidelines are followed, there is no way for *f* or fewer faults to corrupt the flow of information between the remaining nonfaulty processors.

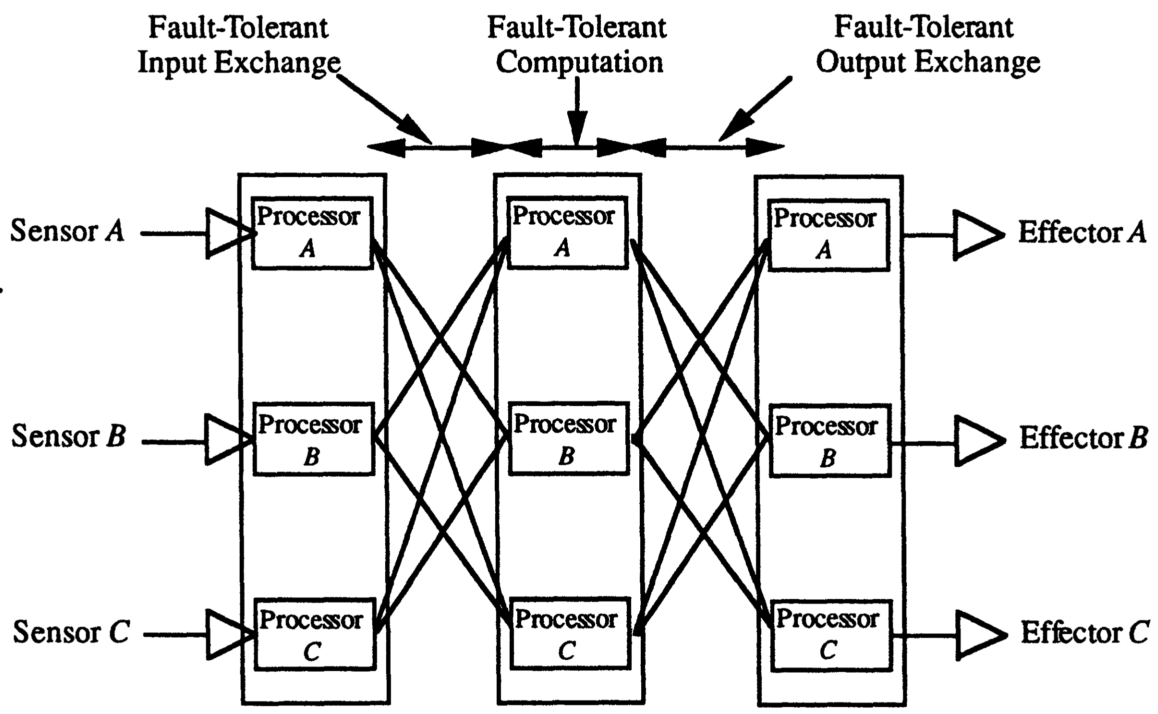


Figure 2.1 Information processing in a Byzantine resilient system.

2.3. Cost and Performance Goals

The previous Byzantine resilient systems at Draper have mainly been designed as hard real-time embedded systems to be used on aerospace vehicles. Hard real-time systems are characterized by the presence of hard deadlines where failure to meet a deadline must be considered a system fault [SAE91]. This requirement places very specific performance demands on the system. These demands must in turn be balanced with the need to keep the system small so that it can be placed in the tight space of the vehicle. Since the systems were built mostly for military or space projects, a premium was placed on achieving high-throughput combined with fault-tolerance instead of reducing costs. This need for high performance resulted in architectures which implemented as much functionality as possible in hardware, often times proprietary, and were highly optimized for a specific platform.

In this generation of processor, an attempt is being made to lower the cost of our system and thus make it available to a wider customer base. Authentication is used in the architecture because of its ability to lower the requirements in the amount of hardware needed to make the system Byzantine resilient. For example, the number of processors needed to protect against one Byzantine fault drops from four to three. These changes are discussed in more detail in Chapters 3 and 5, but reducing the number of processors by even one is a huge improvement. A secondary goal is to build the system with multi-platform capability. If the system runs on a variety of different processors, the user can then choose a package depending on the amount of money they are willing to spend and the level of performance desired.

These goals place a number of constraints on the structure of our architecture. Any use of specialized and/or proprietary equipment must be avoided due to both their high cost and compatibility problems. In fact, as much as possible of the system's functionality should be implemented in software. If the individual protocols needed to implement the required fault-tolerance are written in a high-level programming language and placed as layers between the user and hardware, porting the system from platform to platform is greatly simplified. One of the important purposes behind this thesis is to examine the effect that these policies have on the overall system.

Each protocol has a number of different design options which must be examined to find the one which fits the best with these goals. A great deal of throughput is often sacrificed when functions are done by software instead of hardware. Different solutions to problems need to be tested to determine which helps the system's performance the most. Once all of the design issues are resolved, the resulting architecture must be studied to find the sections which should be optimized, either by writing processor-specific assembly code or adding additional hardware.

3. Fundamental Building Blocks

3.1. Overview

The proposed architecture has been designed using the concept of layers. The layers serve as an interface between the user and hardware which provide the desired Byzantine resilience. The use of software to implement the needed functionality removes any dependence on specific hardware to build the system. The only contact that the layers have with the hardware is through system calls which manage the reading of physical clocks and I/O for sending and receiving messages. Each layer consists of one or two protocols which are responsible for sending all necessary information to the user. Figure 3.1 provides an abstract view of how these layers fit together.

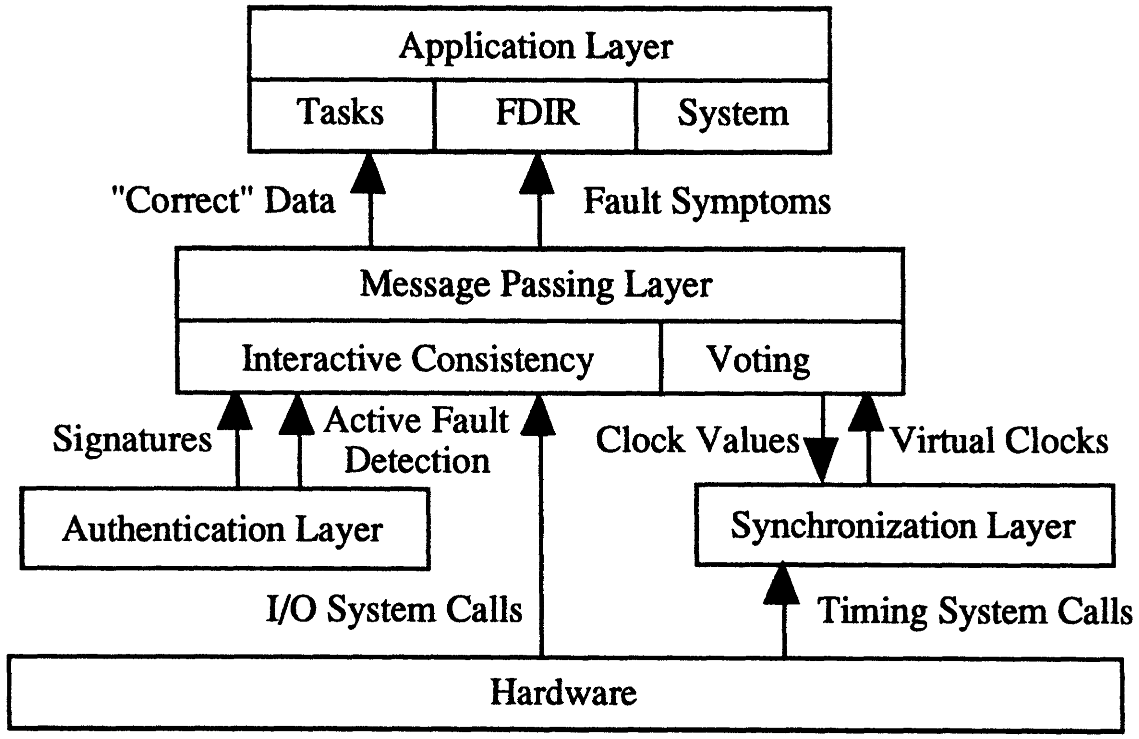


Figure 3.1 Architecture layers.

The layer placed directly beneath the Application Layer is responsible for all communications between the individual processors. Two protocols are implemented in this Message Passing Layer. The interactive consistency protocol oversees the exchanges used to distribute single-source information in a manner which masks possible faults. At the end of an interactive consistency exchange, all nonfaulty processors have a "consistent" copy of the data (the definition of consistency is discussed in Section 3.3).

The voting protocol is used to gather and compare sets of data in order to reach a group consensus value which is the same on all nonfaulty processors. This layer depends on system calls to hardware to process messages and information garnered from the lower layers.

There are two layers found below the Message Passing Layer. The first is the Authentication Layer which provides the digital signatures to be appended to the end of all messages. The authentication protocol within this layer directs the signing and verifying of messages and is responsible for detecting active faults which affect the contents of a message. The second layer is the Synchronization Layer which provides virtual clocks, built on top of the system's hardware clocks, which are synchronized to within a known skew. The clock synchronization protocol found in this layer is responsible for starting synchronization intervals and adjusting the system's virtual clocks to keep them synchronized. The protocol uses the Message Passing Layer to read the virtual clocks on other processors and system calls to read a processor's own hardware clock while fulfilling its functions.

The protocols found within these layers are the building blocks which form this architecture. Each one must be completely specified and designed before any implementation work can begin. The rest of the sections in this chapter give a general outline of each protocol. Chapters 4, 5, 6, and 7 then give an in-depth description of each problem and the final proposed solutions for each.

3.2. Authentication

Authentication forms the foundation upon which the rest of the architecture is built. All of the other protocols are affected in some way by its presence. The restrictions on the types of undetected faults which authentication provides allow a reduction in the requirements necessary for interactive consistency. The presence of signatures provides a new method of achieving group consensus. However, the need for all data to be signed before it can be sent to other processors constrains the options available for synchronization. The authentication protocol must therefore be the first one to be designed.

Authenticated protocols were developed when the need for secure computer communications first became apparent. Users realized that they required an efficient method of verifying the identity of those sending them messages. Protocols were designed to provide a wide-range of services from the establishment of secure interactive communications between principals on different machines to authenticated one-way communications for mail systems and signed communications where the origin and contents of a message could be authenticated by a third party [NS78]. Uses for this last function were seen from the very beginning of work in the field of Byzantine resilience.

The first Byzantine resilient algorithm using authentication appeared in [LSP82]. The authors realized that the ability to append a signature to the end of all messages and allow the receiver to verify the original sender and contents of the message made certain powerful assumptions about faults possible. Faults can be classified as either passive or active. Passive faults have no effect on the contents of the message and usually involve messages being delayed or blocked. Active faults are directly connected to the contents of the message. For example, a processor trying to masquerade as another processor or a noisy link corrupting messages are active faults. The use of authentication allows the designer to assume that all active faults are detected through checking the signatures and therefore do not need to be masked by a separate message passing protocol (i. e., an interactive consistency protocol). This assumption makes a drastic reduction in the complexity of Byzantine resilient systems possible.

	Unauthenticated Protocols	Authenticated Protocols
Number of processors [LSP82]	$3f+1$	$2f+1$
Connectivity [Dol83]	$2f+1$	$f+1$
Communication Rounds [DS83]	$f+1$	$f+1$
Messages [DS83]	$O(n^f)$	$O(nf)$
Voting Required [LSP82, DS83]	Yes	No
Synchronization [DDS84]	Yes	Yes

Table 3.1 Theoretical requirements for f-Byzantine resilience.

The use of digital signatures allows a reduction in both the hardware and communications necessary to provide Byzantine resilience. Table 3.1 compares the theoretical requirements for unauthenticated and authenticated protocols as a function of f, the number of faults which the system protects against. The number of processors and the connectivity between them is reduced by a factor of f. The same amount of communication rounds are needed, but the number of messages which are sent within each round has fallen. The final difference is that the processors no longer have to perform a vote on the entire message since the signatures provide an encoded version which can be compared instead. The effect these changes have on a Byzantine resilient architecture is the main motivation behind this thesis.

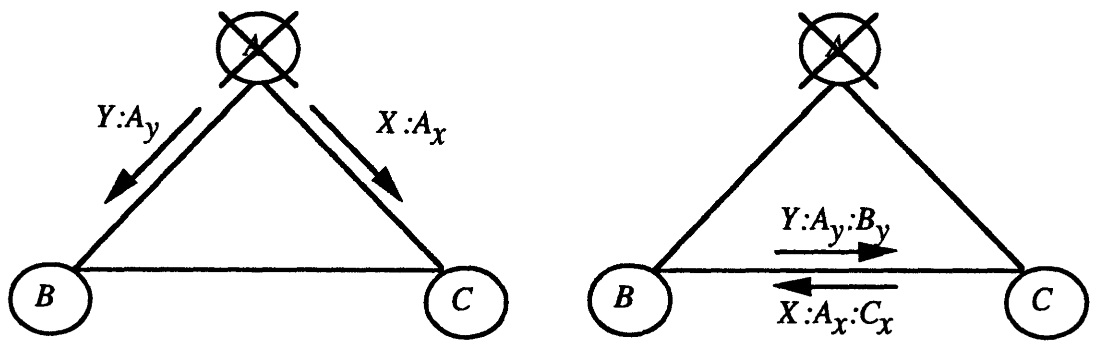
3.3. Interactive Consistency

An architecture's claim of Byzantine resilience depends on the protocol used to provide interactive consistency. This protocol is utilized whenever data appears on only one processor. Such single-source information must by exchanged using fault-tolerant message passing techniques to provide congruent copies, or data which is the same, on all nonfaulty processors. Common examples of single-source data are a reading from one redundant sensor or the value of a processor's local clock. A more exact definition of interactive consistency is a set of processors agreeing on a piece of information originating from a single source in such a way that the following Byzantine Agreement conditions are fulfilled:

Agreement: If any nonfaulty processor decides value *V*, then all nonfaulty processors decide *V*.
Validity: If a nonfaulty processor starts with an initial value *V*, then *V* is the only allowable decision by all other nonfaulty processors.
Termination: All nonfaulty processors decide on a value *V* within a known amount of time.

The value *V* which is agreed upon by all nonfaulty processors is considered to be "consistent." If the processors perform the same operation on this consistent data, they are considered to be operating "congruently."

Obtaining interactive consistency in the absence of faults is a trivial problem since the sender simply has to transmit its data to the other processors. Once Byzantine faults are introduced, the process becomes much more complicated. Figure 3.2 illustrates our problem in the presence of a commonly discussed Byzantine fault, a "two-faced" clock. In this scenario, a faulty processor sends a different value to each of the other processors. In Figure 3.2(a), faulty processor *A* sends signed messages containing conflicting data to the nonfaulty processors *B* and *C*. Authentication does not help to discover this fault since the two varying messages correspond to their signatures. If the protocol were to stop at this point, *B* and *C* have no way of realizing that they have inconsistent, or differing, information. The second round of message passing shown in Figure 3.1(b) is needed between *B* and *C* in order to discover the problem. After the second round, *B* and *C* each have a set of two different messages which have verified as originating from *A* and therefore, upon noticing a disagreement, each chooses the same default value.



(a) An undetected fault. (b) Fault discovered.

Figure 3.2 Detecting a "two-faced" clock.

A protocol must be designed to implement the required message passing and provide interactive consistency in the presence of Byzantine faults. It is vital for the architecture to guarantee that after an exchange, the information on nonfaulty processors is consistent. If a fault were to occur that caused conflicting data to be accepted on nonfaulty processors, the actions of the processors could diverge, causing a failure of the entire system.

3.4. Voting

The second protocol in the Message Passing Layer is known as the voting protocol. This protocol is responsible for calculating a group consensus value which is guaranteed to be the same on all nonfaulty processors. Periodic exchanges are performed when processors are expected to be acting congruently, or performing the same operations on identical data. The goal of these exchanges is to gather a set of values containing an element from each processor to be used in calculating the group consensus value. These voting exchanges are employed mainly by Fault, Detection, Isolation, and Recovery (FDIR) tasks to detect faults.

Voting exchanges result in a set of data with elements from each processor and a group consensus value. Since the processors are expected to be acting on congruent information, the data from nonfaulty processors should be the same as the group consensus value. FDIR tasks generate fault symptoms by comparing the group consensus value to every element in the data set and marking every processor which provided noncongruent data as faulty. Voting exchanges are also used just before an output to filter any computational faults.

The voting protocol is designed after the interactive consistency protocol because it is constrained by interactive consistency requirements. The fault tolerance claims of the entire architecture rest on the correctness of the interactive consistency protocol so the voting protocol must be structured to fit with the other protocol. Also, any requirements added by the new protocol are not allowed to come into conflict with the interactive consistency requirements. The design of the voting protocol must be done to complement the interactive consistency protocol.

3.5. Synchronization

One of the most important and complicated requirements for reliable real time systems is the need for synchronized processors. In fact, Fischer, Lynch, and Paterson show in [FLP83] that it is impossible to make an asynchronous system resilient against even one fault. If one processor stops sending messages, the rest of the system could deadlock. The other processors have no way of deciding whether the faulty processor has failed or is simply far behind the rest. The normal solution to this problem of using time-outs implies, by it very presence, some form of synchronization.

Other technical problems were discovered in the advanced fighter technology integration (AFTI) F-16 program [Mac88]. The digital flight control system (DFCS) was designed so that the three computers in the triply redundant system were not synchronized. The designers believed that computer synchronization introduced a single-point failure caused by electromagnetic interference (EMI) and lightning. The AFTI F-16 program found that the asynchronous aspect of the architecture introduced errors in the inputs due to time-skewed sampling by the different processors. An even more serious problem became apparent in the verification process. The system became untestable in that testing for each of the possible time relationships between individual processors was impossible. This attempt at asynchronous operations only served to reinforce the need for synchronization.

Synchronization requires that each nonfaulty processor have some idea of "real" time within a known skew of all other nonfaulty processors. A clock falling outside of this skew bound is considered a processor fault. Fault containment considerations make it necessary for each processor to have its own local independent clock. Using one global clock makes synchronizing the processors trivial, but also introduces a single-point failure. The problem is that no matter how accurate the individual clocks are, they still tend to drift apart over time. Clock synchronization protocols are used to correct for this drift.

Clock synchronization protocols implement virtual clocks in a way which should be invisible to the user. The method used to synchronize the clocks must be chosen based on the overall characteristics of the architecture. Another important consideration is the amount of overhead introduced by the protocol. In the SIFT computer, twelve out of sixty-six slots in each major frame were dedicated to clock synchronization and redundancy management, an overhead of 18% [PB86]. A protocol which avoids a large amount of overhead when the system is busy is preferred.

4. Authentication

4.1. Overview

The Authentication Layer provides a service upon which the rest of the architecture rests. Its authentication protocol uses digital signaturing techniques to allow a recipient to verify the authenticity of a received message with high probability. These signatures are used by the protocol to isolate the many types of active faults which can occur within the system. There are requirements, both theoretical and practical, which must be fulfilled by these signatures. The technique used to calculate the signatures must also be flexible and comprehensive enough to detect events varying from messages arriving late and outside real-time constraints to an intermediate processor repeatedly transmitting a message. On the other hand, latency is added to message processing every time a message is signed and verified, adversely affecting performance. The signaturing function must therefore be chosen carefully to meet all requirements while accounting for these problems.

Authentication protocols are defined by the type of signaturing scheme which is employed. The most common types of protocols are based on methods using key-pairs to sign and then verify a message. These protocols are classified into two groups, private-key authentication and public-key authentication. In private-key authentication, a sender generates signatures using a private-key; any receiver which wants to verify the message's authenticity must also have access to the private key. On the other hand, public-key authentication makes use of key-pairs where the sender signs the message using a private key, while the receiver applies a public key related to the private key to verify authenticity. Designing the final authentication protocol for this thesis involves analyzing the advantages and disadvantages of these two protocol types, selecting the best for this architecture, and incorporating a signaturing scheme which fulfills all of the requirements mentioned in the next section. Finally, the machinery necessary for signing and verifying messages needs to be build in order to investigate the performance issues involved in authentication.

4.2. Authentication Requirements

Before work on designing the authentication protocol can begin, the many requirements which must be met by the signatures need to be examined in detail. The signaturing scheme is an integral part of any authentication protocol. The integrity of the individual signatures determines the reliability of the entire system, since the architecture may not protect against undetected active faults which have defeated the authentication protocol. Such an undetected active fault could cause a catastrophic system failure. At the same time, attempts to make the scheme more robust often result in poorer performance. There are two main issues which need to be investigated with these factors in mind. The architecture's claim of being Byzantine resilient depends heavily on theoretical assumptions about signatures. Decisions about how strictly these assumption will be upheld must be made. Also, these assumptions need to be translated into implementation requirements for the signatures.

All Byzantine resilient algorithms using authentication assume the ability to generate signatures such that

A1: A sender's signature cannot be forged by another process.

A2: Any alteration of the sender's message can be detected.

A3: The receiver can readily verify the authenticity of a sender's signature-message pair.

These assumptions can never be totally guaranteed in an actual implementation due to the structure of digital signatures. Digital signatures consist of a finite number of bits which can always be generated randomly by a processor. The signaturing scheme must be optimized to provide the needed amount of security without affecting the rest of the system.

Even though assumption A1 cannot be ensured, it is possible to make forging the signature extremely difficult. The probability of a processor forging a sender's signature by random attempts decreases as the number of bits in the signature increases. For example, if we assume that a processor generates messages at the rate of 10^6 per hour, and a probability of system failure of 10^{-10} per hour is desired, then the probability of forging a signature needs to be 10^{-16} per message. The signature would need to consist of at least 53 bits to provide a probability of forgery of 10^{-16}. This is only a lower bound for the number of bits in the signature since this assumes random attempts to forge a signature. Assuring unforgeability to malicious attempts may require a greater signature length.

Similarly, assumption A2 cannot be guaranteed with unity probability since the generation of an n-bit signature for a k-bit message, where $n<k$, implies that there exists a signature which corresponds to at least two messages (and probably even more). Thus, a processor may change the contents of a message with a finite probability that the

changed message's signature is identical to the original message's signature. If we assume that the signaturing scheme uniformly distributes signatures over the message space, that is for all signatures v_i in the signature space Q, v_i is the signature for exactly 2^{k-n} messages, we can estimate the probability that a processor can undetectably corrupt a message, leaving the signature unchanged, to be 2^{-n}. Figure 4.1 shows an example of such a distribution. If a 2-bit signature is appended to an 8-bit message, there are only 2^2, or 4, signatures to represent the 2^8, or 256, possible messages. If a message is corrupted, there is a 2^{-2} ($\frac{1}{4}$) probability that the new message falls into the message space corresponding to the same signature. To provide a probability of undetectable corruption of 10^{-16} per message would again necessitate a signature of 53 bits and a "spectrally white" signaturing scheme.

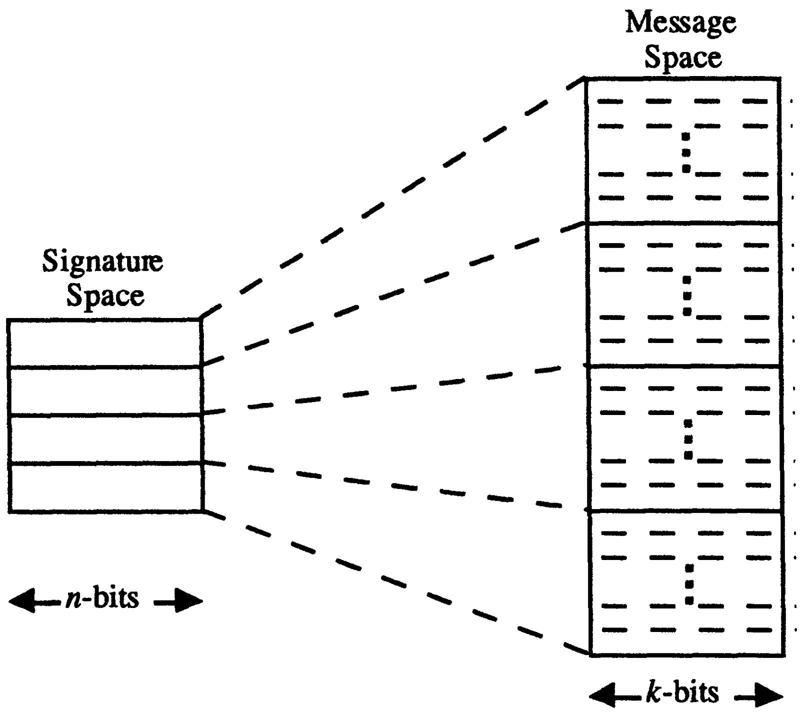


Figure 4.1 Evenly distributed signature and message spaces.

The security required by signatures is determined by the level of intelligence we are willing to attribute to faults. For many applications, we can assume that faulty processors may exhibit malicious behavior, but not to the extent of a malicious cryptoanalyst, particularly in cases in which we are using authentication to protect against random hardware failures in fast hard real-time applications. Therefore, our signaturing scheme need not be cryptographically secure but only robust against randomly malicious behavior. Due to the malicious nature of some hardware failures we still need to exercise some caution in determining a signaturing scheme.

Considerations other than the security of the signatures must be taken into account in designing a signaturing scheme. Certain information must be provided in all signatures to vitiate the many different types of active faults. First of all, signatures must be host-specific to prevent one processor from masquerading as another. In the flawed example shown below, processor *B* pretends to be processor *A* and sends "*A* says go", signing it with *S*. If the signatures are not host-specific, processor *C* would accept the message as coming from processor *A*.

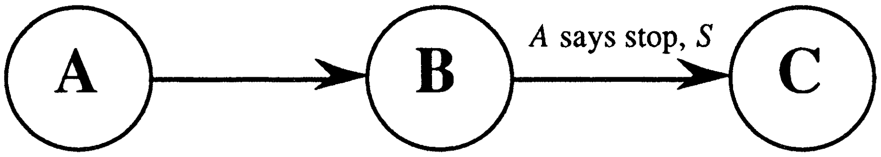


Figure 4.2 Undetected fault with a non-host-specific signature.

Host-specific signatures keep a processor from being tricked as to the origin of the message. Replaying the above scenario with host-specific signatures, if *B* sends the message "*A* says go" and appends S_b to it, *C* will realize that the message did not originate from *A*.

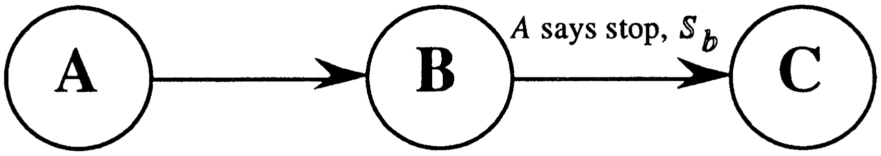


Figure 4.3 Detected fault with a host-specific signature.

Also, signatures must be message-specific in order to prevent an intervening processor from corrupting a message and then appending a correct signature to it. In the flawed example shown below, processor *A* sends “*A* says go” and signs it with S_a. If the signatures are not message-specific, processor *B* could corrupt the message to “*A* says stop,” append S_a to it, and *C* would accept the corrupted message as authentic.

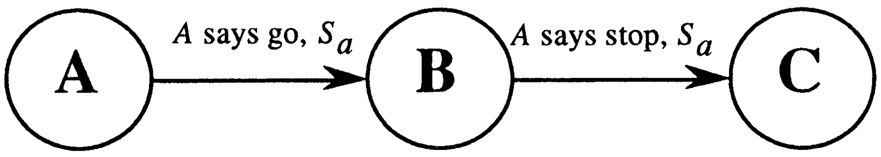


Figure 4.4 Undetected fault with a non-message-specific signature.

Message-specific signatures keep a processor from copying another's signature and appending it to a corrupted message. Replaying the above scenario with message-specific signatures, if *B* corrupts the message from "*A* says go" to "*A* says stop" and attempts to append S_a to it, *C* will detect that the signature and the message do not match and discard the message.

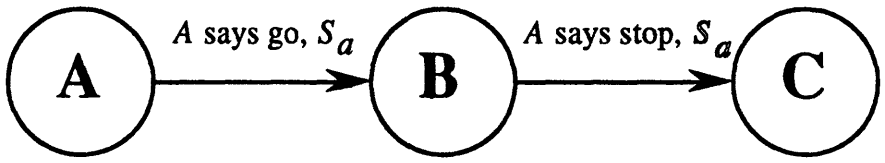


Figure 4.5 Detected fault with a message-specific signature.

However, even with host- and message-specific signatures, an intermediate processor can still erroneously and undetectably repeat a message-signature pair. In the flawed example shown below, processor *A* sends "*A* says go" followed by "*A* says stop." Processor *B* can erroneously save the first message and transmit it twice, absorbing the second message without *C* being any the wiser, since both messages received by *C* authenticate.

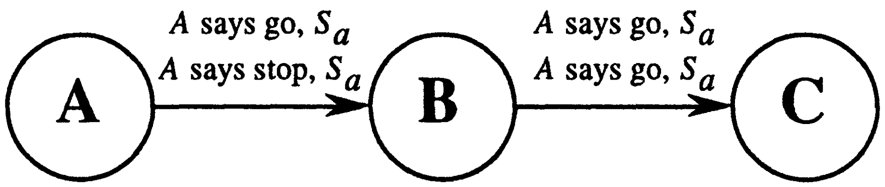


Figure 4.6 Undetected fault with a non-time-specific signature.

To eliminate this possibility, a monotonically increasing sequence number can be attached to each message by the sender. The sequence number introduces a varying message component which ensures that a relaying processor cannot undetectably replay a saved copy of a message. Such replays would be rejected by the receiver because they have identical sequence numbers. Sequence numbers also ensure that, when a source intentionally transmits two identical messages, the signatures will differ because the signature, which is calculated with respect to the message <u>and</u> the sequence number, has a spectrally white dependence on both the message <u>and</u> the sequence number.

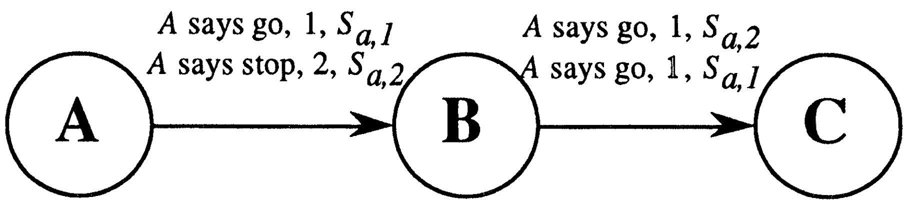

A says go, 1, $S_{a,1}$
A says stop, 2, $S_{a,2}$
A says go, 1, $S_{a,2}$
A says go, 1, $S_{a,1}$
A
B
C

Figure 4.7 Detected fault with a time-specific signature.

4.3. Authentication Protocols

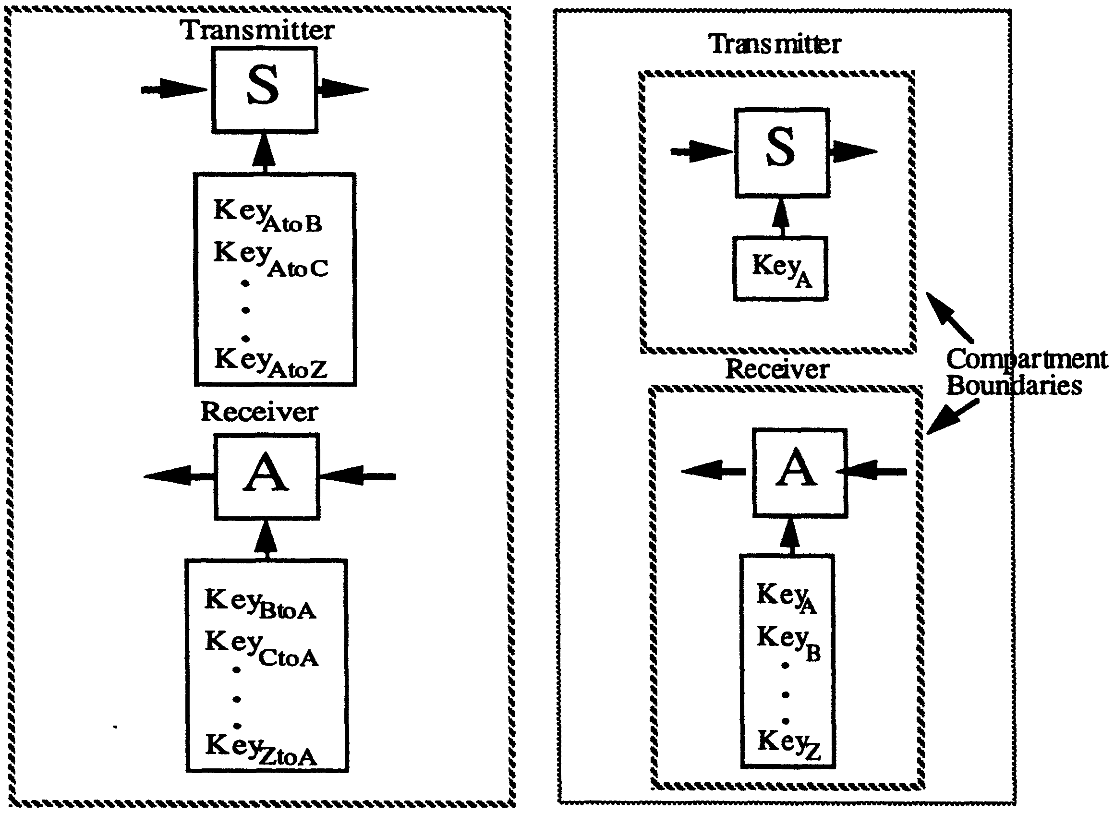
Authentication protocols are responsible for the mechanics of signing messages at the sender and verifying their authenticity at the receiver. There are two basic approaches to authentication used by such protocols, private-key authentication and public-key authentication. Private-key authentication is the traditional type of cryptosystem which has been used for centuries where anyone who wants to verify a message needs to know the key used by the signer. On the other hand, public-key authentication is of more recent origin (at least within this century), first suggested by Diffie and Hellman in [DH76]. Diffie and Hellman proposed using key pairs where one key is utilized to sign the message and the other key is used to verify both the sender and the message itself. There are advantage and disadvantages to both types which need to be weighed before the final protocol can be designed.

4.3.1. Types of Authentication Protocols

The approach for an authentication protocol implementing private (secret) keys uses a signaturing function, $v = S_k(M)$, which generates a signature v for the message M based upon a key k. Every participant that wishes to authenticate messages from a sender must possess k, the key the sender used to sign the message. A receiver i verifies a message from a sender j by computing $S_k(M)$ using the sender's key and comparing that signature with the signature appended to the message. One problem with private key authentication is that the authentication key must be identical to the key used to sign the message. Hence a receiver which is able to authenticate a message from a sender j is also able to forge outgoing messages with j's signature. Two methods may be used to prevent this scenario from occurring.

The first method is to use pair-wise common keys. Each sender j has a different key for each other node in the system. A receiver i has the key that each sender uses to sign messages sent to i. (See Figure 4.8(a).) Thus in the worst case, a faulty node can only forge messages to itself. This solution creates a number of key management problems and precludes the ability to broadcast information to all participants without attaching a separate signature for each potential recipient.

The second method to prevent a receiver from forging a message is to *compartmentalize* the receiver from the transmitter (See Figure 4.8(b).). By this, we mean that the designer must insure that no propagation of key information from receiver to transmitter is possible through the use of hardware isolation/protection mechanisms. One method for doing this would be to only allow the transmit key to be set on power-up. Again, this method creates a number of key management problems; however unlike the previous method, it does allow a node to broadcast messages without attaching a separate signature for each potential recipient.



(a) Pairwise secret key system. (b) Compartmentalized single key system.

Figure 4.8 Approaches to private-key authentication.

In public-key cryptosystems each participant i has an encoding function which generates a n-bit representation of the message, $M_{\#}$, as well a signing function D_i and an authenticating function E_i with the following properties:

B1: $E_i(D_i(M_\#)) = M_\#$.

B2: Both E_i and D_i are easy to compute.

B3: D_i cannot not be inferred from knowledge of E_i with any reasonable effort.

The nomenclature for the functions may be confusing, but it comes from the original public-key proposal in [DH76] which envisioned the public keys being used to send (and therefore encode) messages which could only be read (and therefore decoded) by a receiver with the private key. We have renamed the functions to correspond to their uses in our design, but have kept the original symbols. To apply the public-key cryptosystem scheme to a public key authentication system, sender i encodes the message M_{text} such that $M_\#$ is a n-bit number representative of M_{text}. The spectrally white encoding function is common knowledge; computation of a CRC over the message is an acceptable algorithm. The sender then uses its private signing function D_i on $M_\#$ to calculate the signature. The receiver uses the authenticating procedure E_i, which is common knowledge for all i, to verify the message. To be specific, for processor A to send a message to processor B, the following steps are taken:

1. A computes $M_\# = Encode(M_{text})$.
2. A computes the signature $S_A = D_A(M_\#)$.
3. A sends $\langle M_{text}, S_A \rangle$ to B.
4. B computes $M_\# = Encode(M_{text})$.
5. B strips off S_A and computes $E_A(S_A)$.
6. If $E_A(S_A) = M_\#$, then the message is authentic.

While public-key signature systems do not need secret keys, and broadcasts are possible, they do require the availability of suitable functions E_i and D_i which possess properties B1-B3. The public-key authentication method does eliminate the problems associated with key management and allows efficient broadcasts.

4.3.2. Proposed Protocol Design

The decision about which type of authentication protocol to employ needs to be made based on how authentication will be used by the system. This protocol is placed in a low level layer beneath the message passing protocols. It is necessary for the processors to be able to broadcast message to the other processors without incurring performance hits. Also, remember one of the goals behind this architecture is to implement as much as possible, if not all, of the functionality in software in order to cut costs. Avoiding problems with key management or the need for hardware isolation/protection mechanisms is therefore desired. These factors made public-key authentication the obvious choice for this system.

The final proposed protocol is a public-key cryptosystem based on Cyclic Redundancy Codes (CRC's) and modular inverses. CRC's provide a spectrally white means of encoding the message which is relatively simple to either code in software or build in hardware. From the evaluation discussed previously in Section 4.2, the number of bits necessary to make the probability of forgery adequate for our applications necessitates a CRC of greater than 53 bits. For ease of implementation, this would require a CRC length of 64 bits (the closest multiple of a byte to 53). The signing and authenticating functions use modular inverse keys which are two numbers, P and P^{-1} (P inverse), for which $P \cdot P^{-1} \bmod 2^n = 1$ is true where 2^n is a very large number. P acts as the private signature key while P^{-1} is the public signature key. Several examples of modular inverses are:

$$13 \cdot 5 \bmod 2^5 = 1$$
$$1033 \cdot 569 \bmod 2^{11} = 1$$
$$9294586028090703467 \cdot 3509695877449905 15 \bmod 2^{64} = 1.$$

The signing function, $D_i(M_\#)$, is $P \cdot M_\# \bmod 2^n$ while the authenticating function, $E_i(S_i)$ is $S_i \cdot P^{-1} \bmod 2^n$ where 2^n is the number of possible signature configurations (and n is the number of bits in the signature) and $M_\#$ is a n-bit representation of the message. B1 from the previous section is fulfilled since

$$E_i(D_i(M_\#)) = (P \cdot M_\# \bmod 2^n) \cdot P^{-1} \bmod 2^n = M_\# \cdot (P \cdot P^{-1} \bmod 2^n) = M_\#$$

(Please note that $(x \bmod 2^n) \bmod 2^n = x \bmod 2^n$).

B2 is also upheld because both E_i and D_i contain only n-bit multiplication and modulus functions. The steps necessary to infer D_i from E_i are discussed in the next section, but do require a large amount of effort, satisfying B3.

The protocol uses a combination of these functions to provide its signatures and then authenticate its messages. When processor A sends a message to processor B, the following steps are taken:

1. A computes $M_{\#} = CRC(M_{text})$.
2. A computes the signature $S_A = D_A(M_{\#}) = P \cdot M_{\#} \bmod 2^n$.
3. A sends $\langle M_{text}, S_A \rangle$ to B.
4. B computes $M_{\#} = CRC(M_{text})$.
5. B strips off S_A and computes $E_A(S_A) = S_A \cdot P^{-1} \bmod 2^n$.
6. If $E_A(S_A) = M_{\#}$, then the message is authentic.

4.4. Key Generation

The signing function D_i and the authenticating function E_i require a pair of modular inverse keys, P and P^{-1}, for each processor in the system. P is used by D_i as the private key and is assigned to only one processor. P^{-1} acts as the public key for E_i and a table of all public keys is maintained on every processor. The only key management required by these pairs is at compilation time when the private key initialization routines for each individual processor are placed in separate files to ensure isolation.

The key pairs are generated using an extended version of Euclid's algorithm found in [Knu69]. Euclid's algorithm is a famous method for finding the greatest common denominator (gcd) of two numbers, u and v. Knuth extended it to calculate two more integers, u' and v', such that

$$u \cdot u' + v \cdot v' = \gcd(u,v), \tag{4.1}$$

at the same time $\gcd(u,v)$ is being found. In order to use this algorithm, we note that if

$$P \cdot P^{-1} \bmod 2^n = 1 \tag{4.2}$$

(the definition of modular inverses), then by the definition of the mod function,

$$P \cdot P^{-1} = 2^n \cdot v' + 1, \text{ or} \tag{4.3}$$

$$P \cdot P^{-1} + 2^n \cdot v' = 1 \tag{4.4}$$

where v' is some constant. Therefore if 2^n is substituted for v and a prospective n-bit private key is chosen at random for u, Knuth's algorithm solves

$$P \cdot P^{-1} + 2^n \cdot v' = \gcd(P,2^n). \tag{4.5}$$

If $\gcd(P, 2^n) \neq 1$, then we know that P is not co-prime with respect to 2^n and must try another key. Also, we must discard P^{-1} if it is negative. Otherwise, we may use the key pair, P and P^{-1}. Appendix A contains a list of 64-bit modular inverses generated with this technique.

4.5. Cyclic Redundancy Codes (CRC's)

Cyclic Redundancy Codes, or CRC's, provide the encoding function which is a vital part of our public-key authentication protocol. Encoding functions are responsible for calculating a message-specific sequence of bits which are spectrally white and can be used to detect errors within the actual message. CRC's are a fast and efficient method which are well known for their ease of implementation, both in hardware and software. While not cryptographically secure, the probability of a faulty process randomly forging the correct signature decreases as the CRC length increases, at a rate of 2^{-n} for a n-bit CRC.

CRC's come originally from finite field theory. Sequences of bits are defined as binary polynomials, where each bit represents a coefficient. For example, the five-bit sequence, 10101, would be associated with the fourth order polynomial, $r(x) = 1 + 0 \cdot x + 1 \cdot x^2 + 0 \cdot x^3 + 1 \cdot x^4 = 1 + x^2 + x^4$. When a message is represented by such a polynomial, the CRC of the message is defined as the remainder of the modulo-two division of the message polynomial by a particular CRC generator polynomial [Gal90]. Figure 4.9 shows an example of how a CRC is generated. The CRC-16 generator polynomial, $g(x) = 1 + x^2 + x^{15} + x^{16}$, is used to encode the message 101011. The remainder of the modulo-two division, $s(x)$, is 0111110100000000 which becomes the CRC.

$$\begin{array}{r|l}
 & \quad x^5 + x^2 + x \\
\cline{2-2}
x^{16} + x^{15} + x^2 + 1 & x^{21} + x^{20} + x^{18} + x^{16} \\
 & \underline{x^{21} + x^{20} + x^7 + x^5} \\
 & \quad x^{18} + x^{16} + x^7 + x^5 \\
 & \quad \underline{x^{18} + x^{17} + x^4 + x^2} \\
 & \qquad x^{17} + x^{16} + x^7 + x^5 + x^4 + x^2 \\
 & \qquad \underline{x^{17} + x^{16} + x^3 + x} \\
 & \qquad\qquad x^7 + x^5 + x^4 + x^3 + x^2 + x
\end{array}$$

$s(x) = x^7 + x^5 + x^4 + x^3 + x^2 + x$

Figure 4.9 Example of CRC encoding [RG88].

The following sections are a condensed overview of the CRC techniques used for this protocol. If the reader wishes a more in-depth look into the subject, an article by T. V. Ramabadran and S. S. Gaitonde, titled "A Tutorial on CRC Computations," [RG88] is highly recommended. The most important component in implementing CRC's is the generator polynomial since this polynomial solely determines the error detection properties of CRC's. A detailed explanation of these capabilities and how a 64-bit generator polynomial was derived is found below. This is followed by a description of possible implementation techniques.

4.5.1. Error Detection and Generator Polynomials

The error detection capability of a CRC is controlled by the choice of generator polynomial. The issues involved can be better understood if a corrupted message is represented by the $(n+k)^{th}$ order polynomial, $r(x) = v(x) + e(x)$, where *v(x)* is the original message of *k* bits of data with a *n*-bit signature and *e(x)* contains the bits in error ($e(x) = 0$, if the message is uncorrupted). The message is examined for errors by dividing *r(x)* by the generator polynomial, *g(x)*, and checking for a zero remainder. The only way errors can go undetected is if the error polynomial, *e(x)*, is also divisible by *g(x)*.

Generator polynomials can be specifically tailored to detect certain kinds of errors. Consider a single error pattern represented by $e(x) = x^i$ for some *i*, $0 \leq i \leq n+k-1$. If *g(x)* has more than one nonzero term, it does not divide *e(x)* evenly and therefore detects all single bit errors. More errors can be detected by using a generator with $(1+x)$ as a factor. The resultant CRC's will then also have $(1+x)$ as a factor which always leads to an even number of terms. If *g(x)* and *v(x)* have even parity, all odd number errors are detected. Now, consider a double error pattern $e(x) = x^i + x^j = x^i(1+x^{j-i})$ for some *i*, $0 \leq i \leq n+k-2$ and *j*, $i+1 \leq j \leq n+k-1$. If *g(x)* does not have *x* as a factor and if it does not evenly divide $[1+x^{j-i}]$ for $1 \leq j-i \leq n+k-1$, we can detect all double errors.

Another class of important errors is known as burst errors. A burst error of length *b* is any error pattern for which the number of bits between the first and last error is *b*. Let the generator polynomial be of the form $g(x) = 1 + g_1x + \ldots + g_{n-1}x^{n-1} + x^n$ where the coefficients $g_1, g_2, \ldots, g_{n-1}$ can be either 0 or 1. In other words, *g(x)* has a degree of *n* and is not divisible by *x*. Any burst error of length *n* or less can be represented as $e(x) = x^i(1 + e_1x + \ldots + e_{n-1}x^{n-1})$ for some *i*, $0 \leq i \leq k$, where the coefficients $e_1, e_2, \ldots, e_{n-1}$

can be either 0 or 1. Such a polynomial is not evenly divisible by *g(x)*, and therefore a burst of *n* bits can be detected. Now consider a burst error of length $(n+1)$ represented by $e(x)=x^i(1+e_1x+...+e_{n-1}x_{n-1}+e_nx^n)$. Of the 2^{n-1} possible error patterns of this form, only one, $e(x)=x^ig(x)$, is undetectable. Therefore, the probability of an undetected burst error of length $(n+1)$is only $2^{-(n-1)}$. A similar analysis finds that the fraction of undetected burst errors of length greater than $(n+1)$ is 2^{-n}.

The above description covers only one of the many capabilities that CRC's can provide. Generator polynomials can also be chosen to allow certain errors to actually be corrected. The problem is that deriving these error-correcting codes becomes very complicated and time-consuming when expanded to a large number of bits. CRC's normally do not go beyond 32 bits (16 bits is the most common length). In order to test the viability of using 64-bit signatures for the reliability reasons mentioned in Section 4.3.2, a 64-bit generator polynomial needs to be derived. Since the only purpose for CRC's in this proposed architecture is to detect errors and thus active faults, determining the generator polynomial concentrated only on error detection properties. The optimal structure of such a *n*-bit generator polynomial is

$$s(x)=(x+1)(1+p_1x+...+p_{n-1}x^{n-1}), \tag{4.6}$$

where the $(1+p_1x+...+p_{n-1}x^{n-1})$ term is a primitive polynomial. A polynomial of degree *m* is primitive if and only if it divides X^n-1 for no *n* less than q^m-1 and is an irreducible polynomial in that it is not divisible by any polynomial of degree less than *m* but is greater than zero [PW72]. The $(x+1)$ term allows the polynomial to detect all odd number errors while the $(n-1)^{th}$ order primitive polynomial increases the size of messages for which double errors can be detected. E. J. Watson in [Wat62] compiled a list of primitive polynomials of degree *n*, $1\le n\le 100$. The primitive polynomial of degree 63 multiplied by $(x+1)$ gives the following generator polynomial:

$$s(x)=(x+1)(1+x^{63})=1+x^2+x^{63}+x^{64}. \tag{4.7}$$

4.5.2. CRC Implementation Techniques

One of the main advantages of using CRC's to encode messages is the ease of implementing the function. Traditionally, CRC's have been done in hardware using a simple shift register to process the data bit by bit. Software implementations though have found handling the message on a byte- or even word-level to be easier and faster. This section gives a simplistic description of how to compute CRC's in hardware. A much more in-depth look into the issues involved in hardware implementation can be found in [Gal90] or [Lee81]. Software CRC algorithms are then covered in more detail since the proposed architecture involves one of these choices.

Generating CRC's in hardware can be done using very basic, low-level circuitry. The computation of CRC's involves the modulo-two division of the message by a generator polynomial in order to calculate the remainder. The hardware equivalent of a modulo-two division is a linear feedback shift register (LFSR). Figure 4.10 shows the LFSR circuit for the CRC-16 generator polynomial. To encode the message, the circuit is first initialized with a seed value, usually all zeros or ones for error detection purposes. The message is then fed into the register from the right, bit by bit. The bits remaining in the register when the message has gone completely through are the CRC. Decoding operates in the same manner, with the message and appended CRC being entered into the register. If the remainder is zero, no errors have occurred. The biggest advantage of using hardware for calculating CRC's is in performance. Messages can be encoded by a shift register "on-the-fly", or in parallel with other processing, causing little extra delay. The problem is that specific hardware dedicated to generating CRC's goes against our design and cost goals of providing a software-based architecture.

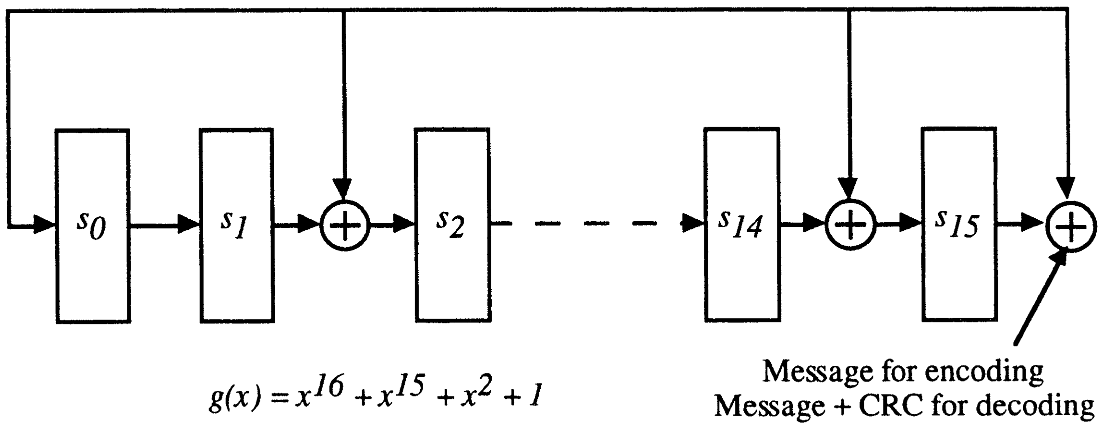

s_0
s_1
s_2
s_{14}
s_{15}
$g(x) = x^{16} + x^{15} + x^2 + 1$
Message for encoding
Message + CRC for decoding

Figure 4.10 LFSR circuit for the CRC-16 generator polynomial.

The technique used by a LFSR to process the message bit by bit is easily performed in software. The problem is that the sequential nature of software prevents generating the CRC with computation on-the-fly. A delay is present, adding to performance overhead, whenever a message is signed and verified. Even worse, this delay increases as the size of the message grows. This latency has caused software algorithms to move away from processing data bit by bit to handling it as bytes or even words. Generating CRC's in larger blocks speeds up performance by decreasing the number of times CRC operations are needed.

Using bytes and words instead of bits removes a number of unnecessary steps in finding a CRC, but the calculations that are left still take time. A faster alternative is to precompute the values for different seeds and place them into a CRC lookup table. The largest timing overhead is therefore moved to a one-time delay during initialization. Once the table values have been computed, the CRC for a message is calculated using constant size blocks with the following steps:

1. Initialize a CRC register the size of the desired digital signature to all zeros.
2. EXOR the input block with the value in the register.
3. Shift the CRC register once to the right for each bit in the input block.
4. Look up the value corresponding to the block in the CRC register and EXOR the CRC register with it.
5. Repeat steps 2 to 4 until the end of the message is reached.

This technique requires only shifts, a table lookup, and two EXOR's for each block of data.

The different options for using CRC lookup tables balance performance requirements against storage capabilities. The number of entries in the lookup table depends on the block size used by the algorithm, requiring 2^e entries for a e-bit block. A bytewise table algorithm requires twice as many lookups as an algorithm using short words (16 bits), but only needs 256 (2^8) entries versus 65,535 (2^{16}) entries. If memory is a definite restriction, a reduced lookup table can be used. The reduced lookup table contains an entry for each bit in a data block where that bit equals one and the other bits are zeros. The values found in the full tables are then calculated using sums of the entries in the reduced table, depending on which bits are set in the seed. This method requires e entries for a e-bit block, but utilizes additional operations to calculate the CRC, meaning performance suffers.

With the fall in the price of memory, cost constraints no longer solely determine how much memory is available in systems. In the proposed architecture, the main limiter on the amount of memory is the small space requirement associated with embedded systems. This change in emphasis has allowed designers to balance performance issues against memory requirements, instead of concentrating solely on saving memory. Of the

CRC lookup table options mentioned above, the reduced lookup table method uses the least memory, but has definite performance problems. On the other hand, the space occupied for tables with word-sized or greater blocks seems excessive. The optimal balance of memory versus performance thus appears to be using byte-size blocks.

4.6. Protocol Implementation Results

The use of authentication introduces a new form of performance overhead to message passing which must be thoroughly explored. Every time a message is exchanged using authentication, it is signed by the sender and then verified by the receiver. This process has a constant adverse effect on the performance of the system. Therefore, the entire authentication layer needs to be implemented in software so the various performance issues can be examined. This implementation was completed using the C programming language and then run on a Sun workstation. Overhead measurements were taken using timer system calls at points in the code. The use of these calls added to the timing results, but an assumption is made that the extra overhead is constant over the various runs.

The main determiner of the amount of overhead added by authentication is the length of the signatures. Up to this point, the only time signature lengths were introduced as a factor was in the discussion of signature security (See Section 4.2). In the example given, it was found that a signature needs to consist of at least 53 bits to provide a probability of forgery of 10^{-16}/message. In support of this conclusion, a 64-bit CRC generator polynomial was derived in Section 4.5.1. The issue of signature length now needs to be examined with performance in mind. The final decision on what length of signature to use must balance both security and performances requirements.

The authentication protocol is built using three functions. The encoding function generates CRC's which act as a n-bit representation of a k-bit message, where $n < k$. A signing function then signs this CRC with a private key found on only one processor to complete the digital signature. When the message reaches the receiver, the encoding function is used again to generate the CRC of the data. The appended signature is then combined with a public key related to the sender's private key by an authenticating function to verify the origin and contents of the message. Constructing these functions implements the entire protocol.

The first step in measuring performance is to implement the encoding function using the varying length generator polynomials shown in Table 4.1. These generator polynomials were chosen to span, in multiples of a byte, from 8 to 64 bits. The LRCC-8 polynomial is a LRC (Longitudinal Redundancy Code) normally used for simple parity checks such as a mod-2 sum of bytes. The CRC-16 polynomial is used all over the world, found in such protocols as the Bisync (binary synchronous) protocol, while the Ethernet polynomial is employed in local area networks [RG88]. Implementing the protocol's functions with these polynomials allows a comparison of the performance tradeoffs involved in the different lengths of signatures.

Title	Generator Polynomial
LRCC-8	x^8+1
CRC-16	$x^{16}+x^{15}+x^2+1$
Ethernet	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}$ $+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$
64-bit	$x^{64}+x^{63}+x^2+1$

Table 4.1 List of varying length generator polynomials.

The various implementation techniques for generating CRC's are discussed in Section 4.5.2. The final choice for this architecture is a CRC lookup table using one-byte input blocks. This method fulfills the requirement of implementing functionality in software while providing the best balance of performance optimization versus memory needs. An important property of using these CRC lookup tables is that a large amount of performance overhead is moved to an initialization period when the entries in the CRC lookup table are calculated. Table 4.2 compares the different properties of varying generator polynomials during this CRC initialization interval. The performance overhead due to initialization remains relatively constant for the first three generator polynomials. The only significant time difference comes when the generator polynomial is lengthened to 64 bits. Determining the memory overhead involves multiplying the number of entries in the table (256) by the number of bytes in each entry. This means that memory

Generator Polynomial	Initialization Overhead (usec)	Memory Overhead (Bytes)	Probability of Forgery (per message)
LRCC-8	620	256	.0039
CRC-16	650	512	.000015
Ethernet	629	1024	$2.33x10^{-10}$
64-bit	795	2048	$5.42x10^{-20}$

Table 4.2 Implementation properties of CRC initialization.

overhead is doubled each time the generator polynomial is extended by a byte. The probability of forgery is simply the probability that a digital signature is randomly forged correctly, or 2^{-n} for a n-bit signature. This means that there is a exponential decrease in the probability of forgery as the length of the signature grows.

The most important performance overhead in generating CRC's is the time needed to actually calculate the final CRC for each message. This time is dependent on the length of both the signature and the message. Section 4.5.2 describes the steps necessary to encode a message using a CRC lookup table. Figure 4.11 plots the timing measurements for generating varying length signatures for k-bit messages, with k increasing exponentially at a rate of 2^i for $i = 1 \ldots 10$. Two important facts are apparent from this graph. First of all, the time necessary to generate CRC's of all lengths increases linearly as the messages become longer. Secondly, the overhead associated with CRC's of 8, 16, and 32 bits is basically the same while the overhead for the 64-bit CRC's jumps significantly higher, becoming almost twice as much at points. This result is explained by the fact that the machine used to run the CRC code operates on 32-bit long word boundaries. The arithmetic for 32 bits or less involves the same operations, while the 64-bit operation take twice as many computations.

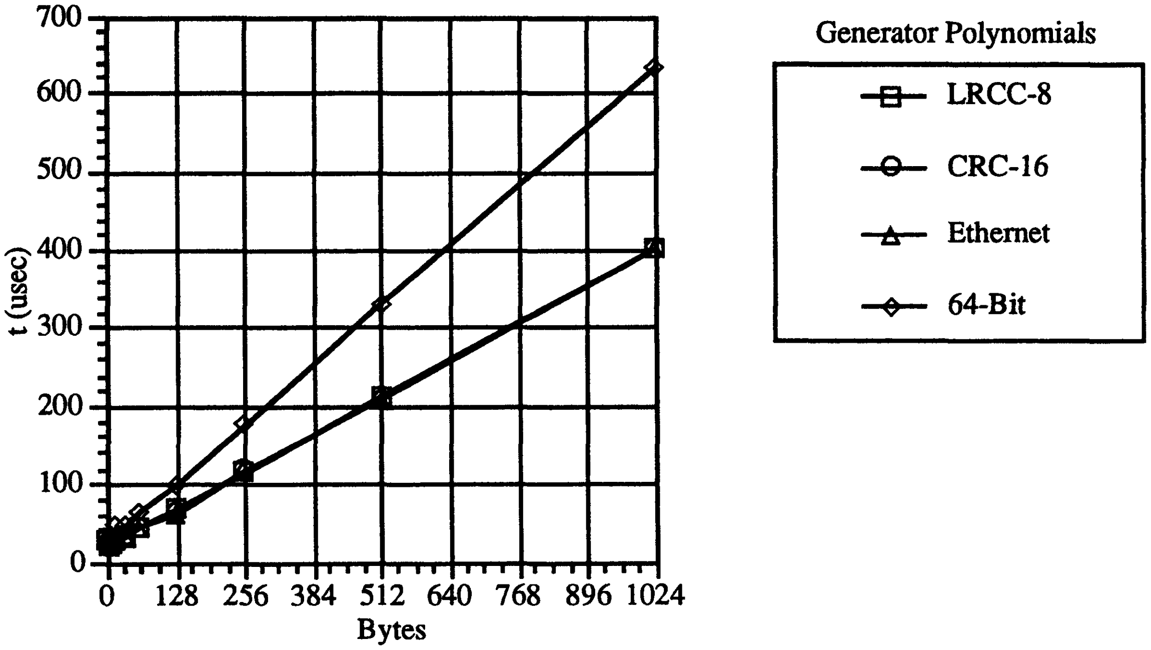


Figure 4.11 Message encoding performance overhead.

The only difference between the signing and authenticating functions are the keys used in generating the final result. Both functions are implemented using a combined multiply and modulus function. The signing function multiplies a n-bit CRC by a n-bit private key, keeping the last n-bits as the final signature. The authenticating function takes this signature and multiplies it by a n-bit public key, keeping the last n-bits to compare with the CRC of the message to verify its origin and contents. For the first three digital signature lengths, the code for these functions is very simple. When a variable of a specific length is assigned the product of two variables of the same length, a modulus function is implied. Therefore, the only code needed is a multiplication statement. This technique works so long as the signature length is equal to or less than the machine's word boundaries. Implementing the 64-bit signing/authenticating function on a machine operating on 32-bit boundaries is more complicated. Long multiplication must be carried out in units of 16-bits, with careful tallying of the carries. Table 4.3 shows the overheads for the different signing/authentication functions. Once again, the overhead associated with digital signatures of one, two, and four bytes is relatively equal. The added complexity of the 64-bit function is confirmed by the significant amount of extra overhead involved.

Generator Polynomial	Signing/Authenticating Overhead (usec)
LRCC-8	18
CRC-16	20
Ethernet	20
64-bit	56

Table 4.3 Signing/authenticating performance overhead.

Deciding the best signature length for use in the proposed architecture involves a tradeoff between reliability and performance. As the example in Section 4.2 shows, any decision based completely on minimizing the probability of a forged signature going undetected would choose a length of 64 bits. The problem is that the process of generating the 64-bit signature has definite performance problems on today's computers. It is perfectly reasonable to expect a processor to be able to perform 32-bit multiplication in one line of code, but 64-bit multiplication is part of the future. The performance overhead introduced by authentication, found with every message exchanged, is too significant to be ignored. The optimal choice for signature length is therefore 32 bits which has a probability of forgery of $2.33x10^{-10}$/message while minimizing the amount of time needed to sign and authenticate messages. In addition to using the 32-bit signature length, the operations used by the functions making up the authentication protocol need to be optimized as much as possible. The computations needed by each routine are simple enough to make writing their code in the assembly language of the processor upon which the system is running viable.

5. Interactive Consistency

5.1. Overview

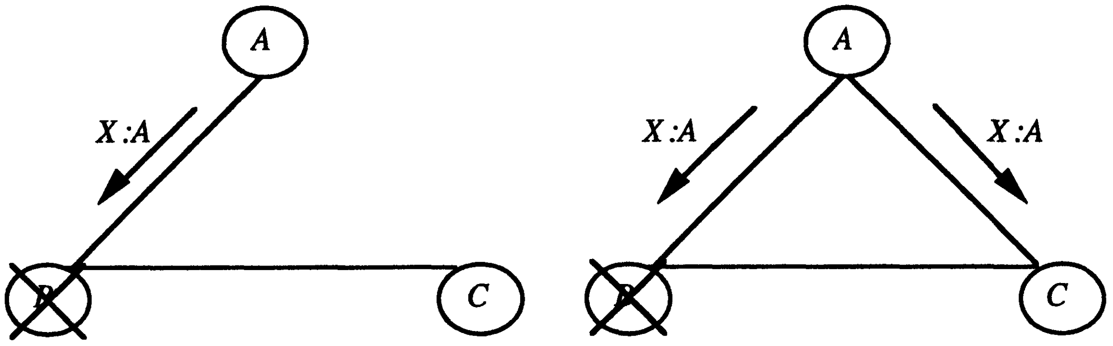
The interactive consistency algorithm is found in the Message Passing Layer, the layer responsible for distributing all data between processors. Specifically, this protocol implements exchange functions which pass single-source information to all nonfaulty processors such that the Byzantine Agreement conditions discussed in Section 3.3 hold in the presence of Byzantine faults. The final design needs to balance the goal of providing a low-cost architecture against having reasonable performance capabilities.

This chapter first examines in detail the many requirements which must be fulfilled by the interactive consistency protocol. The next step is to explore the algorithms which provide the desired properties and any design issues associated with implementing the algorithms. The final section contains an outline of the final proposed protocol design, including algorithms for the routines which make up the protocol.

5.2. Interactive Consistency Requirements

An accusation leveled against Byzantine resilient architectures is that the level of complexity necessary to mask possible faults is too expensive and complicated. Surprisingly, the requirements needed to provide interactive consistency in the presence of Byzantine faults are quite straight forward. First of all, the designer chooses the number of faults, f, that the system needs to be able to withstand. Once this decision is made, the requirements are unambiguous functions of this parameter. The only other factor that affects these results is whether or not digital signatures are used. Since authentication is an integral part of this design, all of the bounds discussed below assume that messages are signed.

The amount of hardware necessary in a Byzantine resilient system is determined by f. When authentication is used, it is possible to reach interactive consistency among all nonfaulty processors so long as the total number of processors in the system, N, is such that $N > f$ [LSP82]. This bound is usually increased to $N > 2f + 1$ to provide a majority of nonfaulty processors. Without a majority, the system operates correctly in that the Byzantine Agreement conditions given in Section 3.3 are fulfilled, but there is no way of telling from outside the system which processors are correct. An additional hardware requirement is that there always exist at least one path between a pair of nonfaulty processors that does not depend on a faulty processor. For a system using the majority processor bound of $N > 2f + 1$, this means that each processor must be connected to every other processor by at least $f + 1$ disjoint paths. Figure 5.1(a) shows how one fault could prevent interactive consistency with the correct number of processors but not enough connectivity. With only one path between processors A and C, a faulty processor B could block delivery of messages between the two nonfaulty processors. The configuration shown in Figure 5.1(b) solves this problem by adding a second path between processors A and C which bypasses processor B.

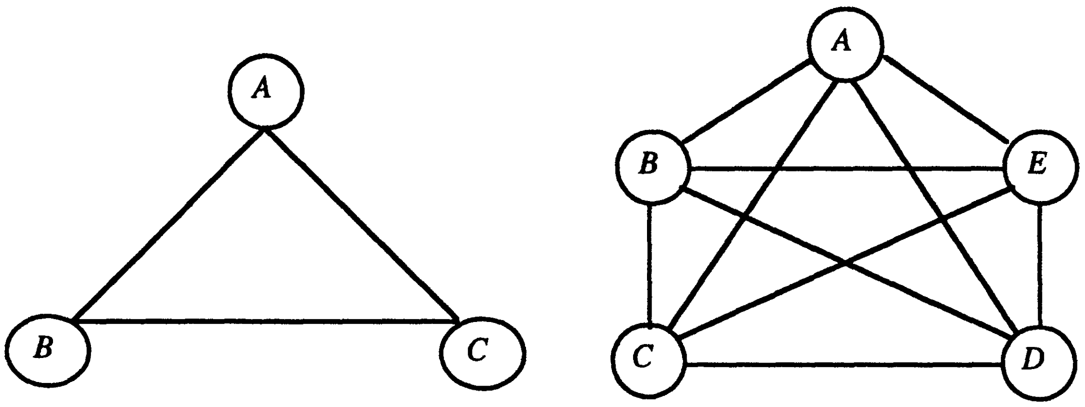


(a) Connectivity of one. (b) Connectivity of two.

Figure 5.1 Connectivity requirement.

The third interactive consistency requirement must be provided by the message passing algorithm implemented by the protocol. Each interactive consistency exchange must consist of at least $f + 1$ rounds, where rounds are defined as an interval of time in which processors chosen by the algorithm transmit messages to other processors and all processors read any values arriving on their links [DS83]. This requirement is necessary to mask faults such as the "two-faced" clock shown in Figure 3.2. The sender fault is not detected until the second round when the nonfaulty processors exchange the values that they received in the first round. Also, if the fault is not at the sender, the $f + 1$ rounds guarantee that at least one correct message will arrive at the other nonfaulty processors over the $(f + 1)^{th}$ disjoint path from the sender. The final requirement says that all of the nonfaulty processors must be synchronized to within a known skew of each other. The mechanics of providing synchronized clocks are implemented by the clock synchronization protocol discussed in Chapter 7. For now, the interactive consistency protocol is designed assuming that the synchronized clocks are present.

The number of faults that the system must mask determines the size and complexity of a Byzantine resilient system. The first fault scenario which must be examined is a specific number of faults occurring simultaneously. This is the quantity, f, which provides the lower bounds described above. Increasing f is an expensive proposition. Designing a system to mask two simultaneous faults instead only one requires five processors with a connectivity of three versus three processors and a connectivity of two. Figure 5.2 illustrates the difference in complexity caused by increasing the number of simultaneous faults that are protected against by just one.



(a) One fault. (b) Two simultaneous faults.

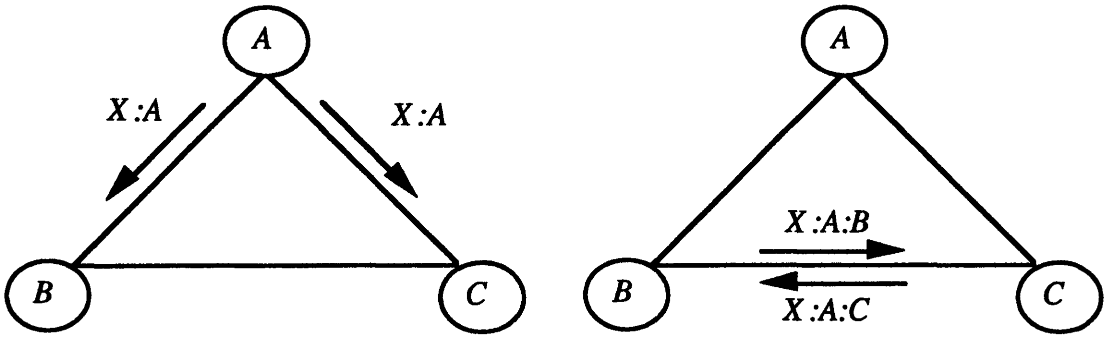
Figure 5.2 Protecting against an increasing number of simultaneous faults.

One of the main advantages of using an authenticated protocol for interactive consistency is the graceful manner in which the system can degrade. If fact, the protocol fulfills the Byzantine Agreement conditions even if only one processor remains nonfaulty. If the surviving processor gets conflicting data, it simply chooses a default value. In a practical system which needs to provide information to external entities (i.e., directions to a flight control system), more care is needed. First of all, the original system must contain enough processors to meet the majority processor bound of $N > 2f + 1$. If more than f simultaneous faults occur, the system fails. If f or fewer faults appear at the same time, each faulty processor and its adjoining links need to be isolated from the rest of the system. The goal is to always keep a majority of nonfaulty processors. So long as more than one processor remains nonfaulty, majority decisions can still be made if another processor becomes faulty. When only two processors are left and they are in conflict, additional steps, usually external, are needed to decide which processor is faulty and remove it from the system.

One of the motivations behind designing the subject system is to provide a low-cost alternative to pre-existing Byzantine resilient systems. This need for low-cost pushes the design towards using the minimal configuration shown in Figure 5.2(a). If a processor becomes faulty, it can be removed from the system while the remaining two processors continue to provide interactive consistency. If another processor starts exhibiting faulty behavior, the sole nonfaulty processor will continue to operate correctly, but the fault must be isolated before any external decisions can be made.

5.3. Byzantine Agreement Algorithms

Lamport, Shostak, and Pease provided an algorithm, called *SM(m)*, for achieving Byzantine Agreement using authentication in [LSP82]. This algorithm has since been optimized and expanded for different situations, but is still the foundation of work in this area. The algorithm is given below exactly as found in [LSP82]. The commander refers to the processor which is actually sending the message while the lieutenants are the processors receiving the message. Basically, in the first round, the general signs his message and sends it to all of his lieutenants. The lieutenants then add their signatures and relay the message to everyone who has not signed it yet. The algorithm is stated more precisely below, letting $x{:}i$ denote a value signed by processor i and V_i be the final set of values.



(a) First round. (b) Second round.

Figure 5.3 Message passing using *SM(m)*.

Algorithm SM(m) [LSP82]

Initially, $V_i = \{\}$.

(1) The commander signs and sends his value to every lieutenant.

(2) For each i,

- (A) If Lieutenant i receives a message of the form $v{:}0$ from the commander and he has not yet received any order, then
  - (i) he lets V_i equal $\{v\}$;
  - (ii) he send the message $v{:}0{:}i$ to every other lieutenant.

(B) If Lieutenant i receives a message of the form $v{:}0{:}j_1{:}j_2{:}\ldots{:}j_k$ and v is not in the set V_i, then

(i) he adds v to V_i;

(ii) if $k < m$, then he sends the message $v{:}0{:}j_1{:}j_2{:}\ldots{:}j_k{:}i$ to every lieutenant other than $j_1,\ldots,j_k$.

(3) For each i: When the Lieutenant i will receive no more messages, he obeys the order, $choice(V_i)$.

An improved version of this algorithm developed by Dolev and Strong in [DS83] uses certain characteristics of digital signatures to give a specific instantiation for the *choice*() function used in deciding the final value. Authentication allows a processor to discover whether a message has been corrupted since leaving the sender and restricts undetected faults to passive ones such as a failure to relay messages. The new algorithm discards all messages that:

1) do not match their signatures,
2) do not have the sender as the first signature, or
3) carry values the same as those already seen.

If, during $(f+1)^{th}$ round, a processor extracts a value, the other f processors are faulty and all of the nonfaulty processors now receive a correct value. If at the end of $f+1$ rounds, a processor has only extracted one value, all of the nonfaulty processors have that value. If any of the nonfaulty processors have extracted two values, all of the nonfaulty processors have extracted two and decide on a sender fault.

5.4. Interactive Consistency Design Issues

The actual implementation of the above algorithm and decision-strategy must take into account a number of different issues. The information provided by the interactive consistency protocol can be used for other purposes than simply fulfilling the Byzantine Agreement requirements. If the design is done correctly, much of the data needed by a Fault Detection, Identification, and Recovery (FDIR) task to detect where faults are occurring can be recovered from the message passing intervals. Also, a greater emphasis needs to be placed on performance issues other than the total number of messages sent by the protocol, taking into account the strengths and weaknesses of the architecture. These issues bring to light some design options which need to be examined. Decisions must be made on the type of message passing to be used, the number of signatures which are appended to messages, the format for each message, and how to keep track of sequence numbers.

5.4.1. Message Passing

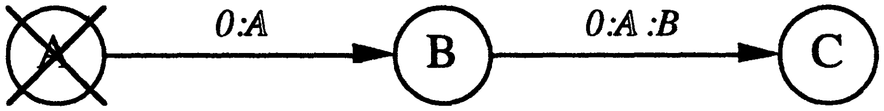
Much of the work in developing new Byzantine resilient protocols has been in finding algorithms which use the least number of messages to achieve agreement. An example of such an optimization is found in *SM(m)* where, after the first round, the processors only relay a message to those that have not already signed it. In previous Draper Byzantine resilient systems, the time taken by a message passing protocol is determined by the number of rounds utilized. Since the processors are all connected via disjoint paths with separate message processing hardware provided for fault tolerance, the number of messages within a given round do not affect the latency of the overall system. So long as the lower bound of $f+1$ rounds is met by the protocol, little can be done to optimize the speed of the interactive consistency exchanges. Therefore, in these hardware-based architectures, the extra hardware required to decide which processors need to send messages each round actually hurts performance. Instead, a broadcast protocol, where each processor automatically sends a message to all other processors and then only uses those messages that it needs, requires less overhead and provides more flexibility to the system.

Broadcast protocols have a number of advantages over the sending of messages on a point-to-point basis suggested by *SM(m)*. In a mission-critical system where the sender is trying to distribute important information, every attempt must be made to overcome what may be a transient fault. A broadcast protocol allows the system more versatility in recovering from a sender fault as well as isolating faults. For example, if the nonfaulty processors all receive corrupted data from the sender in the first round and then valid data in the second, the fault is probably transient and the default decision should be to request retransmission of the information. The added information could also be used for FDIR. For example, if the sender receives a null message in the second round, FDIR would know that either a sender fault has occurred, or the fault is in the relay processor.

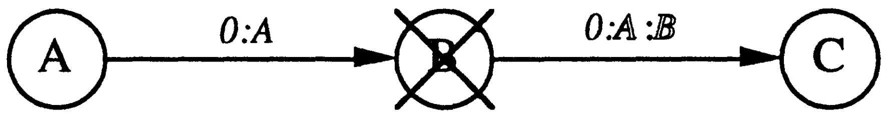
While the advantages from the extra information provided by a broadcast protocol are also present in a software-based architecture, the performance payoff does not necessarily translate. When the message processing for each link is done in separate hardware, the work can be completed in parallel. In software, the processing tasks for each message must be completed sequentially. Therefore, the number of messages sent during a round has some kind of adverse effect on performance. Whether this added latency outweighs the benefits provided by a broadcast protocol is a question that must be answered by the implementation.

5.4.2. *Signature Configuration*

Another design tradeoff between performance and extra information is deciding the number of signatures to append to messages. The *SM(m)* algorithm uses "nested signatures," where each relay processor places its own signature on the end of the message, to perform message reducing optimizations. The processors check the signatures already included in the message to decide which processors to relay the message to in the next round. In a small fully connected system such as the proposed minimal configuration, these extra signatures are not really needed for the correct operation of the protocol. So long as the sender signs the message and each processor can tell where every message comes from, the algorithm still provides consistency. The decision about whether to use nested signatures must be based on other issues.



(a) Isolating a sender fault.



(b) Isolating a relay fault.

Figure 5.4 Using nested signatures to isolate faults.

The prime advantage in using nested signatures appears when a fault occurs. If the fault is active, such as data corruption, the added signatures can be used to help isolate the location of a fault. The receiving node can attempt to pinpoint where the problem transpired by peeling off the signatures and checking them against the message until it reaches one that does not match. Figure 5.4 shows how a processor would try to discover where a fault has occurred. In Figure 5.4(a), the message is corrupted by the sender, processor *A*, and then correctly relayed by processor *B*. When processor *C* receives the message and attempts to authenticate it using the two signatures, processor *A*'s signature fails to verify the contents of the message, while processor *B*'s signature is correct. Since processor B's signature is correct, processor C then knows that the fault occurred before the message arrived at processor B and therefore must have happened at processor A. In Figure 5.4(b), the message is sent correctly by processor A and then corrupted by processor B. When processor C receives the message and attempts to authenticate it using the two signatures, neither one of the signatures verifies if the data corruption occurs before processor B's signature. Processor C then knows only that the fault occurred at or before processor B. This information can be used by a FDIR task to help determine the overall status of the system.

The main disadvantage of nested protocols is the effect on performance. For a system of N processors, each message has $(N-1)$ signatures to authenticate by the last round. When combined with the added latency of signing and sending longer messages, these effects could become prohibitive. There are steps that can be taken to reduce the time needed for processing the extra signatures using the technique shown in Figure 5.5. If the CRC calculated in verifying the received message is kept, a new CRC can be generated using the old CRC as a seed and then feeding any additional bytes such as the last signature through the generator. In Figure 5.5(a), the CRC for the first signature is generated using all zeros for the initial seed value and then feeding the majority of the message, including the message header, data, and previous trailer information, through the register. In Figure 5.5(b), the CRC for the second signature is generated using the previous CRC for its initial seed value and then feeding the first signature and any additional trailer information through the register. This same trick can be used to speed up verifying multiple signatures.

An option that could utilize the advantages of both nested and single signatures is to provide different levels of service within the operating system. Since the main purpose of the architecture is to provide a real-time Byzantine resilient system, the default would be having a single signature from the sender. This option avoids the extra penalties of signing, sending, and verifying multiple signatures and yet still provides the needed consistency. If a fault is detected or upon a periodic basis, the system could move up a level to using nested signatures. The chief purpose of these messages would be to isolate, and hopefully recover, a faulty processor (or link) before another fault occurs. The new messages would take longer to process, but would provide more information for FDIR, making the entire process of recovering from faults faster. A third level of service could be provided for applications where speed is more important than fault tolerance, by removing authentication. This level would not be able to survive malicious faults though and the system would basically become a triply redundant processor.

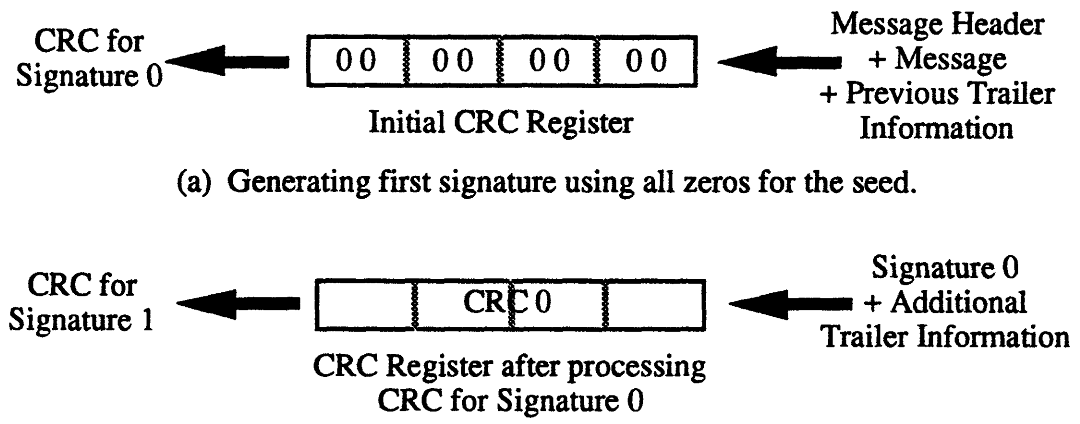

CRC for Signature 0
00
00
00
00
Initial CRC Register
Message Header + Message + Previous Trailer Information
(a) Generating first signature using all zeros for the seed.
CRC for Signature 1
CRC 0
CRC Register after processing CRC for Signature 0
Signature 0 + Additional Trailer Information
(b) Generating second signature using previous CRC for the seed.

Figure 5.5 Generating nested signatures.

5.4.3. Message Format

The format of each message must be designed to incorporate a number of different options. The *choice()* function used by the interactive consistency protocol uses CRC's as its arguments to generate its final value. All of the information necessary to fulfill the implementation requirements for the interactive consistency protocol and to perform synchronization must be included in the message without affecting this *choice()* function. Finally, the format must be ordered so that the nested signaturing scheme discussed in the previous section can be performed. All of these requirements must be combined in such a way that the added length to the message is minimized, since processing the extra information is added to the overhead.

All of the needed information must be placed in either a header which precedes the actual data or a trailer which is appended to the end of the message. Figure 5.6 shows a format using 64-bit signatures which fulfills all of the necessary conditions. The only information which is placed in the header is the message's size and type (i.e., *From_a, From_all, Voting,* etc.). This information is invariant in that it is the same on every processor. This means that CRC's of the headers and data for each copy of a message received by the processor can be compared by the *choice()* function. All of the variant information, which is processor-specific, is placed in the trailer. This includes a sequence number to fulfill the time-specific authentication requirement mentioned in Section 4.2 as well as a timestamp for use in synchronization. After the timestamp, a processor ID is placed in the trailer to identify which processor is generating the signature which follows. This part of the trailer can be expanded to include as many processor ID's and signatures as necessary.

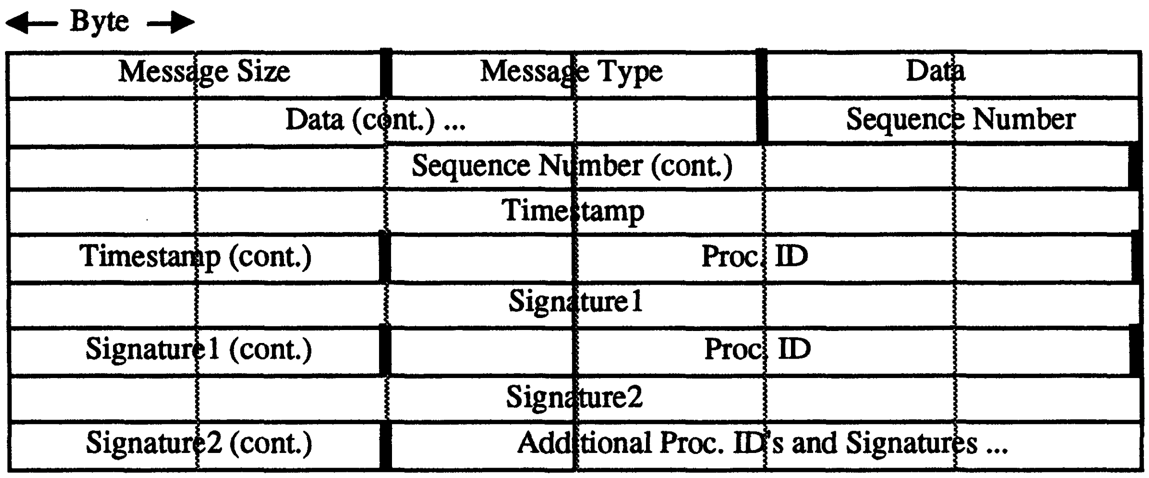
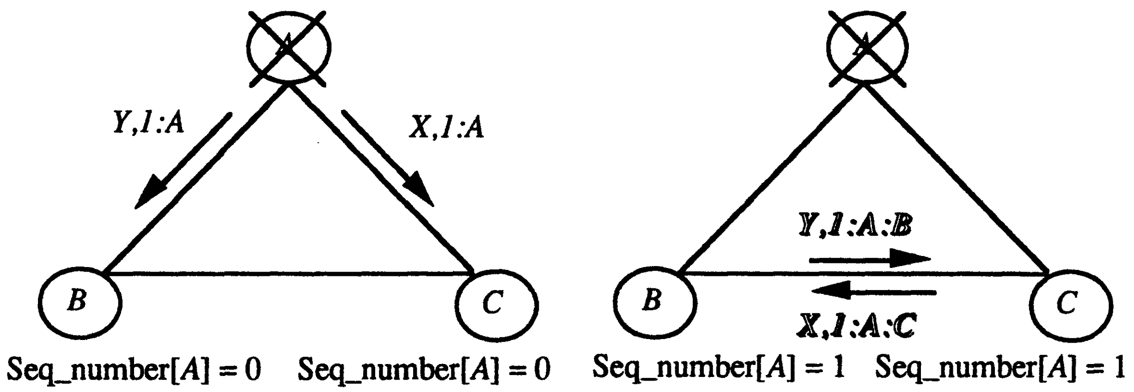


Figure 5.6 Proposed message format using 64-bit signatures.

The important characteristic of this format is that it is possible to authenticate each nested signature without having to process any byte in the message more than once. A message is authenticated in three steps. First, the CRC of the invariant information is calculated using zeros for a seed. This information is saved for use as an argument of the *choice()* function and then placed as the seed to generate the next CRC. The sequence number, timestamp, and processor ID for the first signature are then fed through the CRC generation routine to get the CRC for authenticating the first signature. The first signature and the second processor ID are then feed through the CRC routine, using the previous CRC for a seed, to get the CRC needed to authenticate the second signature. This process can be expanded to more signatures if required and minimizes the overhead necessary for authenticating nested signatures.

5.4.4. Sequence Numbers

In Section 4.2, the rationale behind placing sequence numbers in each message to prevent a relay processor from undetectably repeating an old message is explained. Generating these sequence numbers is an easy operation which can be done by any monotonically increasing function. Each processor keeps a record of the last sequence number that it used and a table with the last sequence number that it received from every other processor in the system. The structure of the interactive consistency protocol though forces the system to take some care in recording sequence numbers from other processors. The problem is that within each interactive consistency exchange, each processor can receive from one to $f+1$ copies of the same message from the sender, due to other processors relaying their own copies. If a processor updates its sequence number table as soon as it gets the first message, it would then reject every other copy as invalid. This could cause the protocol to not operate correctly with certain faults. Figure 5.7 shows the protocol reaching an incorrect result with the "two-faced" clock fault mentioned in Chapter 3. During the first round, processors *B* and *C* receive valid but different messages, both with a sequence number of 1, from processor *A* . Since they update processor *A*'s entry in their sequence number table when these messages arrive, the messages received in the second round are found to be invalid because of their old sequence numbers. Without these second round messages, processors *B* and *C* agree on different values, which goes against the Byzantine Agreement requirements. The solution is to keep a record, separate from the sequence number table, of the largest authentic sequence number to arrive during an exchange. At the end of the exchange, this number can then be used to update the sequence number table.



(a) First round. (b) Second round.

Figure 5.7 "Two-faced" clock with incorrect sequence number record keeping.

5.5. Final Protocol Design

Now that the various requirements and design issues have been examined, an interactive consistency protocol capable correctly and efficiently distributing single-source in the presence of Byzantine faults must be designed. The interactive consistency protocol that was chosen to be implemented in our architecture is based on the extension of *SM(m)* suggested by Dolev and Strong. Three separate uniprocessor exchange routines, one for each processor, are provided to distribute single-source information. A fourth function is used when each processor has a different version of the same data (i.e., each processor reads its redundant sensor). This *From_all* routine completes the steps necessary for all three of the uniprocessor *From_processor* exchanges in two round instead of the six required to carry out each exchange individually. The implementation needs to include two exchange formats:

1) exchanges with nested signatures and the point-to-point message passing used by *SM(m)*, and
2) exchanges with only the sender's signature broadcast message passing.

The performance effects of using nested signatures or broadcasts can then be examined to decide whether the advantages given by the extra information outweigh the costs of providing it.

A exchange routines were designed assuming that the synchronization layer was already in place. The details of synchronization are discussed in Chapter 7, but the basic purpose of the protocol is to provide synchronized clocks which fulfill the following conditions [Sch87]:

Virtual Synchronization: Local clocks, $\hat{c}(t)$, on each processors are synchronized to within a known skew, δ, such that:

$$\left|\hat{c}_q(t) - \hat{c}_p(t)\right| \le \delta, \text{ for } 0 \le t. \tag{5.1}$$

Virtual Clock Rate: Local clocks, $\hat{c}(t)$, drift from real time at a rate bound by a constant, $\hat{\rho}$, such that:

$$0 < 1 - \hat{\rho} \le \frac{\hat{c}_p(t+\hat{\kappa}) - \hat{c}_p(t)}{\hat{\kappa}} \le 1 + \hat{\rho}, \text{ for } 0 \le t. \tag{5.2}$$

The maximum message transmission delay constant, $\Gamma_{\max}$, is combined with the skew, δ, to determine the length of the individual rounds within the algorithms. This time limit is only an estimate. The length of the rounds in the actual implementation are also influenced by the time that is takes for the messages to be processed after they arrive.

Each processor uses a buffer space to process out-going and incoming messages. This buffer space consists of a 3x3 array of individual buffers, each containing memory to store a message, its arrival time, and associated CRC's, as well as a Verify Flag (VF). Figure 5.8(a) shows a single buffer. The Verify Flag (VF) is set when a message is correctly authenticated and is used at the end of the exchange to decide on the final value. The arrival time of a message at the processor is saved for use in synchronization. There are two classes of CRC's which are also stored: invariant CRC's, which are CRC's of only the header and message and are the same on all processors, and variant CRC's which include noncongruent information such as sequence numbers, timestamps, and processor ID's. The invariant CRC's are used by the *choice()* function to decide on a final value at the end of an exchange, while the variant CRC's are used to speed up the process of calculating nested signatures (See the previous section for a discussion of this technique). Figure 5.8(b) shows a simplified version of the entire buffer space. Each processor has space for its own out-going message during first round and incoming messages from the other two processors. A space for each uniprocessor interactive consistency exchange is needed because the multiprocessor interactive consistency exchange (*From_all*) implements all three of the exchanges at once.

Message Buffer		
Message Buffer (cont.)		
Message Buffer (cont.)		VF
Arrival Time	Invariant CRC	Variant CRC

(a) Individual buffer.

	From_A		From_B		From_C	
Processor A	Message Buffer	VF	Message Buffer	VF	Message Buffer	VF
Processor B	Message Buffer	VF	Message Buffer	VF	Message Buffer	VF
Processor C	Message Buffer	VF	Message Buffer	VF	Message Buffer	VF

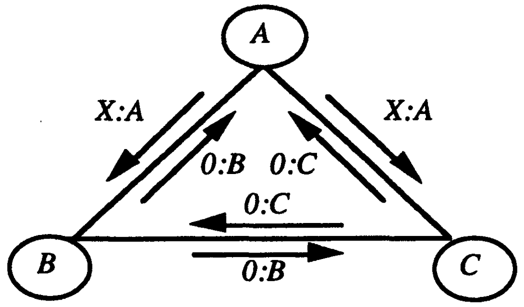
(b) Interactive consistency buffer space.

Figure 5.8 Interactive consistency exchange buffers.

The following sections contain descriptions of both the uniprocessor (*From_a*, *From_b*, and *From_c*) and multiprocessor (*From_all*) routines which implement the interactive consistency protocol. The algorithms outline the steps taken within each function with a strict separation between the two message passing rounds. In the final implementation, this separation will not be so sharp, because the routine must be able to handle messages from the second round arriving early while still rejecting late messages from the first round. Various details of how nested signatures are computed are included as well as how the *choice()* function given by Dolev and Strong is carried out by comparing the invariant CRC's of the messages. The provided algorithms are for the most complicated form of the implementation: interactive consistency exchanges with nested signatures and broadcasts.

5.5.1. Uniprocessor Interactive Consistency Exchange (From_a, From_b, and From_c)

Single-source exchanges are used when only one processor has single-source information to distribute. Figure 5.9 shows the message passing during and the buffer space after a *From_a* exchange using nested signatures and broadcasts and with no faults. In the first round, the sender (processor *A*) transmits a signed message to the other processors while the other processors send out signed NULL messages. When received, only the sender's messages are authenticated (setting the VF flag if the message verifies)



Processor *A*

	From a		From b		From c	
Processor *A*	*X:A*	√				
Processor *B*	*0:B*					
Processor *C*	*0:C*					

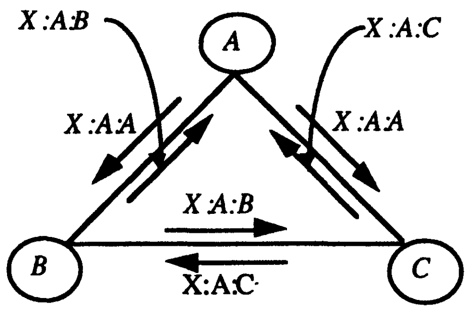
Processor *B*

	From a		From b		From c	
Processor *A*	*X:A*	√				
Processor *B*	*0:B*					
Processor *C*	*0:C*					

Processor *C*

	From a		From b		From c	
Processor *A*	X:A	√				
Processor *B*	0:B					
Processor *C*	0:C					

(a) First round.



Processor *A*

	From a		From b		From c	
Processor *A*	*X:A*	√				
Processor *B*	*X:A:B*	√				
Processor *C*	*X:A:C*	√				

Processor *B*

	From a		From b		From c	
Processor *A*	*X:A:B*	√				
Processor *B*	*0:B*					
Processor *C*	*X:A:C*	√				

Processor *C*

	From a		From b		From c	
Processor *A*	*X:A:C*	√				
Processor *B*	*X:A:B*	√				
Processor *C*	*0:C*					

(b) Second round.

Figure 5.9 *From_a* message passing and buffer space at end of rounds.

while the NULL messages are ignored. In the second round, each processor adds its own signature to the end of the sender's message and transmits it to the other processors. The processors only authenticate messages which are not from the sender, but have the sender's signature as the first signature. When the message passing is done, each processor has at least two verified copies of the message whose invariant CRC's (saved from the authentication process) can be compared to protect against sender faults.

5.5.1.1. Uniprocessor Interactive Consistency Exchange Algorithm

```
On processor $i$,

First round:
    Begin first round at local time, $\hat{c}_i\left(t_{begin}\right)$
    Send messages:
            Erase all buffers.
            If (Sender)
                Place message in correct buffer, leaving room for header at
                the beginning.
            Fill in header (Message size and message type (From_sender)).
            Place sequence number, timestamp, and processor ID in trailer.
            Compute variant CRC and sign with private key.
            Place signature in trailer.
            Send message to all other processors.

    Receive messages:
            While ( $|\hat{c}_i(t) - \hat{c}_i(t_{begin})| < (\delta + \Gamma_{max})$)
            {
                If (Receive message)
                {
                    Place in correct buffer (using the link that the
                    message arrives on and the message type).
                    if (Message from sender, has valid sequence
                    number, has only one signature, and is of the correct
                    type)
                    {
                        Authenticate signature, saving invariant and
                        variant CRC's.
                        if (Message verifies)
                        {
                            Set VF flag.
                            Get and save arrival time.
                        }
                    }
                }
            }
```

Second Round:

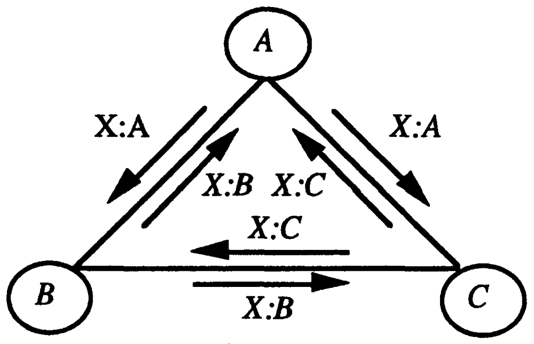
```
    Begin second round at local time, $\hat{c}_i\left(t_{begin} + \delta + \Gamma_{\max}\right)$.
    Send messages:
            If (Have message from sender)
            {
                Fill in additional trailer information for second signature
                (processor ID) on the end of message.
                Compute second signature using CRC based on variant
                CRC from first round plus processor ID and first signature
                and then signing with private key.
            }
            Send message to all other processors.

    Receive messages:
            While (( $|\hat{c}_i(t) - \hat{c}_i(t_{begin})| > (\delta + \Gamma_{\max})$) and
                ($|\hat{c}_i(t) - \hat{c}_i(t_{begin})| < 2(\delta + \Gamma_{\max})$)
            {
                If ( Receive message),
                {
                    If (Message from sender)
                        Dump message.
                    Else
                    {
                        Put in correct buffer.
                    If (Message has two signatures, has a valid
                    sequence number, its first signature is from
                    sender, and is of correct type)
                    {
                        Authenticate message, saving
                        invariant CRC.
                        If (Message Verifies)
                        {
                            Set VF flag.
                            Get and save arrival time.
                        }
                    }
                }
            }
        }
```

```
Message Processing:
    Choose correct  final value using VF flag and invariant CRC's:
            If (Only 1 verified message)
                Choose message.
            If ((2 verified messages) and (CRC1 = CRC2))
                Choose either message.
            if ((2 verified messages) and (CRC1 ≠ CRC2))
                Choose default (NULL).
```

5.5.2. Multi-Source Interactive Consistency Exchange (From_all)

The multi-source interactive consistency exchange is used when each processor has noncongruent information which needs to be distributed (i.e., readings from redundant sensors). The algorithm basically implements a *From_a*, *From_b*, and *From_c* at the same time, only taking two rounds instead of the six needed to do each individually. Three buffer spaces with room for three messages are used to keep the information separate. Figure 5.10 shows the message passing during and the buffer space after a *From_all* exchange using nested signatures and broadcasts and with no faults.



Processor *A*

	From a		*From b*		*From c*	
Processor *A*	*X:A*	√				
Processor *B*			*X:B*	√		
Processor *C*					*X:C*	√

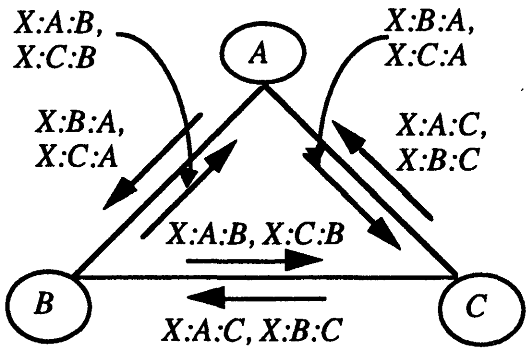
Processor *B*

	From a		*From b*		*From c*	
Processor *A*	*X:A*	√				
Processor *B*			*X:B*	√		
Processor *C*					*X:C*	√

Processor *C*

	From a		*From b*		*From c*	
Processor *A*	*X:A*	√				
Processor *B*			*X:B*	√		
Processor *C*					*X:C*	√

(a) First round.



Processor *A*

	From a		*From b*		*From c*	
Processor *A*	*X:A*	√				
Processor *i*	*X:A:B*	√	*X:B:A*	√	*X:C:B*	√
Processor *C*	*X:A:C*	√	*X:B:C*	√	*X:C:A*	√

Processor *B*

	From a		*From b*		*From c*	
Processor *A*	*X:A:B*	√	*X:B:A*	√	*X:C:A*	√
Processor *B*			*X:B*	√		
Processor *C*	*X:A:C*	√	*X:B:C*	√	*X:C:B*	√

Processor *C*

	From a		*From b*		*From c*	
Processor *A*	*X:A:C*	√	*X:B:A*	√	*X:C:A*	√
Processor *B*	*X:A:B*	√	*X:B:C*	√	*X:C:B*	√
Processor *C*					*X:C*	√

(b) Second round.

Figure 5.10 *From_all* message passing and buffer space at end of rounds.

5.5.2.1. Multiprocessor Interactive Consistency Exchange Algorithm

On processor *i*,

First Round:

Begin first round at local time, $\hat{c}_i\left(t_{begin}\right)$.

Send messages:

Erase all buffers.

```
                Place message in correct buffer leaving room for header at the
                beginning.
                Fill in header (Message size and message type (From_all).
                Place sequence number, timestamp, and processor ID in trailer.
                Compute variant CRC and sign with private key.
                Place signature in trailer.
                Send message to all other processors.

        Receive messages:
                While ( $|\hat{c}_i(t) - \hat{c}_i(t_{begin})| < (\delta + \Gamma_{\max})$)
                {
                        If (Receive message)
                        {
                                Place in temporary buffer.
                                If(Message is from the original sender, has a valid
                                sequence number, is of the correct type, and only
                                has one signature)
                                {
                                        Place in correct buffer in correct buffer
                                        space (use processor ID of first signature to
                                        choose buffer space).
                                        Authenticate signature, saving variant and
                                        invariant CRC's.
                                        If(Message Verifies)
                                        {
                                                Set VF flag.
                                                Get and save arrival time.
                                        }
                                }
                        }
                }

Second Round:
        Begin second round at local time, $\hat{c}_i\left(t_{begin} + \delta + \Gamma_{\max}\right)$.
        Send messages:
                for( Each processor, i)
        {
            If (Have message from processor i)
            {
                Fill in additional trailer information for 2nd
                signature (processor ID) on the end of message.
                Compute second signature using CRC based on
                variant CRC from first round plus processor ID and
                first signature and then signing with private key.
            }
            Send message to all other processors.
        }

Receive messages:
        While (( $|\hat{c}_i(t) - \hat{c}_i(t_{begin})| > (\delta + \Gamma_{\max})$) and
            ($|\hat{c}_i(t) - \hat{c}_i(t_{begin})| < 2(\delta + \Gamma_{\max})$))
        {
            If ( Receive message)
            {
                Place message in temporary buffer.
                If (Message is not from the original sender, has a
                valid sequence number, and is of the correct type)
                {
                    Place in correct buffer, using processor ID's
                    from the two signatures.
                    Authenticate message, saving invariant
                    CRC.
                    If (Message Verifies)
                    {
                        Set VF flag.
                        Get and save arrival time.
                    }
                }
                Else
                    Dump message
            }
        }
```

```
Message Processing:
    Choose correct final value for each sender:
            for (Each processor)
            {
                If (Only 1 verified message from sender)
                    Choose message.
                If ((2 verified messages from sender) and (CRC1 = CRC2))
                    Choose either message.
                if ((2 verified messages from sender) and (CRC1 ≠ CRC2))
                    Choose default (NULL).
            }
```

5.5.3. Implementation Results

Certain questions about the interactive consistency protocol cannot be answered until part of the architecture is implemented in code. The protocol design suggested that two exchange formats be implemented in order to investigate the tradeoffs of using nested signatures instead of only the sender's signature or using broadcast versus point to point message passing. Both nested signatures and message broadcasts provide more information to the system, especially for FDIR. The problem is that both techniques adversely affect the performance of the system. An implementation can be used to examine the extent of this added overhead. This implementation was completed using the C programming language and then run on a Sun workstation. Overhead measurements were taken using timer system calls at points in the code. The use of these calls added to the timing results, but an assumption is made that the extra overhead is constant over the various runs.

The final implementation differs from the exchange algorithms of the previous section in one important way. The algorithms have a strict separation between the responsibilities of the two rounds in that the relay messages are not sent nor accepted until the second round begins. In the actual implementation, messages are relayed and processed as soon as they arrive. A check is made on incoming messages to remove late arriving messages from the previous round, but early arriving messages do not affect the outcome of the message exchange. The reasoning behind this filtering is explained in depth in Chapter 7, but in brief, the only way a faulty processor can undetectably affect the timing of the message exchanges is to hold onto messages, making them late.

The performance overhead introduced by nested signatures is present throughout all of the interactive consistency exchanges, while the issue of using broadcast message passing is only relevant for the single-source exchanges. For this reason, the performance measurements are taken using the single-source *From_a* routine. Also, there is one area in the implemented protocol where the effects of the extra message processing are readily apparent. This point occurs during the first round of message passing while the processors are waiting for messages to arrive. When a message is received, the processor authenticates the signature(s), appends a second signature if nested signatures are being used, and then relays the message to one or two processors, depending on whether broadcasts are being employed. The routine must be able to finish processing the first round messages before the round ends. The performance overhead measurements therefore concentrate on this part of the exchange.

With the optimizations explained in Section 5.4.2 for generating nested signatures, using a nested protocol should add very little to the overhead of processing messages. When messages arrive, they are authenticated by generating a variant CRC of the header, message, and the first part of the trailer (which contains the sequence number, timestamp, and the originating processor's ID), multiplying the signature by a public key, and comparing the two results. In the buffer shown in Figure 5.8(a), a space has been provided to save this variant CRC. Generating a second signature involves using this variant CRC for a seed and feeding the first signature and the relay processor's ID through the encoding function, before signing it with the relay processor's private key. With the message format shown in Figure 5.6, this means that only 12 additional bytes need to be processed. Figure 5.11 shows the performance overhead involved in processing a message which arrives from the sender during the first round. An interesting result is that the time needed to receive the message, add the second signature, and then send the new message to the relay processor remains constant for varying length messages. As expected, little of the total overhead is spent adding the second signature. When compared to the time needed to send messages which takes close to 500 microseconds, the 60 or so microseconds of nested signature overhead is insignificant. The total overhead is greater than the sum of these three measurements due to the time needed to authenticate the first signature, plus scheduling on the Sun workstation.

The use of broadcast message passing has two major effects on the part of the protocol being examined. First of all, each relay processor must be able to handle two messages during the first round, a message containing data from the sender and a NULL messages from the other relay processor. The NULL messages are not authenticated, but they cannot be dumped because the receiving processor must check to be sure that the

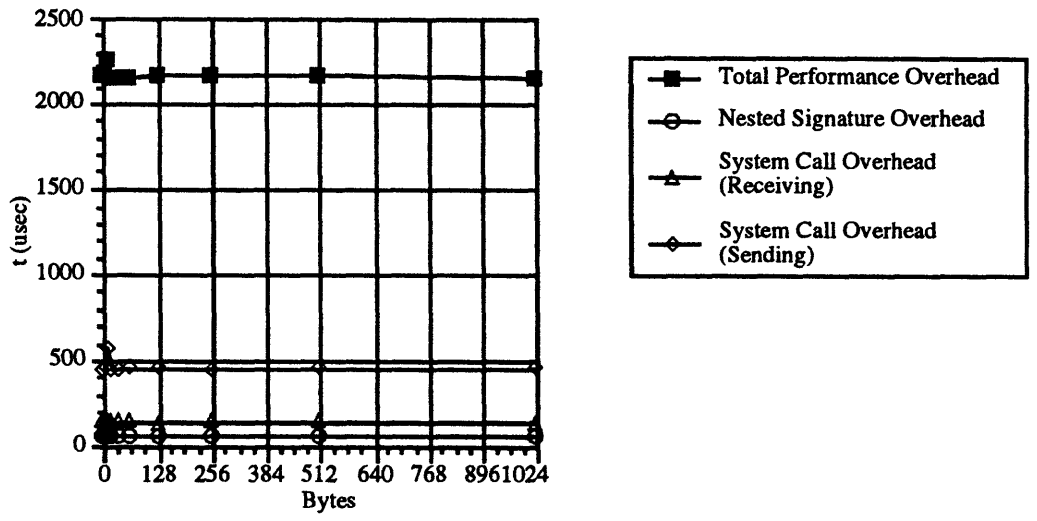


Figure 5.11 Nested signature performance overhead.

message is not an early second round message. Secondly, once the sender's message has been authenticated, the relay processor must send it to both processors instead of just to the other relay processor. Figure 5.12 shows the performance overhead involved in processing a message from the sender when broadcasts are being used. The effects of having to send this second message are immediately obvious. The system call overhead attributed to sending messages has increased from the 500 microseconds found with the point to point message passing used in Figure 5.11 to around 700 microseconds. Once again, the total overhead is greater than the sum of these three measurements due to the time needed to authenticate the first signature, plus scheduling on the Sun workstation.

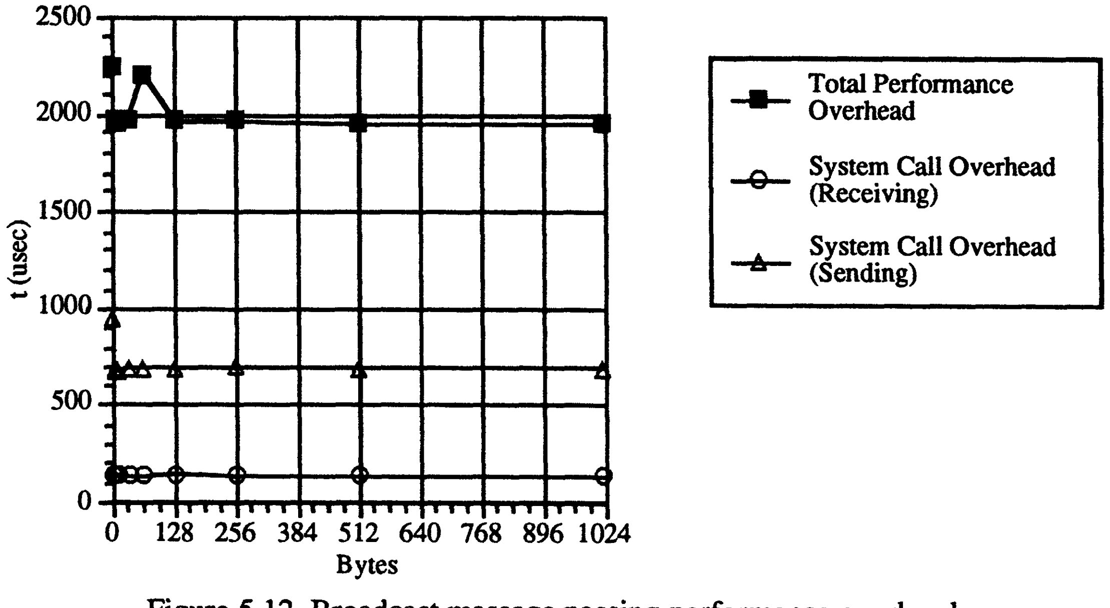


Figure 5.12 Broadcast message passing performance overhead.

A number of conclusions can be drawn from the above performance measurements. First of all, any optimizations which can be done on the I/O system calls need to be incorporated into any final implementation. These system calls provide the interface between the Message Passing Layer and the system's hardware and are used constantly. The drag on the system when two messages are sent by the broadcast protocol is significant. Secondly, any optimizations which reduce the overhead of processing messages during the exchange need to be used. The point to point message passing suggested by *SM(m)* should be used when implementing the single-source interactive consistency exchanges. Also, even though the use of nested signatures has a minor effect on performance overhead, they still only need to be used by FDIR tasks.

6. Voting

6.1. Overview

The second protocol found in the Message Passing Layer is the voting protocol. This protocol is responsible for generating a "group consensus value," a value which is a function of the different versions of the piece of data on different processors. Since the processors are supposed to be operating congruently, in the absence of faults, the group consensus value is the same as the value found on a single nonfaulty processor. Fault, detection, isolation, and recovery (FDIR) tasks use the voting protocol on a periodic basis to generate fault symptoms and detect faults. A voting exchange is also used before values are output externally to filter out any computational errors.

The first step in designing this protocol is to discuss the requirements of the voting exchange and how they interface with the requirements from the interactive consistency protocol. Once the final requirements for the system are known, the voting algorithm can be examined and any design issues resolved. The final step is to combine the requirements and design options into an efficient final protocol design.

6.2. Voting Requirements

The requirements for the voting protocol are a combination of guidelines originating from the voting protocol itself and constraints placed on the architecture by the interactive consistency protocol. The system's claim of being Byzantine resilient rests on fulfilling the requirements discussed in Section 5.2, so the voting requirements must not be allowed to conflict with them. Also, the goal of low-costs makes implementing the voting protocol with the minimal interactive consistency configuration shown in Figure 6.1 a top priority. First of all, the requirements specific to the voting protocol must be explored. Then, each of the interactive consistency requirements must be examined in terms of these voting requirements to show that the two lists are not in violation of each other.

In contrast to the interactive consistency protocol, where the main emphasis is on message passing, the most important element of the voting protocol is its *choice()* function. This function is responsible for calculating a group consensus value from a set of data gathered from all of the processors. The only requirement is that the function calculate the same value on all nonfaulty processors. This is an important point since the voting protocol is not required to be Byzantine resilient in that the sets of data on

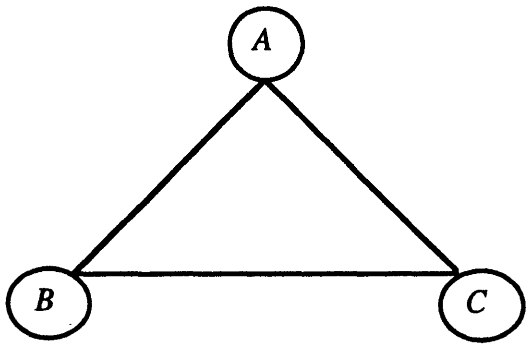
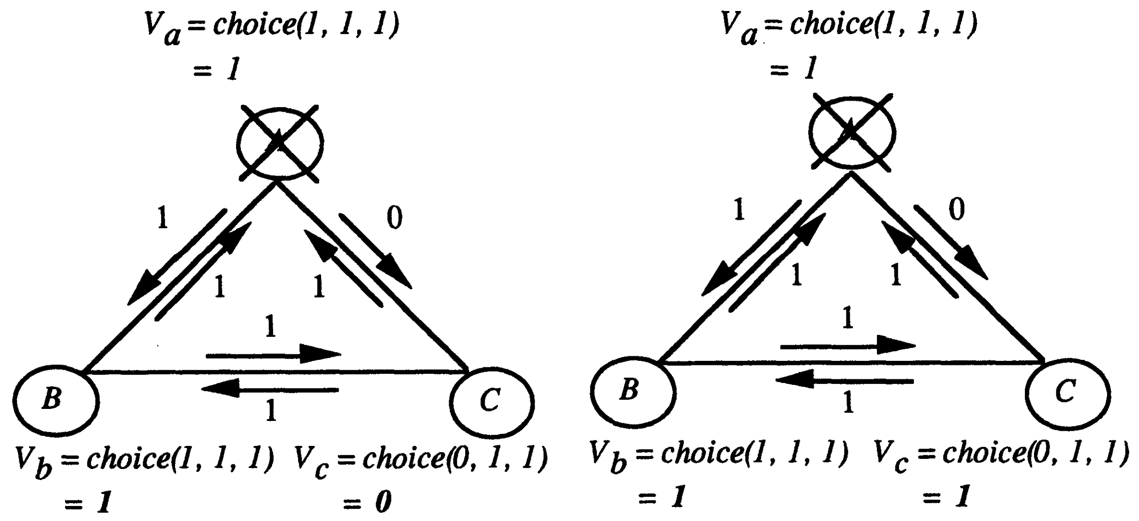


Figure 6.1 Minimal configuration fulfilling interactive consistency requirements.

nonfaulty processors do not need to be consistent. Figure 6.2 illustrates two different choice functions calculating a group consensus value from data collected with processor *A* acting as a two-faced clock and sending different values to processors *B* and *C*. Even though the sets of data vary on the different processors, a "correct" *choice*() function generates the same group consensus value on all nonfaulty processors. In Figure 6.2(a), the choice function simply uses the value from processor *A* as its group consensus value. This function does not operate correctly since with processor *A* being faulty, the nonfaulty processors *B* and *C* have generated different group consensus values. In Figure 6.2(b), the choice function calculates a value which is equal to a majority of elements within the set. This function does operate correctly since even with processor *A* being faulty, the nonfaulty processors *B* and *C* both generate the same group consensus value.



(a) Incorrect choice function.

(b) Correct choice function.

Figure 6.2 Examples of incorrect and correct choice functions.

The hardware requirements for voting depend heavily on the structure of the protocol. There is no lower bound on the number of processors needed in the system. Basically, the *choice*() function is tailored to handle the number of processors present and the number of faults allowed. The easiest solution is to make certain that there is always a majority of nonfaulty processors. There are *choice*() functions which operate correctly without a strict majority, but their design and implementation is complicated. The connectivity and number of rounds requirements are determined by the type of data sets which are desired. If the protocol wants to have consistent sets of data on each processor, the constraints are the same as for the interactive consistency protocol (a connectivity of $f+1$ with $f+1$ rounds). If there is no need for consistent data sets, the voting protocol can be completed in one round, so long as there is a direct link between every processor. In other words, the system must be fully connected. The final requirement of having synchronized clocks is necessary to terminate the voting exchange when a processor becomes faulty and fails to send a message.

The requirements placed on the proposed architecture by the interactive consistency protocol are compared to those imposed by the voting protocol in Table 6.1. The two protocols are not in conflict over the minimum number of processors since the number of processors can be chosen to fulfill the interactive consistency requirement and the voting protocol then designed around that specified number. The number of rounds used by each protocol have no effect on each other, so only the connectivity requirement can come into conflict. The pros and cons of demanding consistent data sets are discussed later in this chapter, but for now, the design is examined on the basis of the more stringent requirement of being fully connected.

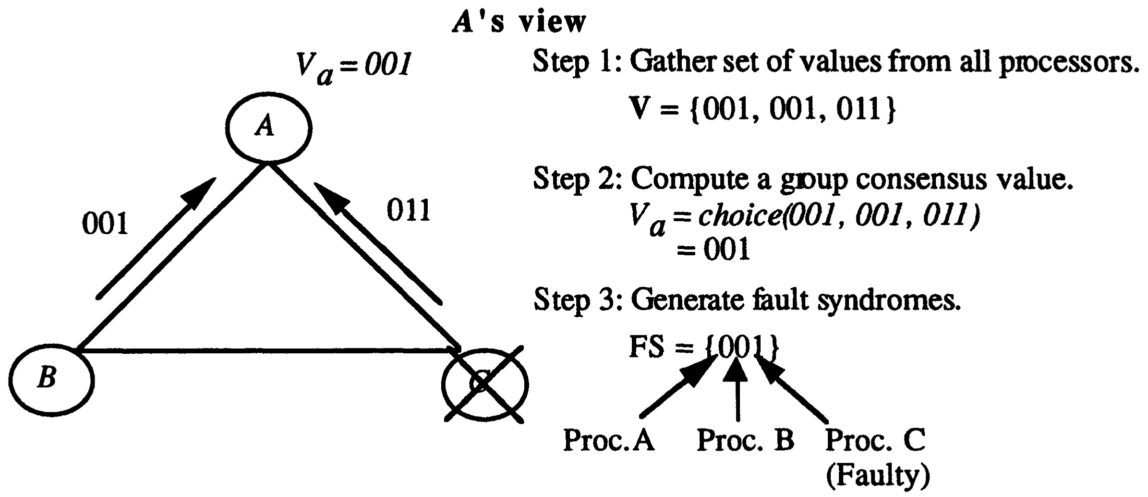
Requirements	Interactive Consistency	Voting
Number of processors	$2f+1$	None
Connectivity	$f+1$	Consistency Req.: $f+1$ No Consistency Req.: Fully connected
Communication Rounds	$f+1$	Consistency Req.: $f+1$ No Consistency Req.: 1
Synchronization	Yes	Yes

Table 6.1 Interactive consistency vs. voting requirements.

In the previous chapter, it was decided that the minimum configuration shown in Figure 6.1 fits all of the hardware requirements for making a system 1-Byzantine resilient while fulfilling our desire for a low-cost implementation. The three processors meets the majority processor bound of $N > 2f + 1$ and still allows flexibility in deciding on a *choice()* function. The configuration also has a connectivity of two and is fully connected, leaving the decision about which type of data sets are desired open.

6.3. Voting Algorithms and Design Issues

The voting protocol is used when the processors are expected to be acting congruently and therefore performing the same operations on identical data. Exchanges are performed to provide a group consensus value which is the same on all nonfaulty processors. The algorithm directing the operation of the protocol can be split into two sections: 1) the message passing necessary to collect data from all processors and 2) the *choice()* function which uses the gathered data to provide the group consensus value. Figure 6.3 gives a general outline of the steps necessary to calculate fault symptoms from the viewpoint of processor *A*. Processors *B* and *C* send a copy of their data to processor *A*, providing processor *A* with a set of three copies of the data. A *choice()* function then calculates the group consensus value. Finally, the group consensus value is used by the FDIR task on processor *A* to generate a fault symptom which points to a fault on processor *C*.



Step 1: Gather set of values from all processors.
V = {001, 001, 011}

Step 2: Compute a group consensus value.
$V_a = choice(001, 001, 011)$
$= 001$

Step 3: Generate fault syndromes.
FS = {001}

Figure 6.3 Example of a voting protocol

The actual mechanics of collecting the data set and calculating the group consensus value must now be designed. Whether or not the data sets on each of the processors are consistent determines what operations the group consensus value can be used for. The presence of authenticating capability adds a new dimension to the format of individual messages which also needs to be explored. Finally, the different types of *choice()* functions must be examined to find the best fit for this architecture.

6.3.1. Consistency of Data Sets

At the end of message passing during a voting exchange, each processor has a set of data consisting of copies from every processor in the system. The protocol must then arrive at the same group consensus value on all nonfaulty processors even if the data sets on the processors are not consistent. Having consistent data sets though makes the results of the exchange much more powerful, but the added constraint also adversely affects performance. As in many of the interactive consistency design options, this issue is a tradeoff between more information and speed.

The relationship between the data sets found on the nonfaulty processors determines the scope of the voting exchange. The operations which can be performed using the data sets are severely limited if the data sets are not guaranteed to be consistent. Inconsistent information can cause processors to act noncongruently. This is not a fault if the information is only used locally on a single processor with no decisions made from the results (i. e., updating a log to show if a fault has been detected). If any system-level actions involving all of the nonfaulty processors are desired, the data sets must be consistent to make sure that operations using the information are congruent and the processors do not diverge.

The main problem with requiring consistent data sets is in the length of time taken to complete the exchange of information. In order to distribute the information correctly, the message passing in the voting protocol becomes an instantiation of the multi-source interactive consistency *From_all* routine described in Chapter 5. This exchange takes $f + 1$, or in our proposed system, 2, rounds to complete. On the other hand, the exchange can be completed in only one round if the consistency requirement is lifted and the system is fully connected. This means that the exchange is completed at least twice as fast as the interactive consistency version.

This performance speed-up from using an one-round exchange definitely outweighs any advantages gained through making the data sets consistent. The voting protocol is meant to be used as a quick check for detecting faults and filtering out computational errors. Implementing the exchange as an interactive consistency routine negates any advantage of having a separate protocol. The best solution is to implement an one-round exchange and then take care to perform interactive consistency exchanges when the data sets are needed for congruent operations.

6.3.2. Authentication of Messages

The authentication layer provides the option of signing the voting messages with digital signatures. Whether or not the signatures are actually required depends on the decision discussed above about providing consistent data sets. If the protocol implements an interactive consistency algorithm to distribute the information, authentication is needed in order to provide consistent data within the hardware constraints of the system ($2f+1$ processors with a connectivity of $f+1$). If the one round exchange is used, the issue becomes more complicated. Authentication is not needed because there is no guarantee that the data on each processor is consistent. Since the goals of an one-round exchange can be achieved with or without authentication, the decision must be performance-based.

Signing messages allows a significant optimization of the *choice()* function. Authentication allows faster implementation of the function for long messages since the signatures can be used as the function's arguments instead of messages, with the added probability of making an incorrect decision due to a forged signature being only 2^{-n} for a n-bit signature. This decrease in the size of the elements in the data sets allows the group consensus value to be calculated using n-bit arithmetic instead of having to process the entire message. The problem is that a definite delay is added for the signing and verifying of messages which could dominate performance for short messages, making unauthenticated message passing a better choice. An option is to decide whether to sign a message or not depending on message length, but the delay of actually making the decision could once again outweigh any speed advantage that is gained.

6.3.3. Choice() Functions

Choice() functions in voting protocols play the important role of calculating a group consensus value which is guaranteed to be consistent on all nonfaulty processors. A secondary role of the exchange is to generate fault symptoms for FDIR. These functions must operate correctly even when the data sets provided by message passing are not consistent themselves. The many different types of *choice()* functions need to be discussed and their advantages and disadvantages examined. Then, architectural and performance issues must be taken into account before selecting the best *choice()* function for the proposed system.

Choice() functions fall into two main categories: those which operate on the entire set of data at the same time and those which perform a series of pairwise comparisons. The most common method in the first category is to perform a strict majority vote and choose the value which is contained in over half the set. Abstractly, this is done on a bit by bit basis, but actual implementations are usually on a byte or word level. Another possible "voting" scheme is to use a plurality *choice()* function where fifty percent of the set do not need to be equal to the final value, just a sufficiently "large" number of the set (plurality). This mechanism is most useful though for systems involving a large number of divergent hosts and does not seem relevant for the small number of processors involved in this project (probably, only three processors). If either one of these functions are used, a second step is needed in the protocol to generate the fault syndromes. Each value in the set must be compared to the group consensus value. If there is a difference, the processor from which the value came from is assumed to be faulty. The second type of *choice()* function carries out individual pairwise comparisons on the set. Each value from another processor is compared to the host's value. If more than one of these values are different from the host's value, the non-host values are then compared to decide whether the host or the other processors are faulty. If the system assumes that only one fault can occur at a time, this second step is not necessary and the host is assumed to be faulty only if its value differs from more than one other value. If only one other value is different from the host's copy, the processor that sent the incorrect version is assumed to be faulty. In a large system trying to protect against more than one fault, this method is not cost-effective. In a system with N processors, the worst case is when $(N-1)$ values differ from the host. This algorithm could use up to $(N-1)!$ comparisons to pinpoint where the faulty processor is. This method has an advantage though over full-set voting in that the fault syndromes can be found as the group consensus value is computed. In systems which are only protecting against one fault, only $(N-1)$ comparisons are need which could be faster than a function which uses a full-set vote and then N comparisons to get the fault syndromes.

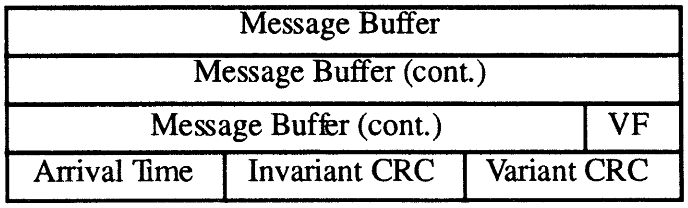
The fact that there are very few theoretical requirements for voting algorithms allows the design of a protocol to allow for characteristics of the overall architecture as well as performance issues. If the system is implemented totally in software as planned, the comparison *choice()* function has definite performance advantages since it only takes one line of code to generate the group consensus value (calculating the fault syndromes would require more) compared to the larger amount needed for full-set voting. This performance advantage can be made negligible to non-existent with the addition of extra hardware to perform votes and comparisons in parallel. The problem is that the specialized hardware would increase the cost of the proposed system.

6.4. Final Protocol Design

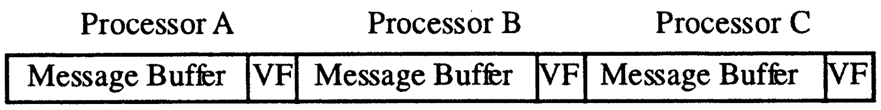
Now that the various requirements and design issues have been examined, a voting protocol which can generate a group consensus value quickly and efficiently must be designed. The message passing section of the protocol is carried out in one round where each processor sends its copy of the data out on all of its links and then waits for a message from every other processor in the system. Synchronized clocks which fulfill the virtual synchronization and clock rate conditions described in Section 5.5 are once again assumed to be present with the maximum message transmission delay, Γ_{max}, and skew, δ, determining the estimated length of the round. The implementation needs to include two different exchange formats:

1) an old-fashioned exchange with unsigned messages and a full-set majority vote of the messages as the *choice()* function and
2) an exchange which signs its messages and uses pairwise comparisons of authenticated signatures as its *choice()* function.

The performance tradeoffs between the two options can then be examined to decide which is the best for the proposed architecture.



(a) Individual buffer.



(b) Voting buffer space.

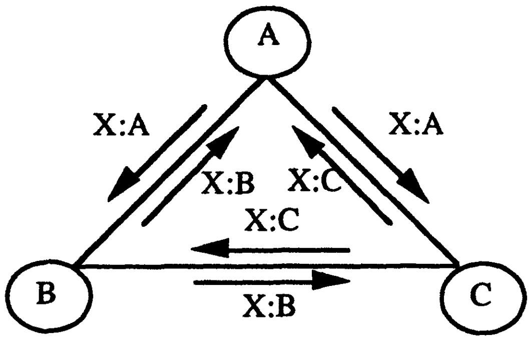
Figure 6.4 Voting exchange buffers.

Each processor uses a buffer space to process out-going and incoming messages. This buffer space consists of a 1x3 array of the same buffer used by the interactive consistency protocol and shown in Figure 6.4(a). The Verify Flag (VF) and Invariant CRC space are used in the authenticated voting version by its *choice()* function. The slots for the arrival time and variant CRC's are holdovers from the interactive consistency protocol and are not used by this exchange. Figure 6.4(b) shows a simplified version of the entire buffer space. The buffer space is much smaller than the one used by the interactive consistency protocol, since only one message from each processor needs to be stored.

The following section contains a description of the voting exchange routine. The algorithms outline the steps taken within each function. Various details of how the messages are authenticated as well as how the *choice()* functions for both exchange formats are carried out are included.

6.4.1. Voting One-Round Exchange

This routine is carried out in two distinct phases. First of all, data sets are gathered containing a copy of data from every processor. Then, a *choice()* function is



	Processor *A*		Processor *B*		Processor *B*	
Processor *A*'s buffer space	*X:A*	√	*X:B*	√	*X:C*	√

	Processor *A*		Processor *B*		Processor *B*	
Processor *B*'s buffer space	*X:A*	√	*X:B*	√	*X:C*	√

	Processor *A*		Processor *B*		Processor *B*	
Processor *C*'s buffer space	*X:A*	√	*X:B*	√	*X:C*	√

Figure 6.5 Voting message passing and buffer space at end of round.

used to calculate the group consensus value. Figure 6.5 show the message passing during and the buffer space after an one-round exchange using authentication with no faults. At the end of the message passing round, each processor has a message from the other two processors which has been authenticated.

6.4.1.1. Voting Exchange Algorithm

```
On processor $i$,

    Begin round at local time, $\hat{c}_i\left(t_{begin}\right)$.

    Send messages:
        Erase all buffers.
        Place message in correct buffer leaving room for header at the beginning.
        If(Authenticated version)
        {
            Place sequence number and processor ID in trailer.
            Compute and save invariant CRC.
            Compute variant CRC and sign with private key.
            Place signature in trailer.
        }
        Send message to all other processors.

    Receive messages:
        While $(|\hat{c}_i(t) - \hat{c}_i(t_{begin})| < (\delta + \Gamma_{\max}))$
        {
            If(Receive message)
            {
                Place in correct buffer.
            }
        }

    Vote messages:
        If(Authenticated voting)
        {
            For( j=0; j < (Number of processors); j++)
        {
            If ($i \neq j$)
            {
                Compute and save invariant CRC.
                Compute variant CRC and authenticate message
                with public key.
                If(Message authenticates)
                {
                    Set verify flag (VF).
                }
            }
        }
        Compare messages using verify flags (VF) and invariant CRC's to
        calculate the group consensus value.
    }

    If(Non-authenticated voting)
    {
        for( j = 0; j < message size; j++)
        {
            Compare bytes of the three messages to calculate group
            consensus value.
        }
    }
```

6.4.2. Implementation Results

The design of the voting protocol cannot be completed until sections of the protocol's functionality are implemented. This implementation is needed to answer a number of questions whose decisions are totally dependent on performance. Two different exchange formats have been proposed. The first option exchanges unsigned messages and then does a full-set majority vote on the messages to generate a group consensus value while the second choice signs its messages and uses pairwise comparisons of authenticated signatures. Both of these designs fulfill the requirements for the protocol, so a decision must be made based on the amount of performance overhead needed to process messages with the two techniques. An implementation was completed using the C programming language and then run on a Sun workstation. Overhead measurements were taken using timer system calls at points in the code. The use of these calls added to the timing results, but an assumption is made that the extra overhead is constant over the various runs.

Both of the design options process messages in two steps. First of all, the message is prepared and then sent to every other processor in the system. Then, the set of messages collected through the message passing is entered into a *choice()* function. The message preparation for the first design option is simple since the only step required is the addition of a header with the message's size and type . Once the set of messages is gathered, the *choice()* function simply needs to vote blocks of the messages until the ends of the messages are reached. On a machine using long word boundaries, the fastest way to implement this routine is to examine the messages in blocks of 32 bits. Implementing the second design option is more complicated. In addition to adding the header to the beginning of the message, a digital signature has to be generated and appended to the end of the message. Once the messages are gathered, the *choice()* function authenticates the messages from the other processors and compares the invariant CRC's of the verified signatures to the invariant CRC of its own message.

Determining the performance overhead introduced by each design choice involves examining the amount of time spent processing messages in functions not duplicated by the other option. First of all, the additional time needed by the option using authentication to sign its message needs to be measured. Then, the time spent in the system calls used to send and then receive messages has to be measured since each options is exchanging different length messages (the option using authentication has longer messages due to the digital signatures). Finally, the time spent actually processing the messages within the *choice()* function needs to be measured.

Two important results can be seen from the performance overhead comparison of the two design options shown in Figure 6.6. First of all, two different overheads are plotted for each option: one line accounting for all of the performance overheads and another line showing the performance overhead without the time taken up by the I/O system calls. Over 800 microseconds are needed to send the two messages to the other processors, approximately 400 microseconds/message. The receiving system call used closer to 100 microseconds for each message, which is less than for sending messages, but still a significant amount. Secondly, there is an obvious difference in the times needed by the two options. The extra computations needed to generate and then sign CRC's means that the time taken to just append the signature to the message is longer than the time required to vote the messages after they are received. When the overhead of authenticating the signatures of messages from other processors is included, this difference becomes even more pronounced. Therefore, the voting protocol needs to be implemented without digital signatures and using a full-set majority vote *choice()* function. Even with all of the optimization made to reduce the overhead introduced by authentication, the process is still slower than the voting of a message.

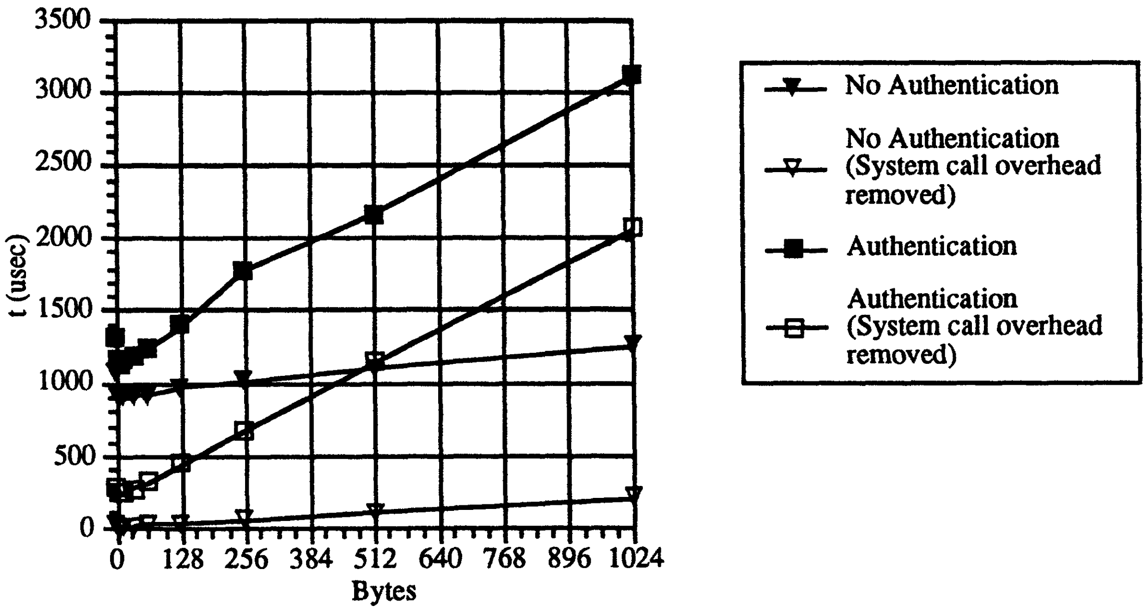


Figure 6.6 Comparison of voting performance overheads.

7. Synchronization

7.1. Overview

The Synchronization Layer is responsible for providing synchronized virtual clocks to the Message Passing and Application Layers. The synchronization protocol within this layer bounds the drift between the virtual clocks on different processors so that their ideas of real time are always within a known skew of each other. These virtual clocks are implemented on top of the hardware clocks in a way which makes synchronization invisible to the user. The synchronization protocol for this system is the last protocol to be designed since certain facets of the design are constrained by decisions made in implementing the rest of the architecture. The amount of synchronization possible is dependent on the specific protocol and parameters of both the protocol and system.

This chapter first gives a general outline of synchronization protocols and then discusses the various design issues involved in implementing a protocol with this architecture. The final choice of protocol is then described in detail with a discussion on how to bound important protocol parameters.

7.2. Synchronization Protocols

Many clock synchronization protocols have been suggested over the years for dealing with different problems or optimized for specific cases. At first, evaluating the different routines was very difficult. Subtle variations in the model and assumptions used to define each protocol complicated any attempt at making comparisons. Adding to the confusion, clock synchronization protocols were split into three different classes: interactive convergence protocols, interactive consistency protocols, and diffusion/flood protocols. This changed in 1987, when Fred Schneider presented a general paradigm from which all of the above categories of clock synchronization protocols could be derived [Sch87].

7.2.1. Schneider's Clock Synchronization Paradigm

Schneider's generalized paradigm placed the study and development of clock synchronization protocols on a totally new footing. The protocols can now be compared using his model of the problem and the advantages and disadvantages of each protocol isolated. The paradigm also pinpoints three subproblems which define each and every clock synchronization protocol. Protocols can be custom-designed based on the needs and characteristics of each system. Most importantly, so long as the new protocol adheres to the model and assumptions underlying the paradigm, proving that it is correct is greatly simplified.

Schneider defines a system as a collection of processors with virtual clocks. Each virtual clock is implemented using a hardware clock and a reliable time source (RTS) which calculates an adjustment value for the hardware clock using a convergence function. A formal definition of each of these components is found in Appendix B where the correctness of the protocol is proved. In general, hardware clocks are viewed as counters which start from some fixed initial value and monotonically increase by increments of one with a rate which must be within a fixed constant, known as the hardware drift rate of the clock, of the rate at which real time passes. Virtual clocks are implemented on top of these hardware clocks so that they also are monotonically increasing with a fixed virtual drift rate. The RTS is an abstraction which is responsible for generating an event at a specific real time which in turn causes the processors to resynchronize their clocks. The RTS then provides an adjustment value to each processor for use in implementing its virtual clock. If the protocol works correctly, the virtual clocks on all nonfaulty processors are always within a known skew of each other.

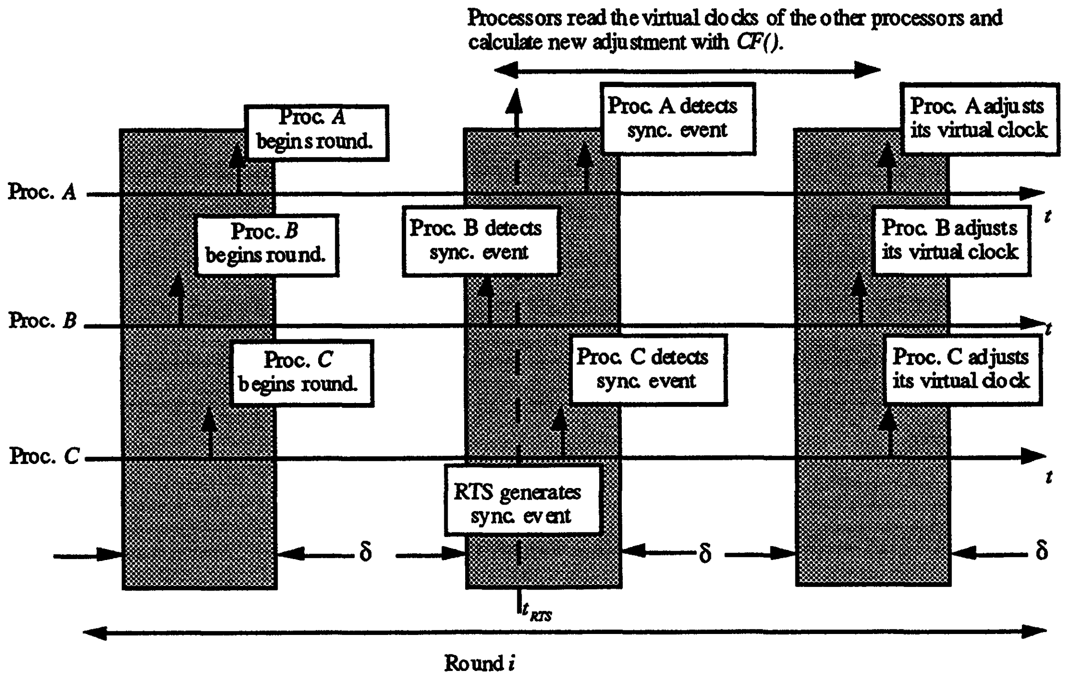

Processors read the virtual clocks of the other processors and calculate new adjustment with CF().
Proc. A begins round.
Proc. B begins round.
Proc. C begins round.
Proc. A detects sync. event
Proc. B detects sync. event
Proc. C detects sync. event
RTS generates sync. event
Proc. A adjusts its virtual clock
Proc. B adjusts its virtual clock
Proc. C adjusts its virtual clock
Proc. A
Proc. B
Proc. C
t
δ
t_RTS
Round i

Figure 7.1 An abstract picture of clock synchronization.

Schneider provides a clock synchronization paradigm which outlines the steps involved in each synchronization interval. Figure 7.1 gives an abstract picture of these events. The RTS generates a synchronization event at a real time, t_{RTS}. This event is only an abstraction of when the synchronization interval would begin on the processors if their virtual clocks were completely synchronized to real time. Instead, each processor then "detects" the synchronization event when their virtual clock reaches its idea of t_{RTS}. All of the nonfaulty processors must detect the synchronization event within a known skew of each other, but it is entirely possible for a fast processor, like processor *B* in Figure 7.1, to detect the synchronization event before the RTS has even generated it. Each processor then reads the virtual clocks of the other processors and calculates a new adjustment for its virtual clock using a convergence function (*CF*). The synchronization interval ends on each processor when it adds the new adjustment to its hardware clock. Restating the paradigm more precisely, let *i* be the number of the current round, adj_p be the adjustment added to the hardware clock for each processor, *p*, C_p be the virtual clock of each processor, *p*, c_p be the hardware clock of processor, *p*, and t_p be the real time at which each processor *p* detects the event [Sch87]:

```
$i := 1;$
$adj_p^0 := 0; adj_p^1 := 0;$
**do forever**
    detect event generated at time $t_{RTS}^{i+1}$;
    $t_p^{i+1}$ = real time now;
    $adj_p^{i+1} := CF\left(p, C_1^i\left(t_p^{i+1}\right), \ldots, C_N^i\left(t_p^{i+1}\right)\right) - c_p^{i+1}\left(t_p^{i+1}\right);$
**od**
```

The above paradigm leaves three important areas unspecified. Different implementations of the "detect event generated at time t_{RTS}^{i+1}" determine when and how often the system resynchronizes. The method that a processor uses to read the virtual clocks on other processors must be chosen to mesh with other characteristics and requirements of the system. Finally, the convergence function used in many way decides the accuracy and precision of the overall protocol. The solutions for these three subproblems totally define a clock synchronization protocol.

7.3. Synchronization Design Issues

Decisions made in the rest of the design constrain and shape the possible synchronization protocols for this architecture. The use of authentication to provide interactive consistency with only $2f+1$ processors and $f+1$ communication links, where f is the number of Byzantine faults which the system can tolerate, places restrictions on the design of the clock synchronization protocol. The trade-off which allows fewer processors and links in the system requires that all single-source information passing between the processors (i. e., a processor's virtual clock value) be signed and then verified using unforgeable signatures. This immediately rules out protocols using analog information, such as the phased-locked clocks proposed in [KSB85] where knowledge of all other clock pulses is necessary. Only protocols which use digital message passing techniques can be integrated into an authentication-based system. The algorithm which is used for interactive consistency plays a major role since the protocol needs to exchange individual clocks. In effect, the synchronization protocol needs to be built on top of this algorithm. These factors must be kept in mind when examining the different implementation options for Schneider's subproblems.

7.3.1. Detecting an event generated at time, t_{RTS}^{i+1}

There are two basic ways of generating and detecting a synchronization event based on increments of real time [Sch87]. The first option depends on the assumption that the virtual clocks are already synchronized to within a known skew. For some predefined value, R, each processor waits until its virtual clock reads a multiple of R before starting the next synchronization interval. Since the clocks are synchronized, the slowest nonfaulty processor must decide to resynchronize within a known skew of the fastest nonfaulty processor's action. This method is used for implementations which already have access to the information needed to resynchronize their clocks. A good example of this is the technique used by the phase-locked clocks found in [KSB85] where each clock can monitor the clock pulses of the other clocks in the system and adjust themselves accordingly. A second way to generate the synchronization event is to have each processor broadcast a message when its virtual clock reads a predefined value and to resynchronize when such a message is received from a correct processor. This method is used in all of the protocols which employ Byzantine Agreement (BA) message passing to synchronize [HSSD83, LL83, LM84, ST85].

Babaoglu and Drummond suggest another way to schedule synchronization events in [BD87]. They wanted to use information collected from ordinary communications to synchronize clocks. Instead of waiting for a predefined amount of time and then resynchronizing, their protocol uses a communication step called a Full Message Exchange (FME) as its event. A FME requires that each processor sends a message to all other processors. The arrival times of the messages are recorded and a new clock is started using the average of these times. The only time a specific synchronization round is needed is when a predefined period passes without a FME. This is an excellent solution to the overhead problem commonly associated with synchronization protocols. When the system is busy, synchronization data is "piggybacked" on messages carrying other information. The only synchronization overhead involved is the time necessary to adjust the virtual clocks. The only time that a specific slot has to be allocated to synchronization is when the system is idle and the extra overhead does not matter. The problem with the protocol as specified in [BD87] is that since it only uses information from one round of messages, it is not Byzantine resilient. The protocol therefore needs to be modified to use $f+1$ rounds of information before it can provide the desired fault-tolerance.

7.3.2. Reading virtual clocks on other processors

Since processors only have access to their own local clock time, they need a mechanism for reading the virtual clocks on all other processors so the convergence function can compute a new adjustment. The architectural constraint imposed by the use of authentication plus hardware limitations prevents a processor from reading all of the other virtual clocks simultaneously. This means that the convergence function cannot directly use values read from other processors as its clock arguments. Instead, each processor must approximate a virtual clock on another processor by reading the difference between the two clocks and adding it to its own [LM84]. Each processor therefore needs to keep a current table of the differences between its own virtual clock and the virtual clocks on other processors.

The way these differences are obtained is an integral part of each clock synchronization routine. The most basic techniques for finding the differences using Byzantine Agreement (BA) message passing rounds were described by Lamport and Melliar-Smith in [LM84]. The first two algorithms in the paper do not place any requirements on how clocks are read and thus $3f+1$ processors are needed to provide synchronization. The third protocol though is an extension of the *SM(m)* algorithm found in [LSP82] and discussed in Chapter 6. In general terms, the protocol sends virtual clocks which have been signed instead of messages. The algorithm accounts for the time that a message takes to be signed, sent, received, and verified at its destination by adding the product of a message transmission delay constant, γ, and the number of times the message has been relayed to the clock. Since authentication is used, a faulty processor cannot corrupt the value of a clock and thus set the clock forward; it can only delay sending the value and in effect, set the clock back. Figure 7.2 illustrates how a faulty processor can set a clock back by holding onto a message. The actual message transmission delay between two processors *i* and *j* (not counting the delay added by the fault) is represented by the variable, γ_{ij}. In the faulty example, processor *C* has no way of telling that the relay message from processor *A* has been delayed by the faulty processor *B* and would therefore use $(\gamma_{ABC} + delay)$ as one of the approximations for processor *A*'s virtual clock. Similar clock reading schemes are used by the protocols of [HSSD83], [LL83], and [ST85], except the number of messages needed is reduced by having each processor periodically broadcast its virtual clock, instead of waiting for a request [Sch87].

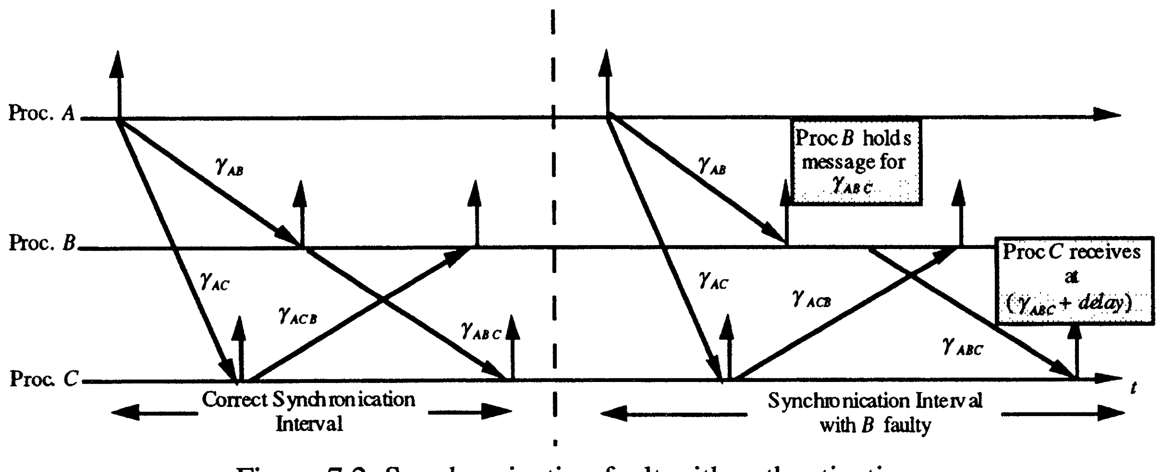

Proc. A
Proc. B
Proc. C
γ_{AB}
γ_{AC}
γ_{ACB}
γ_{ABC}
Correct Synchronication Interval
Proc B holds message for γ_{ABC}
Proc C receives at $(\gamma_{ABC} + delay)$
Synchronication Interval with B faulty
t

Figure 7.2 Synchronization fault with authentication.

7.3.3. *Convergence Functions*

A convergence function (*CF*) uses the virtual clocks of all processors in the system to calculate a new adjustment to a specific processor's hardware clock. Such a function does not guarantee that the virtual clocks on different processors will have the same value. The only requirement is that the new virtual clocks provided by the convergence function are closer together than the old virtual clocks. All convergence functions must have two basic properties [Sch87]. The first requirement is that the convergence function (CF) on processor p be monotonically non-decreasing in its last N arguments such that:

$$\text{If } (\forall i{:}1 \le i \le N{:}x_i \le y_i), \text{ then } CF(p,x_1,\ldots,x_N) \le CF(p,y_1,\ldots,y_n).$$

This property is needed to make sure that the values from the RTS do not decrease. The convergence function (CF) must also be translation invariant so that values on different processors and at different times can be compared. Translation invariance means that:

$$CF(p,x_1+\upsilon,\ldots,x_N+\upsilon) = CF(p,x_1,\ldots,x_N)+\upsilon.$$

Mahaney and Schneider provided a means of comparing the different convergence functions in [MS85] when they defined their functions in terms of precision and accuracy. Table 7.1 is directly reproduced from [Sch87] (with small changes to keep variables within this thesis consistent) and compares the different convergence functions found in previous literature using these measures. The fault-tolerance degree, f (k in [Sch87]), specifies the number of significantly differing clock values resulting from faulty

Name	Fault-tolerance degree, f	Precision $\pi(\delta,\varepsilon)$ (k faults)	Worst Precision ($N \to \infty$, $k=f$)	Accuracy $\alpha(\delta)$
CF_{EA}	$(N-1)/3$	$3k\delta/N+\varepsilon$	$\delta+\varepsilon$	$4\delta/3$
CF_{FCA}	$(N-1)/3$	$2k\delta/N+\varepsilon$	$2\delta/3+\varepsilon$	$4\delta/3$
CF_{Mid}	$(N-1)/3$	$\delta/2+\varepsilon$	$\delta/2+\varepsilon$	δ
CF_{Avg}	$(N-1)/3$	$k\delta/(N-2k)+\varepsilon$	$\delta+\varepsilon$	δ
CF_{CCA}	$(N-1)/3$	$k\delta/N+\varepsilon$	$\delta/3+\varepsilon$	$4\delta/3$
$CF_{Byz}1$	$(N-1)/2$	2Λ	2Λ	δ
$CF_{FW}SE1^1$	$N-1$	$\Gamma_{\max}(1+\hat{\rho})/(1-\hat{\rho})$	$\Gamma_{\max}(1+\hat{\rho})/(1-\hat{\rho})$	$\Gamma_{\max}+2(\delta-\Gamma_{\min})$
$CF_{FW}SE2^1$	$(N-1)/2$	$\Gamma_{\max}(1+\hat{\rho})/(1-\hat{\rho})$	$\Gamma_{\max}(1+\hat{\rho})/(1-\hat{\rho})$	$\Gamma_{\max}+\delta-\Gamma_{\min}$

Table 7.1 Comparison of convergence functions [Sch87].

1 Assumes digital signatures.

processors that the function can use and still generate valid results. Values within a set skew, δ, of each other are considered to be correct. The error involved in reading the clocks is quantified as ε. The precision of a function, $\pi(\delta, \varepsilon)$, determines how close the values obtained by two different computations of a convergence function with varying arguments will be, so long as at least $N-f$ arguments are correct. The accuracy, $\alpha(\delta)$, specifies how close a final value obtained with $N-f$ correct values is to the "correct" real time.

The first four functions in Table 7.1 come from some of the earliest work in Byzantine resilient synchronization protocols. Most of the derivations of precision and accuracy limits for these function can be found in [MS85]. CF_{EA} is presented and analyzed in [LM85] with their interactive convergence algorithm and is the average of all clocks which are no more than a skew from a processor's own clock. CF_{FCA} is an extension of CF_{EA} suggested in [MS85] where all clocks which are within a skew of $N-f$ other clocks are averaged.

The next two convergence functions come from [DLPSW83] and are described as fault-tolerant, because the f highest and f lowest values are discarded. The worst case fault scenario in synchronization is that f of the clock values have been corrupted by faulty processors to either all be slow or all be fast. It is possible for intermediate values to be corrupted, but they would not cause the convergence function to return an invalid result. CF_{Mid} takes the midpoint of the range spanned by the function's remaining arguments while CF_{Avg} averages the leftover arguments. These first four functions do not place any requirements on how messages are sent and therefore have a high fault-tolerance degree in the table. If authentication is used, this fault-tolerance degree would fall to $(N-1)/2$.

The last four convergence functions in Table 7.1 are based on different interactive consistency algorithms. CF_{CCA} uses a cheaper message passing protocol known as Crusader's Agreement to disseminate its clock and thus provides a lesser degree of fault tolerance [MS85]. The final two functions, $CF_{FW}SE1$ and $CF_{FW}SE2$, use an optimized form of Byzantine Agreement, known as Fireworks Agreement, where all correct processors agree on the value of a single correct clock by causing all to terminate the protocol at approximately the same (real) time. Both Crusader's and Fireworks Agreement are cheaper protocols in terms of performance, but they also provide a weaker form of fault tolerance than Byzantine resilience. CF_{Byz} first appears in the *CSM(m)* algorithm found in [LM84] and uses authentication and the fact that clocks can only be set back by faulty processors to select its value. Since the use of digital signatures prevent faulty processors from setting a clock forward, CF_{Byz} chooses the $(f+1)^{th}$ greatest clock. Note from Table 7.1 that the precision of this function is independent of the skew between the clocks. Instead, the precision is a function of Λ, the maximum clock reading error between two clocks. This maximum clock reading error accounts for both the error introduced by the mechanics of reading another processor's virtual clock and drift between processors.

7.4. Final Protocol Design

The Synchronization Layer is placed between the Application and Message Passing Layer and the system's unsynchronized hardware clocks. The layer was the last part of the architecture to be designed, because it depended heavily on the structure of the interactive consistency protocol. Since a processor's virtual clock is single-source information, the method used to disseminate its value between the other processors needs to be Byzantine resilient. The best and certainly the easiest solution to this problem is to extend the interactive consistency mechanism to handle synchronization.

The clock synchronization protocol used in this proposed design is derived from ideas in [LM84] and [BD87]. Chapter 6 describes the implementation of a *From_all* message passing routine which distributes single-source information from every processor in the system using the *SM(m)* algorithm in [LSP82]. Lamport and Melliar-Smith show how to implement a clock synchronization protocol using the above algorithm in [LM84]. The only major difference between their protocol and the our proposed version comes from the observation that a *From_all* provides a Byzantine resilient form of the FME suggested by Babaoglu and Drummond in [BD87]. The final protocol uses information gleaned from a regular *From_all* to perform synchronization, only employing a specialized synchronization exchange called *From_all_sync* when a predefined amount of time passes without a *From_all.*

In the next section, a more in-depth description is provided to show that the protocol follows the steps required by Schneider's clock synchronization paradigm. Then, the extension needed for the Initial Synchronization (ISYNC) period required to synchronize the clocks to within the required initial skew when the system is first started is discussed. Finally, a methodology for calculating the precision of the protocol is explained.

7.4.1. Description of the Clock Synchronization Protocol

Schneider's paradigm states that clock synchronization protocols are completely defined by their solutions to the three subproblems described earlier in this chapter. Our protocol implements two different types of synchronizations: "no-cost" synchronization, where timing information is piggybacked on other messages, and "required synchronization," where a special message passing exchange is performed just to distribute synchronization data. The timing of synchronization events and the type of synchronization used is highly variable, depending on the tasks being run by the system. Both types of synchronization use the Byzantine Agreement technique described by [LM84] in their *CSM(m)* algorithm to read the virtual clocks on other processors. Finally, since a Byzantine Agreement protocol with authentication is utilized, CF_{Byz} is used to calculate the new time adjustments. Outlining the solutions to each one of these subproblems provides an in-depth description of the whole protocol.

Before the protocol can be discussed in detail, a description of the task environment is required. Currently, only two tasks are assumed to be running: a generalized application task and when necessary, a required synchronization task. The different tasks are periodically allocated frames, or intervals of (virtual) time, in which the task is assumed to complete all of its operations. The application task is given R virtual seconds to finish, while the required synchronization task receives S virtual seconds. At the end of this (virtual) time, an interrupt goes off sending the processor to the next task. Within each frame, the task operates nondeterministically, in that it completes each instruction in order but not at predetermined virtual times. This means that tasks on different processors may execute at different rates. Figure 7.4 illustrates how the same task frame running on three different processors might look in terms of the

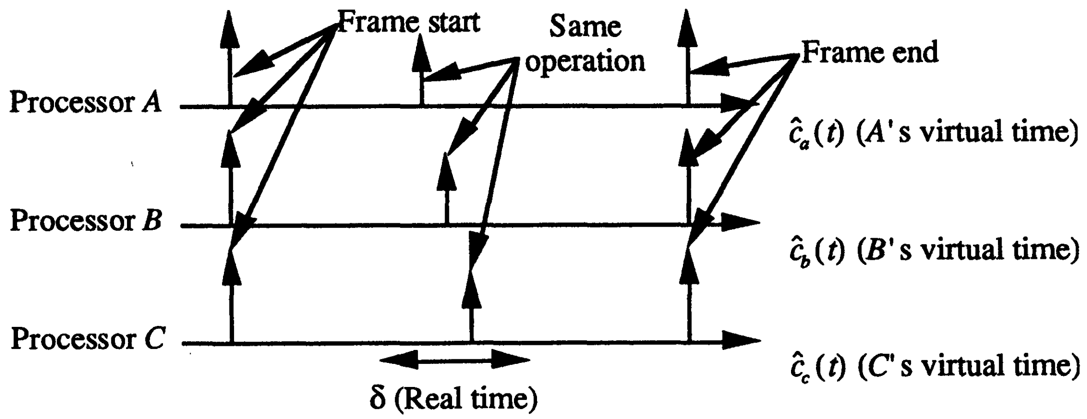

Frame start
Same operation
Frame end
Processor A
Processor B
Processor C
$\hat{c}_a(t)$ (A's virtual time)
$\hat{c}_b(t)$ (B's virtual time)
$\hat{c}_c(t)$ (C's virtual time)
δ (Real time)

Figure 7.3 Task frame behavior.

virtual time on each processor. The frame start and end occurs at the same virtual time due to an interrupt going off, but since execution rates on the processors vary, the same operation within the task happens at different virtual times. The only requirement is that the operations be performed on each processor within a known skew of (real) time.

The time at which a synchronization event is generated is determined by the placement of *From_all* exchanges within the task. At the beginning of each application frame, an interrupt based on the processor's virtual clock is set to jump to a required synchronization frame in which a special synchronization *From_all_sync* exchange is performed. If the application does a *From_all* exchange during its frame, it adjusts its virtual clock using no-cost synchronization. Since synchronization has already been done, the required synchronization frame is not necessary and the interrupt is reset to jump to an application frame. Figure 7.4 illustrates how the tasks might be scheduled on a processor. During the first two application frames, a *From_all* exchange is carried out, so the next frame is also an application frame. In the third application frame, no *From_all* exchanges are performed, so a required synchronization frame with a *From_all_sync* exchange follows. This design should cause a significant reduction in overhead, since specific frames are not needed for synchronization when the machine is busy. The only time synchronization engenders a large overhead is when the system is idle and multi-source interactive consistency exchanges (another cause of overhead problems) are not present.

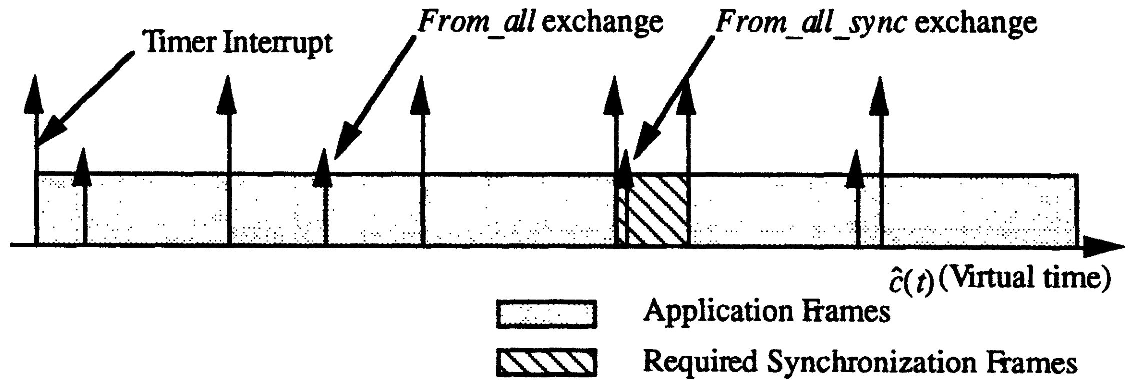
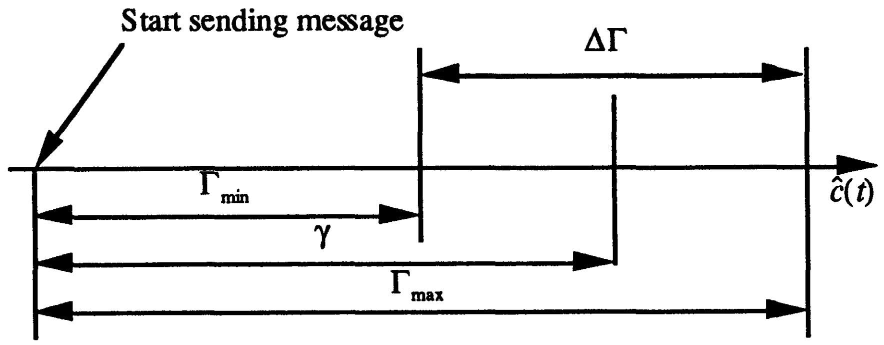


Figure 7.4 Application and required synchronization frames.

Once the processors have decided to synchronize, they must read the virtual clocks on all of the other processors. These virtual clocks are approximated using the processor's own virtual clock and the time difference between the two clocks. The time differences are collected during either a *From_all* exchange if no-cost synchronization is being performed or a *From_all_sync* if required synchronization is needed. The only difference between the two routines is that no actual message is sent during a *From_all_sync* exchange, just the header and trailer of a zero-byte message. Each message is given a timestamp using the virtual clock of the sender. The arrival time of each message is also recorded according to the receiving processor's virtual clock. An estimation of the time difference between two processors is calculated using the following equation:

$$\tau_p[q] = T - \hat{c}_p(t_{arrival}) - r\gamma, \tag{7.1}$$

where $\tau_p[q]$ is an estimation of the difference between the virtual clocks on processors *p* and *q* as seen by processor *p*, *T* is the timestamp in the message from processor *q*, $\hat{c}_p(t_{arrival})$ is the message's arrival time according to processor *p*'s virtual clock, *r* is the number of the current round within the exchange (1 or 2), and γ is the message transmission delay constant. The message transmission delay constant, γ, is an estimation of amount of time needed to sign, send, receive, and authenticate a message. Figure 7.5 shows the different message delays and the message delay error. The (virtual) time needed to process a message is bounded by the constants, Γ_{min} and Γ_{max}. The message delay error, $\Delta\Gamma$, is the difference between the minimum and maximum delays, Γ_{min} and Γ_{max}, while γ is the average of Γ_{min} and Γ_{max}. The error of introduced by using γ to estimate the actual message delay is therefore bounded by $\pm\Delta\Gamma/2$.



Γ_{min}	: Minimum tansmission delay
γ	: Message tansmission delay constant
Γ_{max}	: Maximum tansmission delay.
$\Delta\Gamma$	: Message delay error

Figure 7.5 Message transmission delays and message delay error.

Once estimations of the difference between all of the virtual clocks have been collected, convergence functions (*CF*) are used to calculate a new time adjustment for a processor's hardware clock. In this implementation, the convergence function, CF_{Byz}, is used within the interactive consistency exchange. During the message passing, each processor gathers $f+2$ estimations of the difference between its own virtual clock and every other processor's virtual clock, with the difference between a processor's clock and itself being set to zero. CF_{Byz} is applied to these arguments and returns the $(f+1)^{th}$ greatest difference. Once an estimation of every processor's virtual clock has been made, the processor has a set of *N* virtual times, one from each processor in the system which are used to calculate the new time adjustment. A convergence function such as CF_{EA} can then be used to generate this time adjustment.

7.4.2. Initial Synchronization (ISYNC)

The problem of initially synchronizing the clocks when the system first starts is much more complicated than the issue of resynchronizing clocks which are already within a skew of each other. The designer can assume that all processor will start within a given interval of time, μ, such that

$$\hat{c}_i(0) \le \mu \text{ for } i = 1 \ldots N, \tag{7.2}$$

but this interval must be a large amount of time. Much more care is needed to guarantee that messages sent by another processor during previous round do not overlap into a new round. The algorithm which is used during the system's Initial Synchronization (ISYNC) period is an extension of one found in [LL83].

Lundelius' and Lynch's ISYNC algorithm adds an additional step to each message passing round. ISYNC rounds still consist of (virtual) time intervals when messages are sent and then each processor waits long enough to guarantee that all of the messages from other processors have arrived. The difference is that each processor then waits for a second interval to be certain that new messages are not received by other processors in their first waiting period and then sends a READY message to all other processors indicating that it is prepared to begin the next round. However, if a processor receives $f+1$ READY messages before it completes its second waiting interval, it ends the interval early and sends its own READY message. The extra message passing guarantees that no clock values from the previous round will be received after a processor has begun a new round. The number of ISYNC rounds required to bring the system's virtual clocks within a desired skew is determined by the precision of the convergence function, CF_{Byz}, and the synchronization protocol. This process is discussed in the next section.

7.4.3. Determining the Precision of the Synchronization Protocol

Determining the precision of a clock synchronization protocol involves finding the maximum amount of skew which can occur between the virtual clocks in the system, assuming that the clocks were synchronized before the last resynchronization period. This maximum skew places a lower bound on the desired skew parameter, δ, around which the clock synchronization protocol is designed. The process of deriving this precision must take into account characteristics of both the protocol itself and the system (and therefore hardware) on which it is running. An assumption is made that the protocol operates "correctly." A proof of the correctness of the proposed protocol, based on a proof provided in [Sch87], is found in Appendix B. Actually finding a realistic value for the lower bound of δ, or $\delta_{\min}$, is a separate process.

The worst skew between virtual clocks happens just before a resynchronization interval. Therefore, deriving the precision of the clock synchronization protocol is done in two steps. First of all, the worst-case precision of the virtual clocks immediately after a resynchronization must be found. Then, the maximum amount of drift which can occur before the next resynchronization interval needs to be added. The maximum amount of skew found just before the resynchronization interval is the sum of two values such that

$$\pi(\delta,\varepsilon)+\xi=\delta_{\min}\leq\delta, \tag{7.3}$$

where $\pi(\delta,\varepsilon)$ is the precision of the convergence function (and therefore the virtual clocks immediately after the protocol makes its time adjustment) and ξ is the maximum amount of drift in (real) time between resynchronizations.

Before any of these parameters can be derived, the different time frames of reference need to be explained more thoroughly. These different frames of reference complicate the concept of clock synchronization. Each frame of reference is layered on top of one another, and they tend to drift apart when left alone. The sole purpose of clock synchronization is to continuously drag these frames back together.

There are four frames of reference which must be taken into account when analyzing this protocol. The first three are standard to all clock synchronization protocols. First of all, a Newtonian, or real time (t), frame is assumed. Real time is not directly observable, but is more of an abstraction around which the other frames are built.

The second time frame, known as hardware time ($c(t)$), is linked to real time by the hardware drift rate condition which says that:

$$0 < 1-\rho \le \frac{c_p(t+\kappa)-c_p(t)}{\kappa} \le 1+\rho, \text{ for } 0 \le t, \tag{7.4}$$

where κ is the hardware clock tick width and ρ is the hardware drift rate. For most hardware clocks based on crystal oscillators at constant temperature, the hardware drift rate, ρ, is around 10^{-4} $\left[\text{sec}/_{\text{sec}}\right]$. The third time frame, known as virtual time ($\hat{c}(t)$), is layered on top of hardware time. Virtual time drifts from hardware time due to error in reading the counter which is implementing the virtual clock. At any one time, the virtual clock can be up to half of one least significant bit (*lsb*) of the virtual clock away from the hardware clock.

Figure 7.6 illustrates how these three time references would look on a $\left[\text{sec}/_{\text{sec}}\right]$ coordinate system. Real time does not drift on this coordinate system so it is represented by the line which intersects the (1 sec, 1 sec) point. Hardware time can drift from real time at ρ $\left[\text{sec}/_{\text{sec}}\right]$ which is represented by the solid lines branching away from the real time line. At 1 second, the only guarantee is that hardware time is within $\pm\rho$ seconds of real time. Added to this uncertainty is the virtual time drift which provides the dashed lines bordering the hardware time bounds. At 1 sec elapsed (real) time, virtual time is guaranteed to be within $\pm \frac{1}{2} lsb$ seconds of hardware time. This combined drift continues to widen as real time passes, unless a clock synchronization protocol is used to bring the virtual clocks closer together again.

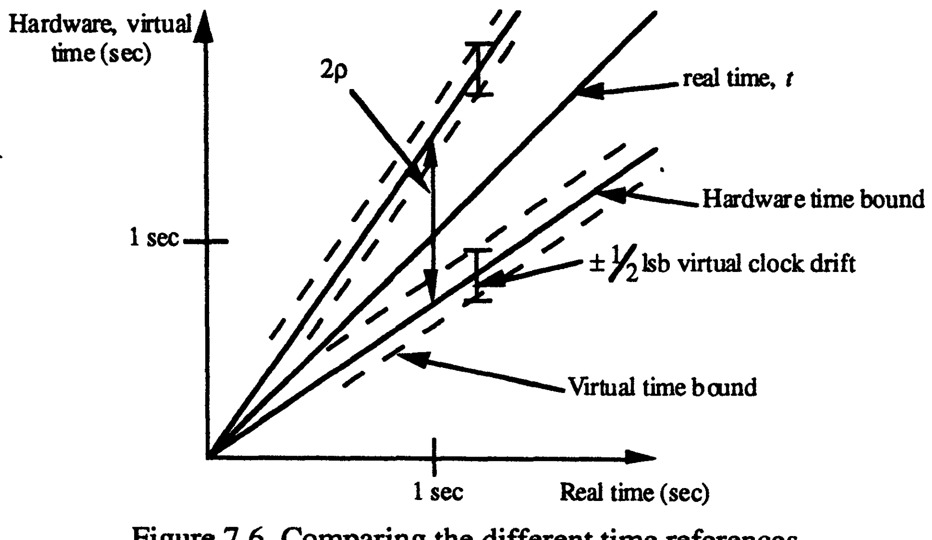
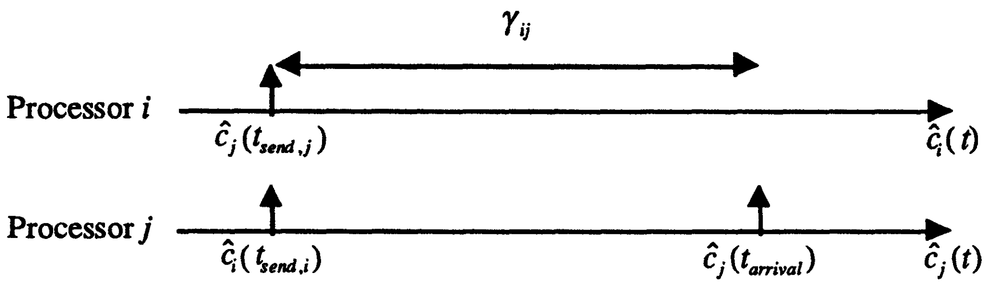


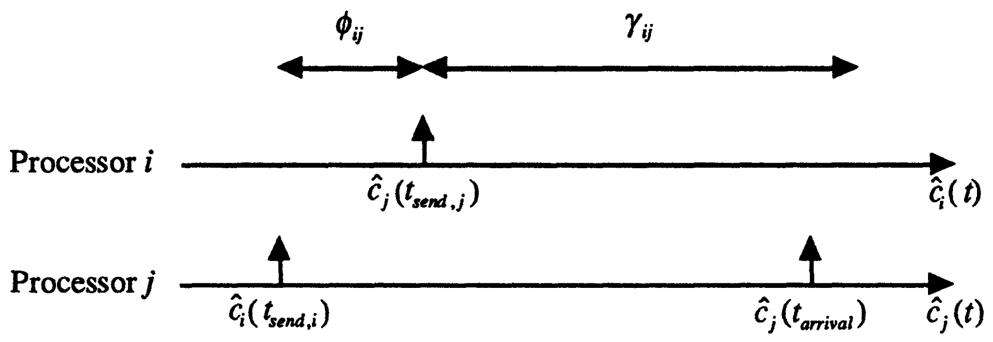
Figure 7.6 Comparing the different time references.

The last time frame of reference is required due to the way tasks run within their frames. Task operations are not deterministically scheduled to occur at specific virtual times. The task simply executes each instruction in order as the task reaches that point in its routine. Task execution rates may vary on the different processors due to such intangibles as the processor's temperature, error state, age, or other characteristics, causing each processor to reach the same event at different virtual times. Since no-cost synchronization is carried out in conjunction with an application task performing a *From_all* exchange, the virtual time at which synchronization is done can therefore be different on each processor. This is a significant change from previous designs where synchronization intervals were scheduled to begin at the same virtual times on each processor [HSSD83, KSB85, LL83, LM84, PB86, Sch87]. Figure 7.7 illustrates the difference that this makes in message passing. In Figure 7.7(a), processors *i* and *j* send messages when their virtual clocks reach their idea of the real time, t_{send}. These events occur on the two processors at the same virtual time (relative to the specific processor) but at the different real times, $t_{send,i}$ and $t_{send,j}$. When processor *j* receives the message at virtual time, $\hat{c}_j(t_{receive})$, the (virtual) time difference between when processor *i* sends the message and when processor *j* receives it is simply the transmission delay, γ_{ij}. In Figure 7.7(b), processors *i* and *j* send their messages when their virtual clocks reach their idea of different real times ($t_{send,i}$ and $t_{send,j}$, respectively). These events not only happen at different real times, but at different virtual times on each processor. This task execution drift adds a term, ϕ_{ij}, to the (virtual) time difference between when processor *i* sends the message and when processor *j* receives it. The task execution frame of reference has one major difference between it and the hardware and virtual time frames of reference. The task execution drift is reduced to zero every time a new frame is begun, because each frame boundary is deterministically scheduled using an interrupt. Therefore, the task is begun at the same virtual time on each processor. In order to quantify this drift, the designer needs to find a maximum bound, Φ, for the number of instructions/application frame (assuming that $R > S$) by which the tasks can differ. The time length of the frame (R seconds for an application frame) can be used to translate this drift into the virtual time frame of reference.

Once these time frames of reference are understood, the process of determining the precision of the clock synchronization protocol can begin. The first step is to determine the skew between the virtual clocks immediately after a resynchronization by quantifying the precision of the convergence function. Our proposed protocol uses CF_{Byz} which guarantees that virtual clocks are within 2Λ, twice the maximum clock reading error, of each other. Therefore, the important parameter which must be quantified is Λ.



(a) Message passing with virtual time-based synchronizations ($\hat{c}_i(t_{send,i}) = \hat{c}_j(t_{send,j})$ and $t_{send,i} \neq t_{send,j}$).



(b) Message passing with task execution-based synchronizations ($\hat{c}_i(t_{send,i}) \neq \hat{c}_j(t_{send,j})$ and $t_{send,i} \neq t_{send,j}$).

Figure 7.7. Virtual time- vs. task execution-based synchronizations.

The maximum clock reading error, Λ, is made up of the maximum amount of message delay error which is possible plus the maximum amount of task execution drift which can occur. Message delay error, $\Delta\Gamma$, is shown in Figure 7.5. The error is composed of the variable time which is needed by both the sender and the receiver to process a message. In order to get a realistic value for this parameter, the designer must determine the minimums and maximums for the following times:

1) the time needed to sign the message, to include generating the CRC and multiplying the CRC by the private key (signing time),
2) the time needed to send the message (sending time),
3) the time needed for the message to travel to the receiving processor (propagation delay),

4) the time needed for the receiver to realize that a message has arrived and to collect the message (receiving time), and

5) the time needed to authenticate the message, to include generating the CRC and multiplying the CRC by the public key (authentication time).

The estimated message delay is guaranteed to be within $\pm\Delta\Gamma/2$ of the actual delay. In previous designs, the maximum message delay error would be the same as the maximum clock reading error. In our proposed protocol though, task execution drift must be accounted for. In Figure 7.5, the different times are in terms of one processor's virtual clock. In order to guarantee that every processor has reached its own virtual time when it sends messages, the message delay error must be corrected for worst-cast task execution drift. This worst-case drift occurs at the end of an application frame. At this point, the task execution drift is bounded by $R\Phi/m$ virtual seconds, where R is the length of the application frame in virtual seconds, m is the maximum number of instructions in the application frame, and Φ is the bound on execution drift in terms of instructions per frame. Therefore, the worst-case precision of the protocol after a resynchronization is

$$\pi_{\max}(\delta,\varepsilon) = 2\left(\Delta\Gamma + R\Phi/m\right). \tag{7.5}$$

Once a value for $\pi_{\max}(\delta,\varepsilon)$ is known, finding the precision of the clock synchronization protocol involves correcting for the maximum amount of drift which can occur between the clocks before the next resynchronization. The worst-case scenario for the time between resynchronizations is when an application synchronizes its virtual clocks at the very beginning of one application frame and then does not perform a *From_all* exchange during the next application frame. This means that the clocks are not resynchronized until the end of the following required synchronization frame, an interval bounded by $2R+S$ virtual seconds. The virtual clocks can therefore drift a total of

$$\xi = 2\cdot\rho\cdot(2R+S) + \left(\tfrac{1}{2}lsb\right) + S\Phi/m, \tag{7.6}$$

with the $S\Phi/m$ term added for task execution drift since the resynchronization occurs at the end of the required synchronization frame. Using Equation (7.3), the lower bound for achievable skew is thus

$$\delta_{\min} = 2\Delta\Gamma + 2\rho(2R+S) + \left(\tfrac{1}{2}lsb\right) + (2R+S)\Phi/m. \tag{7.7}$$

8. Conclusions and Recommendations

8.1. Overview

This architecture study provides the groundwork for implementing a new generation of Byzantine resilient processors using authentication. A layering scheme is proposed which can be placed between the user and hardware. These layers are made up of protocols which provide the basic building blocks of the architecture. The final design proposed for each protocol is described in the following section. The final section discusses topics which still need to be researched before the system can be fully implemented.

8.2. A Byzantine-Resilient Architecture using Authentication

The proposed architecture is built around the use of digital signatures to authenticate the origin and contents of messages. The use of authentication allows a significant reduction in the theoretical requirements necessary for providing Byzantine resilience, or the ability to continue correct operation in the presence of arbitrary or even malicious faults. This decrease in the requirements led to the goal described in Chapter 2 of providing a system which combines the stringent standards embodied by Byzantine resilience with the lower costs necessary to make the system viable for more markets than previous Byzantine resilient processors.

An investigation of the basic building blocks required by the proposed architecture is found in Chapter 3. The result of this investigation is the layering scheme shown in Figure 8.1. These layers are designed to be placed between the user and the system's hardware, keeping the mechanics involved in providing Byzantine resilience invisible to the user. Each layer is composed of one or two protocols responsible for providing services for the layer. The design of these protocols is described in Chapters 4, 5, 6, and 7 and serves to specify the proposed architecture.

Chapter 4 covers the design of the authentication protocol, the fundamental building block upon which the rest of the architecture rests. First of all, an in-depth study of the theoretical and practical requirements which must be fulfilled by the protocol is done in Section 4.2. The two different options for authentication protocols, private-key and public-key authentication, are described with the reasoning behind choosing public-key protocol for this proposed architecture in Section 4.3. The design of the protocol is

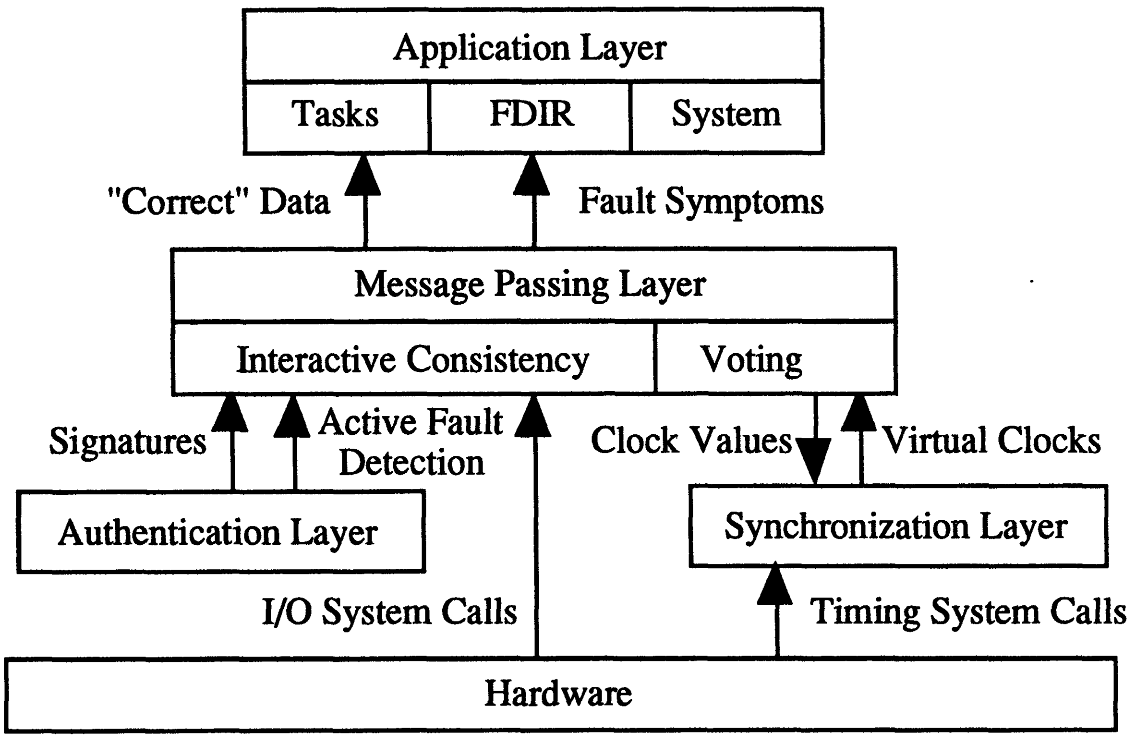


Figure 8.1 Layering scheme for the proposed architecture.

then broken into the three functions required by public-key authentication: the encoding, signing, and authenticating functions. The encoding function generates a n-bit representation of the message for the signing function to sign with a private key to get the digital signature. The authenticating function uses this signature and a public key related to the private key to provide a result which can be compared with an encoded n-bit representation of the received message to authenticate the message. A method for generating private and public key pairs using modular inverses is described in Section 4.4. This is followed in Section 4.5 by a discussion of the issues involved in using Cyclic Redundancy Codes (CRC's) to implement the encoding function. The final design of the authentication protocol proposes employing a 32-bit CRC generator polynomial to encode the message with 32-bit modular inverse keys used by the signing and authenticating functions. A 32-bit CRC does not provide the reliability imparted by the 64-bit generator polynomial derived in Section 4.5.1, but its implementation requires less performance overhead to authenticate messages. A corollary to the decision to use the faster 32-bit CRC is the suggestion that the functions in the authentication protocols be optimized as much as possible by implementing them using the assembly code of the specific processors upon which the system is running.

The interactive consistency protocol found in the Message Passing Layer is described in Chapter 5. This protocol is responsible for distributing data between processors in a way which satisfies the Byzantine Agreement conditions found in Section 3.3. The protocol provides the Byzantine resilience for the entire system as it prevents faulty processors from influencing the data on nonfaulty processors. Section 5.2 investigates the requirements placed on a system using authentication which desires to provide interactive consistency. This discussion is followed by a description of Byzantine Agreement algorithms in Section 5.3 which concentrates on the *SM(m)* algorithm from [LSP82] and the improvements suggested in [DS83]. The following section examines the various design issues involved in implementing this algorithm which are then incorporated into the final protocol design provided in Section 5.5. Section 5.5 describes the structure of four routines which make up the design of the protocol: three single-source exchanges (*From_a, From_b,* and *From_c*) and a multi-source exchange *(From_all)* which distributes data from all of the processors. An implementation of parts of the protocol in Section 5.5.2 resulted in a final design proposal for the interactive consistency protocol which employs the point to point message passing of the *SM(m)* algorithm without the algorithm's use of nested signatures. An additional result of this study is the realization of the amount of performance overhead taken by the I/O system calls used to send and receive messages and the need for optimization of these routines in any final implementation.

The second protocol of the Message Passing Layer is the voting protocol discussed in Chapter 6. This protocol is responsible for generating a group consensus value which is guaranteed to be the same on all nonfaulty processors. The first step in designing the protocol, found in Section 6.2, involved examining the requirements of the voting protocol and making certain that they did not conflict with the requirements set by the interactive consistency protocol. Section 6.3 describes the basic voting algorithm with its message passing interval and *choice()* function, followed by an investigation of the design issues involved in implementing these sections of the protocol. Section 6.4 describes the structure of a routine which implements the voting protocol. An implementation of parts of this routine in Section 6.4.2 resulted in a final design proposal for the voting protocol which exchanges unsigned messages and then uses a full-set majority vote *choice()* function to calculate the group consensus value.

The synchronization protocol discussed in Chapter 7 implements the final layer of the architecture, the Synchronization Layer. This protocol provides the synchronized clocks which are needed to guarantee that tasks will terminate even when a processor becomes faulty and stops sending messages. In Section 7.2, the clock synchronization problem is split into three subproblems using a paradigm found in [Sch87]. The solutions to these subproblems completely specify a clock synchronization protocol. The design issues involved in implementing the subproblems are discussed in Section 7.3. Section 7.4 describes a method for synchronizing clocks which minimizes the amount of performance overhead needed by the protocol. Two forms of synchronization are proposed: no-cost synchronization and required synchronization. The no-cost synchronization is performed using information which has been "piggybacked" on the messages of a normal multi-source interactive consistency *From_all* exchange. Required synchronization is only needed when an application frame finishes without performing a *From_all* exchange. When this occurs, a required synchronization frame is allocated for a *From_all_sync* exchange. Once approximations of the virtual clocks on other processors are gathered, a time adjustment is calculated using a technique suggested by the *CSM(m)* algorithm from [LM84]. The final part of the chapter, Section 7.4.3, describes the how to determine the precision of the synchronization protocol, taking into account characteristics of both the protocol itself and the system (and therefore hardware) on which it is running. This process must account the different types of drift present in the system, including task execution drift which is a problem not present in previous clock synchronization protocols.

8.3. Topics for Further Research

There are a number of areas which need to be researched before this architecture can be fully implemented. This study only examines the protocols which make up the basic building blocks of the architecture. Many more features need to be added to the design before the final system can be built.

The most obvious area which needs to be worked on is the Application Layer. The scheduling paradigm used in Section 7.4.1 to describe the clock synchronization protocol is far too simplistic for any full implementation. Issues such as preemptive scheduling have to be examined. Also, a Fault Detection, Isolation, and Recovery (FDIR) task needs to be designed which makes use of the special characteristics of this architecture. For example, one of the decisions made in designing the interactive consistency protocol is to not use nested signatures during regular exchanges. FDIR could turn on a nested signature feature to help isolate where a fault has occurred.

An additional area which needs to be examined is the relationship between the architecture layers and the hardware upon which they are running. A constant theme resulting from the performance overhead measurements is the necessity of optimizing the

I/O system calls which are used to send and receive messages over the links connecting the processors. The time spent in these system calls increases the message processing overhead during each message exchange. It is important that this message processing overhead be kept low, because it determines how long the rounds within each exchange have to be to make sure no messages are dropped. Another area which is dependent on the hardware is the precision of the clock synchronization. Section 7.4.3 describes the steps needed to determine this quantity, but the process needs to be verified using actual hardware.

Appendix A 64-Bit Modular Inverses

Number	Private Signature Key (P)	Public Signature Key (P^{-1})
Channel A.	E7767EDF DD6B0AA9	6A6FFF29 DFCF3999
Channel B	300FEC3E 66358FED	6A554369 724D25E5
Channel C	80FD00BF 6AA91DFB	4DEE4FB 4816533

Table A.1 List of 64-bit modular inverses used in the implementation.

Number	Private Signature Key (P)	Public Signature Key (P^{-1})
1.	9C1E8A09 615B11EB	53170A9C EDD38EC3
2.	89E1143E 72908DE5	15791CE 3567E7ED
3.	F795B5A2 3931E005	6FF0B378 6B4FECCD
4.	1FF06382 C836B5B5	6DF4AEBD 5142509D
5.	8C79563D B40701D7	507FF3C7 F04081E7
6.	F44A41EE 42F13DC9	5EA41492 27216C79
7.	3811B26 7F04F271	5D91E9B2 37388E91
8.	EEECFE42 D516023B	6C66FD3B 4CC386F3
9.	F290151B 771CCFDF	2F366CB6 8F47AC1F
10.	E45697F9 7687760B	5507BB2A D278E5A3
11.	89D2143 ECFF2F3D	3E24754B 5067A015
12.	A75E386F A1E71E43	E165F06 A0539E6B
13.	F6E2FC72 64A95DF1	721B5D3D C4E5F311
14.	A0C76A8C 6D6AB859	5737DBB5 5F790FE9
15.	E32623C6 28E15553	21F4BE57 5BC1B6DB
16.	3BC330AD B095EC6F	7A66AAE8 6940F28F
17.	E022AE85 4AB669A9	663AE420 10F04A99
18.	C61C94FE 2BA18883	6AED9721 14B9062B
19.	B8B66F5D 627067E7	75CC1313 8855BBD7
20.	95694E7C 588C0ACF	5A90010C 6286BC2F
21.	96529F8D FDC508E5	6407D85D 786374ED
22.	1CC59F7F FF59489	7DD04763 7FD221B9
23.	B77DC52F 6E2143B1	5CA750A1 A5012551
24.	AE677A59 8027C49	5F579CD1 1F0A35F9
25.	BBE13B79 65670CE5	FEF94F8 CDB3D0ED

Number	Private Signature Key (P)	Public Signature Key (P^{-1})
26.	C787CD21 928A0A8F	7EB7CA01 83A9D46F
27.	DFC61DDB 7C881655	6852A610 299C36FD
28.	2A49A338 1B9E3CD7	3B58ACB8 EBB476E7
29.	14A7E74C D115D5AD	7C1342A9 DC505625
30.	458014C7 EFC1CC39	41546A5C 36126209
31.	9EDAED4D 9F510EBB	43768199 A7DA0673
32.	56C28594 552D3373	2B8F9744 C6CA29BB
33.	D711FA5 62C6DF39	341D21EB DF8C5F09
34.	A960A010 E33B73EF	5D50D2BD 8561B0F
35.	869D6590 CD16A0DF	70276B0F B151B1F
36.	36F8E37 D655D2DB	296ECA85 D3A3D953
37.	5C8D110E 4A5A0C15	1BE9B869 5323633D
38.	C4CABCC1 54EA1DF	7A55A8FF 6FAE5A1F
39.	C32B7D71 A7864A29	959C0C1 6C88F219
40.	1F351E66 F33A1059	5772D503 674437E9
41.	25BE4471 2EC32645	7D552145 EAF8FC8D
42.	57758D69 9390DB1D	642B67E1 3F28BF35
43.	4C167AFB C3836C5	5DCAC7E2 FE23D80D
44.	54EF6434 392EA25	6B9A2E6A 869521AD
45.	8C190FEE 30EDC787	73EAA00B 9EA94E37
46.	C5D8D71E 4E5FFDA7	242DBC7B B9A0DA17
47.	B5463794 19D93187	7F94D135 9AFDC437

Table A.2 List of other 64-bit modular inverses.

The actual clock synchronization protocol is represented in the model by the reliable time source (RTS). The three subproblems from the paradigm are formally defined by the functions performed by the RTS. First of all, the RTS is responsible for generating the synchronization event. This function is formalized in terms of $r_{\min}$, $r_{\max}$, and β as [Sch87]:

RTS1: The RTS generates synchronization events at real times $t^1_{RTS}, t^2_{RTS} \ldots$ such that

$$\left(t^1_{RTS} = 0\right) \wedge \left(\forall i{:}0 < i{:}r_{\min} \le t^{i+1}_{RTS} - t^i_{RTS} \le r_{\max}\right)$$

and the real time t^i_p at which processor p detects the event produced at t^i_{RTS} satisfies

$$\left(t^1_p = 0\right) \wedge \left(\forall i{:}0 < i{:}0 \le t^i_p - t^i_{RTS} \le \beta\right).$$

Secondly, the RTS reads the virtual clocks on the rest of the processors and uses a convergence function to provide a new time adjustment, or

RTS2: At t^i_p, processor p obtains a value V^i_p that can be used in adjusting $\hat{c}_p$ to be consistent with the Virtual Synchronization (B.3) and Virtual Drift Rate (B.4) conditions.

B.3. Formal Definition of our Clock Synchronization Protocol

The most important step in using Schneider's proof for a specific clock synchronization protocol is in showing that the protocol fulfills certain assumptions. These assumptions characterize the three subproblems of his paradigm and provide the only structure for the actual model. Once a protocol has been defined in terms of the required variables, it can replace the RTS within the model and therefore use the proof to demonstrate the correctness of its operation.

The first subproblem, generating the synchronization event, is mathematically described by RTS1. Providing bounds for the variables, $r_{\min}$, $r_{\max}$, and β, shows how the protocol fulfills this first function of the RTS. The parameters, $r_{\min}$ and $r_{\max}$, are the lower and upper bounds for the (real) time interval between when the first nonfaulty processor decides to resynchronize each time, while β bounds the (real) time which can elapse between when the first and last nonfaulty processor resynchronizes. The proof of correctness accounts for hardware clock drift, but any variables which depend on events occurring on separate processors must account for task execution drift and virtual clock drift. In Section 7.4.3, a bound on the greatest amount of task execution drift which can occur within an application frame was found to be $R\Phi/_{m}$, where Φ is a bound on execution drift in terms of instructions per application frame, R is the frame time in (virtual) seconds, and m is the maximum number of instructions in the application. This factor needs to be added (for upper bounds) to the (virtual) time necessary for the interval to be completed on one processor with respect to its virtual clock. Correcting for virtual clock drift is simply a matter of dividing this new interval by $(1+\hat{\rho})$ for a lower bound or by $(1-\hat{\rho})$ for an upper bound.

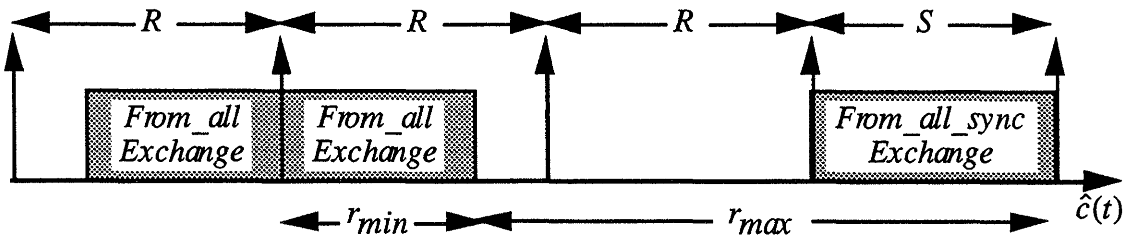


Figure B.1 Placing bounds on $r_{\min}$ and $r_{\max}$.

Placing bounds on $r_{\min}$ and $r_{\max}$ for the proposed clock synchronization protocol involves examining the scheduling of resynchronizations. Figure B.1 illustrates $r_{\min}$ and $r_{\max}$ for our protocol, in terms of one processor's virtual clock. Since resynchronizations occur every time there is a *From_all* exchange, $r_{\min}$ is simply the least amount of (real) time necessary to carry out one *From_all* exchange. A lower bound for $r_{\min}$ can be calculated by assuming that each round of the *From_all* exchange must be long enough for all of the other processors reach the point of sending their messages and then for the message to be processed. A lower bound on the length of a message passing round does not correct for task execution drift, since the drift is returned to zero at the beginning of every frame (due to the scheduled interrupts), not at the end of synchronization intervals. Since the *From_all* exchange uses two message passing rounds, a lower bound for $r_{\min}$ is

$$r_{\min} \geq \frac{2\left(\hat{\delta}+\Gamma_{\min}\right)}{\left(1+\hat{\rho}\right)}, \qquad \text{(B.5)}$$

where $\Gamma_{\min}$ is a lower bound on message transmission delay.

On the other hand, $r_{\max}$ is taken when a *From_all* exchange occurs at the beginning of an application frame, followed by an application frame without a *From_all*

Appendix B. Clock Synchronization Proof of Correctness

B.1. Overview

One of the most important results of using Schneider's clock synchronization paradigm to design a protocol is the ease of proving correct operation. In [Sch87], Schneider provides a proof of clock synchronization which, except for certain general assumptions, leaves solutions for his three subproblems open. Once a specific protocol is cast in term of these assumptions, the proof can be used to show that a protocol satisfies the correctness conditions. A number of corrections to Schneider's clock synchronization proof are suggested by Shankar in [Sha91]. Shankar redefined certain components in the system to remove the reliable time source (RTS), but these changes did not affect the actual proof. The differences between Schneider's and Shankar's results come from algebra errors and some latitude in Schneider's arguments which are removed in the new version [Sha91]. For reference, Appendix C contains a list of each of the variables used in this analysis with a definition of each.

In the following section, a more detailed description of Schneider's model is given with the definitions necessary for understanding the proof. Our clock synchronization protocol is then represented in terms of the assumptions about the three subproblems used in the proof. Once it is shown that our protocol fulfills these assumptions, it follows that the clock synchronization protocol operates correctly . The details of proving the individual theorems can be found in [Sch87] and [Sha91].

B.2. Schneider's Formal Model of the System

Before the clock synchronization proof can be discussed, the components making up the system need to be defined and mathematically characterized. One of the problems in analyzing the various clock synchronization protocols found in the literature is that the models used in the proofs vary depending on the author. Before any comparison of two protocols can be done, the differences in the models have to be accounted for. The proofs are hard enough to understand without having to deal with this extra complication. Schneider attempts to provide a model which is general enough to be used for all of these protocols. The system provided by the model is made up of virtual clocks implemented by a reliable time source (RTS) on top of hardware clocks. Correct operation of these components is defined as fulfilling certain requirements presented in terms of mathematical bounds.

Every processor has its own physical clock responsible for providing an idea of real time. The hardware clock on a correct processor is assumed to implement a function $c_p(t)$ on processor p which maps a real time t to a clock time. The initial value of each hardware clock is bounded by a constant μ such that

$$0 \le c_p(0) \le \mu. \tag{B.1}$$

The hardware clocks are implemented as counters which increase by one in response to periodic events known as ticks. The width of these ticks may vary as the clock advances, causing the clock's value to drift from real time. The amount that a hardware clock varies from real time must be bounded by

$$0 < 1-\rho \le \frac{c_p(t+\kappa)-c_p(t)}{\kappa} \le 1+\rho, \text{ for } 0 \le t, \tag{B.2}$$

where κ is the hardware clock tick width and ρ is the hardware drift rate. If a hardware clock conforms to these assumptions, it is considered to be nonfaulty. No such assumptions are made about faulty clocks.

Clock synchronization protocols implement a virtual clock function $\hat{c}_p(t)$ on processor p which maps a real time t to a virtual clock time. The requirements for nonfaulty operation provide the correctness conditions which the proof must show a protocol fulfilling. First of all, the difference between virtual clocks on correct processors p and q must be bounded so that

$$\textbf{Virtual Synchronization: } \left|\hat{c}_q(t)-\hat{c}_p(t)\right| \le \hat{\delta}, \text{ for } 0 \le t, \tag{B.3}$$

where $\hat{\delta}$ shows how closely the virtual clocks are synchronized. Also, the virtual clock must advance at a rate bounded by a virtual clock drift rate $\hat{\rho}$ so that

$$\textbf{Virtual Drift Rate: } 0 < 1-\hat{\rho} \le \frac{\hat{c}_p(t+\hat{\kappa})-\hat{c}_p(t)}{\hat{\kappa}} \le 1+\hat{\rho}, \text{ for } 0 \le t, \tag{B.4}$$

where $\hat{\kappa}$ is the virtual clock tick width and $\hat{\rho}$ is the virtual clock drift rate. It is important to note that this virtual drift condition relates virtual time to real time, instead of the relationship between virtual and hardware time used in Section 7.4.3.

exchange. The parameter, $r_{\max}$, encompasses the (real) time interval between the end of the *From_all* to the end to the required synchronization frame's *From_all_sync* exchange (See Figure B.1). An upper bound for $r_{\max}$ can be found by assuming that the first *From_all* exchange is completed instantaneously while the *From_all_sync* exchange uses the entire required synchronization interval. Finding this bound using frame boundaries removes the need to correct for task execution drift (tasks are assumed to complete before the end of the frame and task execution drift is returned to zero at the frame boundaries by the scheduled interrupts.). The resulting bound (correcting for virtual clock drift) is

$$r_{\max} \leq \frac{(2R+S)}{(1-\hat{\rho})}. \tag{B.6}$$

The final parameter used to describe clock synchronization event generation is β. Since β is an upper bound on the (real) time which can elapse between when the first and last nonfaulty processor resynchronizes, the parameter must account for the largest amount of task execution drift possible. When the fastest nonfaulty processors decides to resynchronize, the slowest nonfaulty processor's virtual clock must be within $\left(\hat{\delta} + {}^{R\Phi}\!/_{m}\right)$ real seconds of the fastest nonfaulty processor's virtual clock. Thus, the slowest nonfaulty processor might task as long as $\frac{\hat{\delta} + {}^{R\Phi}\!/_{m}}{1-\hat{\rho}}$ real seconds until it reaches its resynchronization point.

Specifying the remaining subproblems has already been covered in previous sections. The second subproblem involves reading virtual clocks on other processors. The only assumption made about this process is that an upper bound, Λ, can be placed on the error involved in reading another processor's clock. An explanation of how to quantify Λ is found in Section 7.4.3. The resulting bound is

$$\Lambda \leq \Delta\Gamma + {}^{R\Phi}\!/_{m}, \tag{B.8}$$

where $\Delta\Gamma$ is the message delay error and ${}^{R\Phi}\!/_{m}$ is a correction term for task execution drift. The final subproblem involves a convergence function which is described in terms of precision, π, and accuracy, α . A comparison of the different precision and accuracy of the various convergence functions can be found in Table 7.1. Since this protocol uses CF_{Byz}, both the precision and accuracy of the function are 2Λ.

$r_{\min}$	$\dfrac{2\left(\hat{\delta}+\Gamma_{\min}\right)}{\left(1+\hat{\rho}\right)}$
$r_{\max}$	$\dfrac{(2R+S)}{\left(1-\hat{\rho}\right)}$
β	$\dfrac{\hat{\delta}+{}^{R\Phi}\!/_{m}}{1-\hat{\rho}}$
Λ	$\Delta\Gamma+{}^{R\Phi}\!/_{m}$
π	2Λ
α	2Λ

Table B.1 Bounds on parameters specifying clock synchronization subproblems.

Appendix C. Glossary of Notation

This glossary contains a list of the notation used throughout this thesis. The numbers in parentheses after each definition show the section of the thesis in which the variable is defined.

adj_p	Adjustment added to processor p's hardware clock to implement its virtual clock. (7.2.1)
$c_p(t)$	Value of p's hardware clock at real time t. (B.2)
$\hat{c}_p(t)$	Value of p's virtual clock at real time t. (B.2)
$D_i(M_\#)$	Signing function ($P \cdot M_\# \bmod 2^n$) which uses a private key, P, for processor i to sign a message $M_\#$. (4.3.1)
$E_i(S_i)$	Authenticating function ($S_i \cdot P^{-1} \bmod 2^n$) which uses the public key, P^{-1}, for the private key on sender i to authenticate the signature, S_A. (4.3.1)
$e(x)$	Polynomial representing errors in a corrupted message. (4.5.1)
e	Number of bits in the block size used to feed data through a CRC generation function. Also, the number of bits in each entry of a CRC lookup table.(4.5.2)
f	Fault-tolerance degree, specifies the number of Byzantine faults masked by the architecture. (7.3.3)
$g(x)$	Generator polynomial for calculating CRC's. (4.5)
k	Number of bits in a message. (4.2)
M_{text}	A k-bit message to be signed or verified. (4.3.1)
$M_\#$	A n-bit representation of M_{text}. (4.3.1)
m	Maximum number of instructions in an application frame. (7.4.3)
N	Total number of processors in the system. (7.3.3)
n	Number of bits in a digital signature. (4.2)
P	Modular inverse of P^{-1}, used as a private key by $D_i(M_\#)$. (4.3.2)
P^{-1}	Modular inverse of P, used as a public key by $E_i(S_i)$. (4.3.2)
$r_{\max}$	Maximum real time between synchronizations. (B.2)
$r_{\min}$	Minimum real time between synchronizations. (B.2)
R	Length of an application frame in (virtual) seconds. (7.4.1)
S	Length of a required synchronization frame in (virtual) seconds. (7.4.1)
S_i	Signature appended to the end of a message sent by processor i. (4.3.1)
$S_k(M)$	Signaturing function which generates a signature for the message M based upon a private key k. (4.3.1)

$s(x)$	Polynomial representing a CRC. (4.5)
t	Real time. (7.4.3)
x^i	Polynomial representing a bit in the i^{th} bit position. (4.5)
α	Accuracy of a convergence function, specifies how close a final value obtained with $N-f$ correct values is to the "correct" real time. (7.3.3)
β	Bound on the real time between when the first and last nonfaulty processor resynchronizes. (B.2)
$\hat{\delta}$	Virtual skew, maximum allowable difference between the virtual clocks. (B.2)
Φ	Bound on the maximum amount of task execution drift in terms of number of instructions/application frame. (7.4.1)
ϕ_{ij}	Actual amount of task execution drift between processors i and j. (7.4.1).
Γ_{max}	Maximum message transmission delay. (7.4.1)
Γ_{min}	Minimum message transmission delay. (7.4.1)
$\Delta\Gamma$	Message delay error. (7.4.1)
γ_{ij}	Actual message transmission delay between processors i and j. (7.3.2)
γ	Estimated message transmission delay constant. (7.4.1)
κ	Hardware clock tick width, fixed (real) time interval between ticks of the hardware clocks. (B.2)
$\hat{\kappa}$	Virtual clock tick width, fixed (real) time interval between ticks of the logical clocks. (B.2)
Λ	Maximum clock reading error between any pair of processors. (7.4.3)
μ	Range of initial values of the hardware clock. (B.2)
π	Precision of a convergence function, determines how close values obtained by two different computations of a convergence function with varying arguments will be, so long as at least $N-f$ arguments are correct. (7.3.3)
ρ	Hardware clock drift rate, rate at which the hardware clocks drift from real time. (B.2)
$\hat{\rho}$	Virtual clock drift rate, rate at which the logical clocks drift from real time. (B.2)
$\tau_p[q]$	Estimation of the difference between the virtual clocks on processors p and q as seen by processor p. (7.4.1)
ξ	Maximum amount of total drift in (real) time between resynchronizations. (7.4.3)

References

[BD87] O. Babaoglu and R. Drummond, "(Almost) No Cost Clock Synchronization," *Proc. Seventeenth International Symp. on Fault-Tolerant Computing*, July 1987.

[DDS84] D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Systems," IBM Research Rept. RJ 4292 (46990), May 8,1984.

[DH76] W. Diffie and M. E. Hellman, "New Direction in Cryptography," *IEEE Trans. on Information Theory*, Vol. IT-22, No. 6, November, 1976.

[DLPSW83] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl, "Reaching Approximate Agreement in the Presence of Faults," *Proc. Third Symp. on Reliability in Distributed Software and Database Systems*, 1983.

[Dol83] D. Dolev, "The Byzantine Generals Strike Again," *Journal of Algorithms*, Vol. 3, 1983.

[DS83] D. Dolev and H. R. Strong, "Authenticated Algorithms for Byzantine Agreement," *SIAM Journal of Computing*, Vol. 12, No. 4, November 1983.

[FL82] M. Fischer and N. Lynch, "A Lower Bound for the Time to Assure Interactive Consistency," *Information Processing Letters*, Vol. 14, No. 4, 13 June 1982.

[FLP83] M. J. Fischer, N. A. Lynch, and M. S. Paterson., "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, Vol. 32, N. 2, Apr. 85.

[Gal90] R. Galetti, "Real-Time Digital Signatures and Authentication Protocols," Master of Science Thesis, Massachusetts Institute of Technology, 1990.

[HL91] R. E. Harper and J. H. Lala, "Fault-Tolerant Parallel Processor," *Journal of Guidance, Control, and Dynamics*, Vol. 14, No. 3, May-June 1991.

[HLS87] A. L. Hopkins, Jr., J. H. Lala, and T. B. Smith III, "The Evolution of Fault-Tolerant Computing at the Charles Stark Draper Laboratory, 1955-85," *Dependable Computing and Fault-Tolerant Systems, Vol. I : The Evolution of Fault-Tolerant Computing*, Springer-Verlag, Wien, Austria, 1987.

[HSSD83] J. Halpern, B. Simons, R. Strong, and D. Dolev, "Fault-Tolerant Clock Synchronization," *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, 1983.

[Knu69] D. Knuth, The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, 1969.

[KSB85] C. M. Krishna, K. G. Shin, and R. W. Butler, "Ensuring Fault Tolerance of Phase-Locked Clocks," *IEEE Trans. on Comp.*, Vol. C-34, No. 8, August 1985.

[Lee81] R. Lee, "Cyclic Code Redundancy," *Digital Design*, July 1981.

[LL83] J. Lundelius and N. Lynch, "A New Fault-Tolerant Algorithm for Clock Synchronization, "Proc. *3rd ACM Symp. on Principles of Distributed Computing*, 1983.

[LM84] L. Lamport and P. M. Melliar-Smith, "Byzantine Clock Synchronization," *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, 1984.

[LM85] L. Lamport and P. M. Melliar-Smith. "Synchronizing Clocks in the Presence of Faults," *Journal of ACM* , Vol. 32, No. 1, Jan. 1985.

[LSP82] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, July 1982.

[Mac88] D. A. Mackall, "Development and Flight Test Experiences with a Flight-Crucial Digital Control System," NASA Technical Paper 2857, Nov. 1988.

[MG78] D. L. Martin and D. Gangsaas, "Testing of the YC-14 Flight Control System Software," *Journal of Guidance and Control*, Vol. 1, No. 4, July-August, 1978.

[MS85] S. R. Mahaney, and F. B. Schneider. "Inexact agreement: Accuracy, Precision, and Graceful Degradation." *Proc. Fourth ACM Symp. of Principles of Distributed Computing*, 1985.

[NS78] R. M. Needham and M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," Xerox Report CSL-78-4, Sept. 1978.

[Pal87] D. L. Palumbo. Personal communication, NASA Langley Research Center, Hampton, VA, Sept. 1987.

[PB86] D. L. Palumbo and R. W. Butler, "A Performance Evaluation of the Software-Implemented Fault-Tolerance Computer," *Journal of Guidance and Control*, Vol. 9, No. 2, March - April, 1986.

[PSL80] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults", *Journal of ACM* Vol. 27, No. 2, 1980.

[PW72] W. W. Peterson and E. J. Weldon, Error-Correcting Codes, Second Edition. Cambridge, MA: MIT Press, 1972.

[RG88] T. V. Ramabadran and S. S. Gaitonde, "A Tutorial on CRC Computations," *IEEE Micro*, Vol. 8, No. 4. August, 1988.

[RH89] J. Rushby and F. von Henke, "Formal Verification of a Fault Tolerant Clock Synchronization Algorithm," NASA Contractor Report 4239, 1989.

[RKS90] P. Ramanathan, D. D. Kandlur, and K. G. Shin, "Hardware-Assisted Software Clock Synchronization for Homogenous Distributed Systems," *IEEE Trans. on Computers,* Vol. 39, No. 4, April 1990.

[SAE91] SAE/AS-2A Subcommittee RTMT Statement on Requirements for Real-Time Communication Protocols (RTCP), Issue #1, SAE ARD50007, August 2, 1991.

[Sch87] F. Schneider, "Understanding Protocols for Byzantine Clock Synchronization," Department of Computer Science, Cornell University, Technical Report 87-859, August 1987.

[Sha91] N. Shankar, "Mechanical Verification of a Schematic Byzantine Clock Synchronization Algorithm," NASA Contractor Report 4386, 1991

[ST85] T. K. Srikanth, and S. Toueg, "Optimal Clock Synchronization," *Proc. 4th ACM Symp. on he Principles of Distributed Computing*, August, 1985.

[Wat62] E. J. Watson, "Primitive Polynomials (Mod 2)," *Mathematics of Computation*, No. 16, 1962.