AFRL-IF-RS-TR-2001-53
In-House Report
May 2001

DISTRIBUTED VOTING FOR SECURITY AND FAULT TOLERANCE

Kevin A. Kwiat and Benjamin C. Hardekopf

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

AFRL-IF-RS-TR-2001-53 has been reviewed and is approved for publication.

APPROVED: [illegible]

KEVIN A. KWIAT
Program Manager

FOR THE DIRECTOR: [illegible]

WARREN H. DEBANY, JR.
Technical Advisor
Information Grid Division

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	MAY 2001	In-House May 99 - Dec 00

4. TITLE AND SUBTITLE Distributed Voting for Security and Fault Tolerance	5. FUNDING NUMBERS PE - 61102F PR - 2301 TA - 04 WU - 03
6. AUTHOR(S) Kevin A. Kwiat Benjamin C. Hardekopf	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/IFGA 525 Brooks Road Rome, NY 13441-4505	8. PERFORMING ORGANIZATION REPORT NUMBER AFRL-IF-RS-TR-2001-53
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/IFGA 525 Brooks Road Rome, NY 13441-4505	10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2001-53

11. SUPPLEMENTARY NOTES
Much of the content of this report served as a thesis for a master of science in computer science at State University of New York at Utica/Rome. Thesis defended on 28 Nov 00 by Lt. Ben Hardekopf before thesis committee consisting of Drs. Michael Pittarelli, Jorge Novillo and Kevin Kwiat. Project Engineer: Kevin Kwiat/IFGA, 315-330-1692.

12a. DISTRIBUTION AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution unlimited	

13. ABSTRACT *(Maximum 200 words)*
Faults present risk to the success of an aerospace mission, so they will continually be a concern of the fault tolerance community. In this report, we take up the issue of security in conjunction with fault tolerance. This motivated us to devise new approaches to distributed voting. Within a LAN (and some cases a WAN) we replace the almost ubiquitous 2-phase commit protocol with one that is light-weight and improves both performance and security without losing any of the traditional fault coverage. Accompanying this algorithm is one that we propose for resolving correct-but-possibly-not-identical votes within a WAN. Both of these algorithms are used to uniquely enhance the integrity of distributed information systems -- protecting them from faults and hostile attacks.

14. SUBJECT TERMS Distributed Information Systems; Distributed Voting; Fault Tolerance; Security; Integrity; Availability	15. NUMBER OF PAGES 112
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	U/L

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Dependability . . . 1

1.3 Relevance to Aerospace . . . 3

1.4 Security and Fault-Tolerance, An Overview . . . 4

1.5 Purpose of This Technical Report . . . 7

2 Background – Distributed Voting **8**

2.1 Motivation . . . 8

2.2 2-phase Commit Protocols . . . 12

2.2.1 Security . . . 14

2.2.2 Performance . . . 15

2.3 Byzantine Protocols . . . 19

2.4 Security Requirements . . . 20

2.5 Secure Voting Algorithms . . . 21

3 Timed-Buffer Distributed Voting (TB-DVA) **26**

3.1 Assumptions . . . 26

3.2 Description . . . 27

3.3 Discussion . . . 28

3.3.1 Authentication . . . 29

3.3.2 The Interface Module 30

3.4 Simulation 31

3.5 Correctness 33

3.5.1 Temporal Logic of Actions 33

3.5.2 The Specification 38

3.5.3 Proof of Partial Correctness 47

3.5.4 Proof of Termination 48

3.5.5 Lessons Learned 49

3.6 Performance 50

3.7 Intrusion Tolerance 57

4 Digital Signature Distributed Voting **61**

4.1 Assumptions 61

4.2 Description 62

4.3 Discussion 63

4.4 Performance 65

5 Conclusion **66**

A TB-DVA Proof of Partial-Correctness **74**

B TB-DVA Proof of Termination **100**

List of Figures

2.1 Comparison Chart for Centralized and Distributed Voting. 11
2.2 System Architecture 13
2.3 Pseudocode for DVA1 24
2.4 Graph of the expected time for DVA1 versus the number of processors and probability of failure 25

3.1 TB-DVA Flowchart 58
3.2 State Transition Model of Interface Module's Timer 59
3.3 Graph of the expected time for TB-DVA versus the number of processors and probability of failure 60

Chapter 1

Introduction

1.1 Motivation

Security and fault-tolerance are both extremely important issues in many areas of computer technology. Applications range from safe-guarding confidential data such as financial or medical records to monitoring safety-critical systems such as in nuclear power plants. Much work has been done in both security and fault-tolerance over the years. However, currently extra effort is being expended on investigating applications where the domains of security and fault-tolerance join together – i.e., systems that are required to have attributes of security *and* fault tolerance . The need for highly secure, highly fault-tolerant computer systems is acute in today's world. As an example, the United States President's Commission on Critical Infrastructure Protection identifies the nation's electrical, transportation, telecommunication, and financial systems as critical points, all of which heavily depend on computer technology [1].

1.2 Dependability

Dependability can be defined as the trustworthiness of a system, so that a high level of confidence can be placed in the service it performs. Dependability encompasses (but is not limited to) the concepts of *availability* (the service is ready whenever it is needed), *continuity* (the service is never prematurely terminated), and *security* (unauthorized access and tampering of the system is not permitted) [2, 3].

One aspect of dependability is fault-tolerance, which is mainly concerned with the first two concepts described above – availability and continuity. Even when faults are present a fault-tolerant system delivers useful service.

Security is another important aspect of dependability. Even if in normal operation a system is perfectly reliable, the introduction of malicious logic can have a wide range of undesired effects, from denying availability of the system (i.e., the denial of service threat) to causing the system to give users incorrect results (i.e., the integrity threat).

There are a number of systems that require both security and fault-tolerance. An example might be a military air traffic control or a target acquisition system. Such a system might have a number of independent sensors which gather data on an aircraft's IFF (Identification Friend or Foe), altitude, heading, etc., and vote amongst themselves to resolve the redundant data into a coherent picture for the user. This system must be able to function correctly in the presence of faults in its component sensors. Also essential is security against an intruder (who could range from a curious college student or a malicious terrorist organization) tampering with the data that is sent to the user in any way.

Unfortunately, developing such systems is not necessarily as simple as taking two separate designs, one highly secure and the other highly reliable, and combining them together. Unforeseen interactions between the two designs could negate either the security or the fault-tolerance of the combined version, or in the worst case render the system neither secure nor fault-tolerant. For this reason, it is important to consider both objectives together during the design stage, rather than designing them in isolation from one another.

1.3 Relevance to Aerospace

As evidenced by recent news stories, the aerospace realm is not immune from the concerns of security as well as fault-tolerance. Although a recent hacking incident at NASA [4] was deemed not to have endangered a shuttle mission, at the very least, it calls attention to the potential threats posed to the aerospace community. Networking on-board systems with those that are ground-based raises the concern of opening avenues for highly detrimental attacks. To be truly comprehensive, dependable aerospace information systems must tolerate faults that manifest themselves as a result of random phenomena or deliberate interference.

Historically, aerospace missions have been among the first to use fault tolerance. Early visionaries [5, 6] of the Apollo Program foresaw using redundancy to combat failure: by having a second spacecraft accompany a crew to the moon's surface, their return would be ensured should their primary landing vehicle be damaged. Redundancy at this level of granularity never came about because durability improvements in the eventual design of the Lunar Module reduced the risk to the crew of having only one of them on the moon.

Leaping to the present we see that special design techniques have been required for the computers used in aerospace missions. The Self-Testing And Repairing (STAR) Computer (1971), the Fault-Tolerant Multiprocessor (1975), the Fault-Tolerant Spaceborne Computer (FTSC) (1976), and the Multi-Microprocessor Flight Control System (1981) are practical computer systems that perform critical mission functions and have been specifically designed to ensure mission success [7]. Redundancy of selected computer components within these designs plays an important role in reducing the risk associated relying upon any single component to operate flawlessly. In one of the most highly visible applications of fault-tolerant computing, the Space Shuttle makes use of redundancy at the level of general-purpose computers to ensure that flight-critical operations such as ascent, reentry, and landing are performed in spite of the failure of any one computer.

Distributed computer systems, as an automatic consequence of their architecture, can be configured for concurrent operation in addition to offering resilience against hardware failure [8]. This attractive dual-property was also observed by the designers of the Space Shuttle computer in their vision of on-board systems for advanced Space Shuttles and Space Stations [9].

Distributed systems are important not only to on-board systems, but also to ground based systems. Tracking systems placed around the globe are inherently physically distributed; yet they have to be linked to command centers. Also, aerospace missions may concern multiple ground-based centers that must coordinate their separate activities through communication networks. Connectivity to this degree creates distributed systems of distributed systems. Dependency among these systems' components requires fault-tolerant design to combat the likelihood of mission failure due to system component failure.

1.4 Security and Fault-Tolerance, An Overview

The goals of security and fault-tolerance, while not exactly the same, do overlap to some extent. Both security and fault-tolerance are concerned with the integrity of the data being operated on and of the system itself. The goal of security is to protect these from malicious, intelligently directed outside attacks; while the goal of fault-tolerance is to protect them from faulty internal system components and random outside interference (e.g. from natural radiation). Similarly, both security and fault-tolerance are concerned with the availability of the service the system performs; security because it must worry about denial-of-service threats, fault-tolerance again because of faulty system components.

The main difference between the goals of security and fault-tolerance is that security is trying to guard against an intelligent, directed attack, while fault-tolerance is guarding against probabilistic phenomena. In this regard, fault-tolerance can afford to take a more optimistic viewpoint – if a sequence of events exists that circumvents the fault-tolerance measures that have been put in place, but it is wildly improbable that that specific sequence would ever occur, we can be fairly safe in ignoring it. However, if we are concerned with security as well, we cannot safely ignore such a concern. Since security is guarding against directed action rather than probabilistic events, we have to assume that if a harmful sequence of events exists someone will attempt to exploit it, and therefore we must guard against it. In this sense then, designers of secure systems must be much more pessimistic than designers of fault-tolerant systems. This is one source of difficulty in combining the two – the assumptions that the designers work under are different, and therefore the combined design is working under different assumptions than either of the designers planned for.

Another source of strain in a secure and reliable design is the diametrically opposed strategies of security and fault-tolerance. In security, the more centralized and compact the design the easier it is to guard. If a design is spread out, and perhaps administered by several people over a wide area, then incompatible configurations and uneven updates of the system can easily introduce vulnerabilities for an attacker. On the other hand, a main strategy used in fault-tolerance is *redundancy*. If a component is critical to the system and has some probability of becoming faulty (as all components do), it makes sense to have several redundant versions of the component working independently. If one fails, there are backups to take over the work and the system can continue operation. In fault-tolerant terms, the more spread-out and distributed the system is, the more robust it is and the better the fault-tolerance of the system. If a system is spread out far enough, and has enough redundancy, even localized disasters such as hurricanes or earthquakes cannot completely shut the system down. Now, security *can* make use of some types of redundancy, such as the redundant bits in a checksum used to detect altered data. However, in general redundancy can be detrimental to a secure system because it adds to the complexity, making the system more difficult to secure, as well as more difficult to verify as secure. A simple example of this type of dichotomy is replicated databases distributed over a wide geographic area. This type of system is becoming common [10, 11]. The purpose is to increase the longevity of the data and to make the system more reliable. However, the difficulty is that having multiple copies of the data distributed across multiple sites makes it much easier for an attacker to find the data, making the system less secure. It is therefore clear that when critical data or systems are being made reliable, security needs to factored in to the design from the outset, not as an afterthought.

It is similarly true that when designing a secure system it is important to consider fault-tolerance, even if reliability is not a primary goal. As an easy example, take a security program that runs in the background, monitoring the system and raising alerts if necessary (such as if an intruder is detected). In such a system, no news is good news – if there is no alert, the system is assumed to be secure. But if the monitoring program were to quietly fail, the system could be compromised without any warning being given. It is obviously necessary to add some form of fault-tolerance and/or fault-detection in such a system so that if the program did fail, it could either recover or notify the system of its failure. Another more complex example is cryptographic security. Research has shown that many cryptographic algorithms are vulnerable in the presence of random hardware faults [12]. One example deals with breaking cryptosystems on tamper-resistant devices in the presence of transient faults [13]. It has been shown that by inducing transient faults in devices such as smartcards (e.g. by exposing them to high levels of radiation), knowledge about the private keys within the devices can be deduced. In order to guard against these types of attacks, it is necessary to enable these devices to detect and correct any such faults (or at least detect them and abort the operation).

Computer security does not necessarily involve the question of how to tolerate faults because fault tolerance can be grafted onto the system at some other level. For example, fault tolerance is designed into the security kernel of operating systems so that unavoidable faults do not result in security policy compromise [14]. It then becomes conceivable to build trusted services upon this underlying layer that will remain secure even in the presence of faults. It should be noted that evidence [15] strongly suggests the need for security measures at the lower levels of software abstraction to form the foundation for secure computing at the higher levels. This motivates us to consider dependability at the processor level without assuming an over-arching computing environment to provide security. We employ redundant processors to achieve fault tolerance, but we do not treat it and security as two entirely separate issues; instead, we seek to provide both of them at the same time using the same resources.

1.5 Purpose of This Technical Report

The fields of security and fault-tolerance are very broad, and there are many types of systems that have the requirement of integrating security and fault-tolerance. This technical report focuses in on one specific problem in this area – secure distributed voting algorithms. Distributed voting is a well-known fault-tolerance technique that has been used for many years. More recently there have been various schemes proposed with the purpose of making distributed voting more secure. This technical report will analyze these protocols and indicate some of their weaknesses. One common weakness is that the schemes all utilize *exact* voting, and do not consider the requirements for *inexact* voting [7] (these terms will be explained in more detail later). The technical report will then describe in detail a protocol that combines the requirements of security, fault-tolerance, and performance while remaining general enough to handle both exact and inexact voting.

The second chapter of this technical report describes the history of distributed voting, and analyzes some of the current protocols that are used today. Problems with these protocols in regards to security and/or performance are pointed out, and a general criterion necessary for secure distributed voting is developed. Several algorithms which meet these requirements are presented and their weaknesses are analyzed. The third chapter introduces one conception of distributed voting which overcomes these weaknesses. It is analyzed in regards to both security and performance, and its limitations are pointed out. The fourth chapter introduces a second scheme for secure distributed voting that overcomes some of the limitations of the first. This protocol is also analyzed for security and performance, and its own limitations are pointed out. The fifth chapter concludes the technical report by summarizing the main features of the technical report and discussing future work to be accomplished in this area.

Chapter 2

Background – Distributed Voting

2.1 Motivation

Replication and majority voting are the conventional methods for achieving fault tolerance in distributed systems. The system consists of a set of redundant processors all working on the same task in parallel, then voting on the individual processors' results to pick one as the correct answer. This technique has been in use for quite some time – it was first proposed in the context of electronic computing by John von Neumann circa 1945. The early incarnations used *centralized* voting: each of the processors would send their result to a central tallyer, which would analyze these votes and determine a majority. There are several problems with this technique. One is that the central tallyer represents a single point of failure for the system; if it fails, the entire system fails. Another problem is that the system is not very configurable – once it is set up to do centralized voting, it is difficult to utilize the system for other tasks. For these reasons, another technique was developed called *distributed* voting. In these systems, there is no central tallyer. The processors communicate among themselves in order to determine the majority vote. This means that there is no longer one single point of failure for the system, because if one of the processors drops out the others can operate without it. Another attractive feature of using distributed systems is dual mode operation: when a task is highly critical, such as in a space vehicle's launch phase, the processors operate in fault-tolerant mode; when fault tolerance is not required, such as in the vehicle's cruise phase, the processors cease being redundant and can execute different subtasks in parallel. Redundant processor operation would be reinvoked during the vehicles descent phase. Such systems have been in use for some time; for example it has been used by the Space Shuttle's Primary Computer System since the 1970s [9].

Distributed voting in the past has largely been restricted to tightly bound multi-processors and small local-area networks. However, given the current trend towards distributing resources over a very large area (e.g., via the Internet), it is worth investigating the benefits of this technique in such wide area networks. Distributing data and computation over a wide area network is becoming a standard practice. Critical databases have already been replicated and dispersed to various geographical sites to increase their longevity [10, 11]. Redundant computations are also distributed in order to combat localized network failures and attacks, increasing both security and fault-tolerance. As a consequence, redundant computations on replicated data at remote locations must somehow coordinate their results in order to present a majority result to the user. One example of this requirement is gathering data from distributed sensors with overlapping areas of coverage. Determining a majority result from these sensors produces the lowest probability of error for the widest range of observation probabilities [16]. Data need not be identical. It may even be made different deliberately: data diversity [17] is a software fault tolerance strategy where a related set of points in a program's data space are obtained, executed upon using the same software, and then a decision algorithm (i.e., voter) determines the resulting output.

Centralized voting (having a distinguished coordinator which collects the votes from all voters and then determines the majority) is a simple solution to the problem of resolving the output of redundant voters in a wide area network. However, as networking becomes more ubiquitous the advantages of distributed (i.e., decentralized) voting become clear.

Use of a centralized coordinator, which may be quite distant from the participating voters, could consume much more bandwidth than distributed voting, in which the voters need only communicate among themselves. Transmitting results from the voters to the coordinator may involve many network hops and accrue more overall delay than having the voters communicate among themselves. Designating a node that is close to the redundant voters to act as a 'delegate' coordinator may not be possible because it entails placing complete trust in that delegate and assuring that a dependable communications link exists between it and the result's final destination.

Another problem with centralized voting is the possibility of link failure which may partition the network, rendering communication between either the voters and the coordinator or the coordinator and the user impossible. In a distributed scheme, as long as a majority of the voters can communicate a final result can be calculated; and as long as the user can communicate with any of the participating voters it can obtain that result.

The fact that the coordinator is receiving messages from each and every voter makes network congestion in its vicinity likely, especially if it is responsible for many redundant tasks carried out at the same time (and hence is receiving messages from many voters at once). Decentralized voting distributes the message traffic attendant on each task and thus tends to confine it to the participating voters.

Decentralized voting also allows the necessary computation for determining the majority to be distributed and calculated in parallel among the voters. Insufficient computing capacity of the coordinator can restrict the usefulness of centralized voting. Research has been done in the area of software agents that perform centralized voting [18], but no consideration has been given to agents that may not be able to compute the majority, but only apply it. Such "bounded rational agents" have limited decision capabilities due to restrictions placed upon them regarding the computational resources they can consume [19]. This may be a problem when the task of comparing two votes involves complex calculations, such as when the votes may be somewhat different, yet still be equal. Determining the majority of correct-yet-different results calls for "inexact voting" that, being potentially far more complex than a mere bit-wise comparison of results [20], can readily exceed an agent's limited decision making power. Requiring the coordinator to correctly decide among results that can differ but still be correct is understandable when one considers, e.g., the tolerances of sensor

	Advantages	Disadvantages
Centralized	simple to implement	single point-of-failure; rigid architecture
Distributed	no single point-of-failure; flexible architecture	complex to implement; reliance on committing voter

Figure 2.1: Comparison Chart for Centralized and Distributed Voting.

readings. Being unable to compute a majority, an agent that obtains it from elsewhere could nonetheless use it to, for example, manipulate an actuator through a microcontroller.

A final consideration when using centralized voting is the possibility of an adversary observing the network. Such an adversary could, using network traffic analysis, easily determine the importance of the coordinator from the sheer number of messages it was receiving. Being distinguished in this manner makes the coordinator a tempting target for attack. Once the coordinator has been compromised, the attacker has complete control over the results seen by the user. In decentralized voting, no voter is more important than any other. Done correctly, an attacker would have to compromise a majority of the voters before being able to control the results seen by the user, greatly increasing the cost of any successful attack.

Figure 2.1 contrasts the advantages and disadvantages of centralized and distributed voting. Despite some disadvantages, distributed voting is an important and much-used fault-tolerance technique. A contribution of the techniques described in this technical report is to reduce those disadvantages of distributed voting evident in the third quadrant of Figure 2.1.

There are two main types of algorithms that can be used for distributed voting: 2-phase commit protocols and Byzantine protocols. They will both be described next. Then several protocols designed for secure voting will be presented and analyzed.

2.2 2-phase Commit Protocols

Software voting has had several embodiments ([21, 22, 23, 24]) in the development of fault-tolerant computing. More recently, distributed voting has been used for fault diagnosis in linear processor arrays [25] where, in the absence of a centralized voter, the array elements share error flags stemming from output comparisons performed between connected elements. Despite the variety of applications, current distributed voting schemes continue to subscribe to a common protocol: once the voting is complete and a majority result has been determined, one processor is chosen to commit the majority result to the user. Thus they are all examples of a *2-phase commit protocol* [26]: voting is undertaken in the first phase, in which all participants share results, followed by a second phase, in which the committal is executed. A distinguished process coordinates the committal. This type of protocol is prominently used for distributed voting, as described in [27]. The algorithm presented in [27] is representative of this class of protocols, and will be described and analyzed here to show the limitations all members of this class possess. The algorithm has no formal name, and will be referred to as DVA1 throughout the rest of this paper.

The particular distributed environment assumed by this algorithm is a bus architecture local-area network with general purpose workstations attached. Also attached is an interface module (a simple buffer augmented with a timer) that is local to the user and is used to hold the final result for the user to access (see Figure 2.2). Because the interface module is local to the user, it is assumed to exist outside the fault containment boundary. Being local to the user, the interface module need not be concerned with errors in communication; instead the availability of routines at the user's site that handle message acknowledgement or request for retransmission can be safely assumed because lower-level protocols must deal with communication channel noise regardless of how failures of the processors are tolerated. Coupled with the fact that it is far less complex than a processor, we can safely assume it is fault-free. It is further assumed that there are mechanisms in place to ensure that the processors follow the given protocol faithfully (i.e., processors may not fail arbitrarily, but only by halting or giving incorrect votes); any faulty processors will not disable commu-

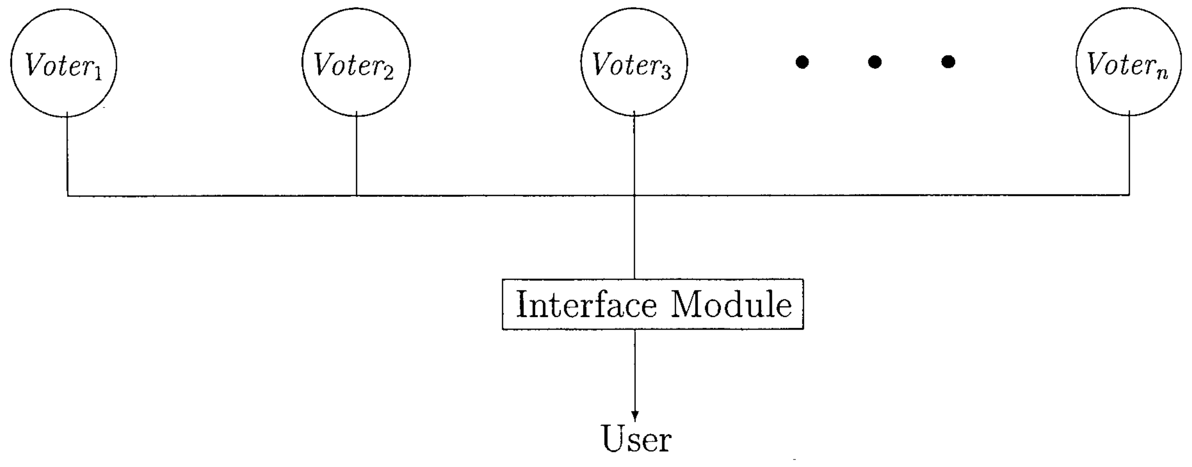


Figure 2.2: System Architecture

nications among the other processors (e.g., by appropriating the broadcast medium); and processors are able to detect internal faults through a self-diagnostic routine initiated upon demand. The requirement for processors being unable to disable communication can be met by using a network designed to provide fair access to all attached hosts, such as token-ring or token-bus networks [28].

The pseudocode for DVA1 is given in Figure 2.3. The algorithm is fairly simple. Once a task has been given, each processor calculates its result and broadcasts it to all the other processors. Once all the votes have been broadcast, each processor analyzes the votes in order to determine the majority. Whichever vote is in the majority is taken to be the final result, and one of the processors who is in the majority is arbitrarily selected to commit that result to the interface module. Any processor whose vote was not in the majority is faulty, and must initiate a recovery routine. If no majority vote exists, then the processors must run self-diagnostics. Once it has been determined who is faulty and who is not, an arbitrary non-faulty processor is chosen as the coordinator, which commits its result to the interface module.

2.2.1 Security

The weakness of this approach lies in the second phase – the committal. If the committing voter fails just prior to, or during, the committal, an incorrect result can be committed. However, when considering only fault tolerance, the 2-phase commit protocol is a very legitimate and hence widely adopted approach for distributed voting. Unless one assumes a sufficiently high probability of processor failure (much higher than usual), the likelihood of a fault occurring in the coordinator during committal is negligible. When security is also an issue, however, it becomes apparent that entrusting committal to the coordinator can have disastrous results. Even if the normal failure rate of a system is very low, a malicious attacker can cause errors to happen in exactly the wrong place and time – i.e., at the coordinator during the committal. An attacker who takes control of the coordinator can cause it to commit an incorrect result regardless of the answer the processors came up with together. Security measures can be put in place to protect each processor, but the fact remains that this algorithm provides a single critical point (the coordinator) that must be protected, or the security of the whole system is compromised. And since the role of coordinator is assigned arbitrarily among the voters, all voters must be protected as if they were the coordinator, increasing the total cost of the system.

2.2.2 Performance

While the main difficulty we are concerned with is the lack of security in this type of protocol, another important consideration is performance. Any alternative algorithm we may come up with to improve security should at the very least have equivalent performance, and ideally have even better performance.

An unpleasant impact of distributed voting for fault-tolerance is that, in addition to the communication requirements it entails, the time to perform the requisite result comparisons lowers the throughput of the fault-tolerant task. Increasing fault tolerance by offloading the voting onto all of the processors, thus avoiding the use of a centralized tallyer, does not come without a cost. Distributing the majority voting has been shown to adversely impact system throughput in even a small multiprocessor [29]. In the case of the SIFT (Software Implemented Fault Tolerance) Program approach (circa 1975), as much as 60% of the processor's raw throughput was consumed by the software-implementation of the voting [30]. It was estimated that the execution of these software-intensive functions in the MAFT multiprocessor was *two orders of magnitude* too slow for a usable system [31]. Therefore, when using software-based majority voting it is imperative for the fault tolerance protocol be chosen so that it yields the best performance.

Assigning the responsibility of voting to the processors necessitates that they invoke special software routines for this purpose. The overhead these routines accrue is not just from having to gather the votes from the other processors. Resolving the votes to produce a majority can also substantially add to the overhead. *Inexact voting* is the term applied to deciding the majority among processors whose outputs may not exactly agree but could nevertheless all be correct due to tolerances in the output specification. Inexact voting is more complex than a mere bit-wise comparison of the processor outputs. An algorithm for inexact voting is so dependent on the type of task that it is impossible to formulate a general algorithm for determining if the processors' outputs are different from one another within an allowable range [20]. This only exacerbates the performance problems of such systems.

In DVA1, the total time taken to send to final answer to the user is composed of three parts - the time for each processor to compute its result, the time for all the votes to be broadcast, and the time for the processors to compare the votes. This can be expressed by the equation

$$T_{total}(n) = T_{compute}(n) + T_{vote}(n) + T_{analyze}(n)$$

where n is the number of processors. Since the processors are operating in parallel, the time for all the processors to compute their results is independent of the number of processors, therefore $T_{compute}(n) = \Theta(1)$. Since each processor must broadcast its vote, the number of broadcasts rises linearly with the number of processors, hence $T_{vote}(n) = \Theta(n)$.

In the worst case, at most one processor will be correct. When the processors are analyzing the votes, they must compare the first vote with all the other votes, the second vote with all but the first vote, and so on. This means that $T_{analyze}(n) = O(n^2)$. It's also important to recall that these operations might be much more complex than a simple bit-wise comparison, making this stage a performance bottleneck.

In the best case, all the votes would be correct. When the processors are analyzing them, they would compare the first vote to all the rest and be done. Therefore the time to analyze them would rise linearly with the number of processors, and $T_{analyze}(n) = \Omega(n)$.

It's clear that the total time to send the final result to the user depends most heavily on the analysis time of the algorithm (given, of course, sufficiently large n to overwhelm whatever time the computation step entails), so

$$\begin{aligned} T_{total}(n) &= O(n^2) \\ T_{total}(n) &= \Omega(n) \end{aligned}$$

Given an assumption of a relatively low failure rate, the lower bound is a good estimate of the algorithm's average performance.

To quantify this analysis, we can calculate the expected time for the algorithm as a function of the number of processors and the probability of failure for the processors. Each processor will have an array of votes representing the values calculated by all the processors, once the values have been broadcast. Each processor will then compare these votes to each other to determine the majority. For the purposes of this analysis we assume that no two faulty votes will agree. Although this assumption implies that two agreeing voters constitute a majority, it is made to give us the worst-case performance because it causes the maximum number of comparisons. We further assume that the comparison of the votes is the most time-consuming portion of the algorithm, overwhelming the other steps. The rationale for this assumption is the potential complexity of the comparisons, as described earlier.

Given these assumptions, it's clear that the time for the algorithm depends on how many votes agree (are correct) and how many disagree (are faulty). When comparing the votes, the first vote is compared with all the others. Any votes that match this first vote can be set aside and not looked at again. Then the next vote which did not match the first vote is taken and compared with all the remaining votes, and so on until each vote has either been matched with another vote or been compared with all other votes. The more correct votes there are, the more votes will be matched and taken out of consideration, and the fewer comparisons there will be.

The actual number of comparisons given a particular number of correct and incorrect votes also depends on the relative positions of the elements in the array of votes. If, for instance, the very first vote is correct, then the first step (of comparing the first vote with all other votes) will remove all correct votes from consideration, leaving only the incorrect votes to compare themselves. However, if the first vote is incorrect, it must be compared with all the other votes, matching none, and hence leaving all the correct votes to be compared with the next vote. In fact, the closer the first correct vote in the array is to the head of the array, the fewer comparisons must be made.

Therefore, in order to calculate the expected completion time for the algorithm, the expected number of comparisons as a function of the number of processors and the number of correct votes must be calculated first. Let n be the number of processors and i the number of correct votes.

Since the probability of failure of each processor is equal, the probability of the first correct vote being in the jth position is the number of possible ways that i correct votes can be arranged so that the first is in element j divided by the total number of ways that i correct votes can be arranged in the array. This turns out to be

$$\frac{\binom{n-1-j}{i-1}}{\binom{n}{i}}$$

Each incorrect vote before the first correct vote in the array must be compared with every other vote - the first vote must be compared against $n-1$ other votes, the second against $n-2$, and so on. Once the first correct vote is the one being compared against the others, it will match all the other correct votes and remove them from consideration. At that point, all the remaining (incorrect) votes will be compared against each other. Therefore, the number of comparisons given that the first correct vote is in element j is given by the formula

$$(j+1)n - \frac{(j+1)(j+2)}{2} + \frac{(n-i-j)(n-i-j-1)}{2}$$

Therefore, the expected number of comparisons $E_c(n,i)$ is given by

$$\begin{aligned} E_c(n,i) &= \textstyle\sum_{j=0}^{n-i} (P(\text{first correct vote is in the } jth \text{ element}) \times (\# \text{ of comparisons})) \\ &= \textstyle\sum_{j=0}^{n-i} \left(\frac{\binom{n-1-j}{i-1}}{\binom{n}{i}} \left((j+1)n - \frac{(j+1)(j+2)}{2} + \frac{(n-i-j)(n-i-j-1)}{2} \right) \right) \end{aligned}$$

At this point, finding the expected time for the algorithm simply entails finding the probability that there are i correct votes out of n processors given a probability of failure ϕ, and multiplying by the expected number of comparisons given that particular i and n. $E(t)$ is given by

$$E(t) = \phi^n \left(\frac{n(n-1)}{2} \right) + \textstyle\sum_{i=1}^{n} \left(\binom{n}{i} \phi^{n-i} (1-\phi)^i E_c(n,i) \right)$$

If we then graph this function with $2 \leq n \leq 11$ and $.01 \leq \phi \leq 0.5$, we obtain the graph in Figure 2.4. As expected from the asymptotic analysis, the time is linear with the number of processors at low failure probabilities, turning quadratic at higher probabilities.

2.3 Byzantine Protocols

A second class of protocols that could be used for distributed voting are the so-called "Byzantine" protocols. The SIFT (Software Implemented Fault Tolerance) Program [32] was the first attempt at supporting fine-grained flexibility in fault-tolerant operation that entailed the need for decentralized voting. This program also served as the seedbed for solutions to important problems in fault-tolerant distributed systems such as the Lamport's "Byzantine Generals Problem" [33]. Byzantine algorithms were developed as a way to tolerate voters that can fail in a totally arbitrary manner, such as sending conflicting results to two or more different sets of participating voters. The goal of these algorithms is to achieve *Byzantine agreement* – all participants should have a globally consistent view of the system. Byzantine faults are the most malicious kind of processor faults that can be considered, and therefore are the most difficult to tolerate.

While these algorithms are much more secure than the 2-phase commit algorithms, they are also much more complex. The theoretical requirements necessary to guarantee correct system behavior in this situation can be summarized as follows: in order to tolerate f Byzantine faults, it is necessary to have $3f + 1$ independent participating voters in the Byzantine fault-tolerant scheme, where the voters are connected by $2f + 1$ disjoint communication paths, and go through $f + 1$ rounds of information exchange to arrive at exact consensus [33]. In order to tolerate just one fault, the system would consist of 4 processors, each having 3 independent communication paths, and would go through 2 rounds of communication. To tolerant as few as 3 faults, the system would require 10 processors, each having 7 independent communication paths, going through 4 rounds of communication. Some functionalities have different redundancy requirements depending on the service provided. Providing a majority result to a user requires only corresponding results from at least $\lceil \frac{2f+1}{2} \rceil$ of $2f + 1$ processors. However, update to the processors requires up to $3f + 1$ processors in order to ensure consistency. Barborak and Malek [34] offer a thorough survey on the fault-free segement of a processor population achieving agreement on a system status in spite of the possible inadvertent or even malicious spread of disinformation by the faulty segment of that population.

Because of the great complexity inherent in a Byzantine fault-tolerant system, and hence the large cost of building one, not many such systems have been implemented commercially. This type of protocol is therefore too expensive for being used in a commercially viable (i.e., economically feasible) distributed voting system.

2.4 Security Requirements

The two types of protocols that are commonly considered for distributed voting have been reviewed. It was found that the 2-phase commit protocols, while suitable for purely fault-tolerant applications, exhibited very poor security. The Byzantine protocols, on the other hand, provide very good security – but are much too complex and expensive to be practical.

The reason that the 2-phase commit protocols are insecure is that the coordinator is unsupervised during a committal; it can commit anything it wants, and the interface module has no idea if the result it gets is consistent with the results of the other processors. These protocols are not strict enough to provide adequate security.

The reason that the Byzantine fault-tolerant systems are so complex is because they attempt to achieve a globally consistent view of the system. This requirement is stricter than we require for a distributed voting scheme – we merely wish to ensure that the result passed on the user is consistent with the results of the non-faulty, non-malicious voters. We do not need or desire a globally consistent view.

Therefore, we require a middle ground between these two extremes in order to arrive at a secure, practical distributed voting system. The protocols described next were attempts to arrive at a suitable compromise.

2.5 Secure Voting Algorithms

There have been several protocols proposed that attempt to overcome the problems described earlier. For example, the algorithm presented in [35] works as follows (in a very simplified presentation):

1. A client sends a request to one of the voters.
2. The voter multi-casts the request to the other voters.
3. The voters execute the request and send a reply to the client.
4. The client waits for $f + 1$ replies from different voters with the same result, where f is the number of faults to be tolerated; this is the final result.

While this strategy obviously is not subject to the same problem as the 2-phase commit protocol, since in essence *all* the voters commit a result, it does require substantial computation on the part of the client, which must collect and compare all the replies until $f + 1$ have been collected that carry the same result. As a result, this system does not scale very well.

Another protocol that attempts to alleviate this problem is described in [36]. It makes use of a *(k,n)-threshold signature scheme.* Informally, this describes a scheme wherein a public key is generated, along with n shares of the corresponding private key, each of which can be used to produce a partial result on a signed message m. Any k of these partial results can then be used to reconstruct the whole of m. In this particular protocol, n is the number of voters, and k is set as one more than the number of tolerated faults. Each voter signs its result with its particular share of the private key and broadcasts it to the other voters. The voters then sort through the broadcast messages for k partial results which agree with its own result and can be combined into the whole message m, where m would be the signed final result. The voter then sends m to the client, which accepts the first such valid m sent. Again, this protocol is not subject to the error inherent in the 2-phase commit protocol (since the digital signature assures the client that $f+1$ voters agreed with the result, otherwise the result could not have been signed), and it is also not computationally expensive for the client. It achieves this by shifting to the voters the large computational burden involved in sorting through the the votes looking for matches. However, the large number of messages that must be sent between the voters, plus the effort involved in digitally signing and validating these messages, adversely impact the performance of the system.

Other protocols have also been proposed, each with its own advantages and disadvantages [37, 38, 39, 40]. However, all of the above schemes for securing the distributed voting process make the common assumption which underlies the idea of state-machine replication – two different voters, starting in the same state and following the same instructions, will inevitably arrive at the same result. While there are many cases when this assumption holds, there are also times when it does not. This is true in the case of so-called *inexact voting.*

In inexact voting, two results do not have to be bit-wise identical in order to be considered equal, as long as they fall within some pre-defined range of tolerance. This situation often arises when data is gathered from sensors interacting with the real world – it is extremely unlikely that two different sensors will collect exactly the same data, even if they are arbitrarily close to one another and sampling the same phenomena; therefore some analysis needs to be done to determine if the sensors' data is effectively equal, even if not identical.

In such situations the schemes described above will encounter problems, because of the common assumption they all make that the replicated voters' data will be identical. For example, the second algorithm described above, which uses a (k,n)-threshold scheme, cannot be used for inexact voting – in order for the partial results to be combined together into a whole result for the client, the partial results must be identical.

While some of the algorithms could be modified to handle inexact voting, the performance cost incurred through multiple inexact comparisons would be prohibitive. For example, the first algorithm described in this section, in which all voters send their results to the client, would force the client to make multiple inexact comparisons in order to determine the majority. Since inexact comparisons can be very complex operations, this places an unacceptable burden on the client.

One possible solution, which under certain assumptions provides a mechanism that is at once fault-tolerant, secure, and high-performance is described and analyzed in chapter 3. Another potential solution which operates under looser assumptions than the first, and hence can be more generally applied, is described in chapter 4.

```
Phase 1:
    /* Each $P_i$ in a node does the following, $(1 \leq i \leq n)$ */
      1. vote = function( "parameters") /* computation depends on the application */
      2. broadcast("node", vote, P_name) /* broadcast "vote" to all processors (P_name) in "node"*/

Phase 2:
    /* Each $P_i$ does the following, $(1 \leq i \leq n)$ */
      vote[$j$] = recv_msg($P_j$, vote, P_name) $1 \leq j \leq n, j \neq i$ /* rec. vote from all processors */
      If (vote[$i$] = vote[$j$]) $\forall_{i,j}$, $1 \leq i,j \leq n$
        begin
          result = vote[$i$]; /* result contains vote to be committed * /
          vote_commit("all"); /* vote commit with param "all" */
        end;
      else if (vote[$l$] = vote[$j$]) for at least k votes s.t. $k \geq \lceil \frac{n}{2} \rceil$
        begin
          result = vote of majority; /* result contains value of the majority *
          if (($P_i$ =$P_c$) .AND. (vote[$i$] $\neq$ result)) /* if current coordinator not in majority */
            select_coord ($P_k \in$ majority); /* select new coordinator */
          vote_commit ("majority"); /* commit the majority value */
          if ($P_i$ = $P_c$)
            local_recovery ( for all $P_j$ not in majority); /* start recovery of processors not in majority
*/
        end;            else /* no majority */
        begin
          $P_i$(status) = local_diagnostic ($P_i$), $1 \leq i \leq n$ /* start local diagnostics */
          if ($P_i$(status) = "okay")
            begin
              select_coord ("node"); /* select new coordinator from "okay" processors */
              if ($P_i$ = $P_c$) /* new coordinator does the following */
                vote_commit(new_majority); /* commit new majority */
            end;
          else if ($P_i$(status) $\neq$ "okay") $\forall_i$ , $1 \leq i \leq n$ /* all processors have failed */
            print("complete node failure, external recovery required");
        end.
```

Figure 2.3: Pseudocode for DVA1

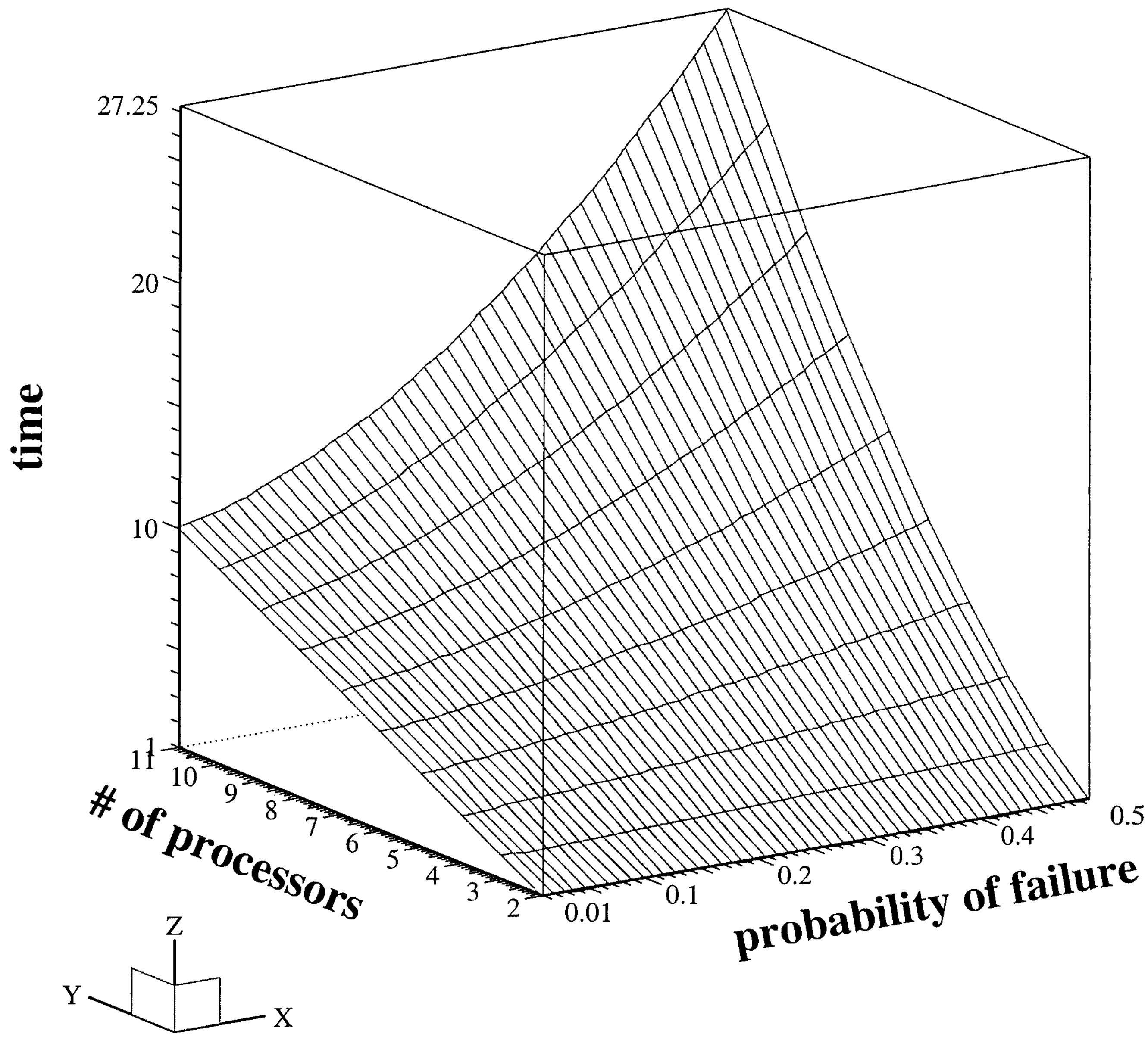


Figure 2.4: Graph of the expected time for DVA1 versus the number of processors and probability of failure

Chapter 3

Timed-Buffer Distributed Voting (TB-DVA)

3.1 Assumptions

The proposed algorithm has two sets of participants. One is the set of voters, which can be arbitrarily large but must have at least three elements. These voters are completely independent; the only exchange of information that takes place between them is the communication of the voters' individual results. The other set contains the user and an interface module. The interface module buffers the user from the voters (see Figure 2.2). The interface module consists, in its abstract form, of a simple memory buffer and timer. A task is sent from the user, through the interface module, to the voters. At the termination of the algorithm, the interface module passes the final result back to the user. This system is physically identical to the system used in DVA1, except for the timer embedded in the interface module.

The environment for the algorithm is a network with an atomic broadcast capability and bounded message delay (e.g., a local area network). It is assumed that a fair-use policy is enforced, so that no host can indefinitely appropriate the broadcast medium [28]. It is also assumed that no voter will commit an answer until all voters are ready – this can be easily enforced by setting an application dependent threshold beyond which all functional voters should have their results ready; any commits attempted before this threshold is reached are considered automatically invalid. Each voter can commit only once – this is enforced at the interface module, which ignores commits from a voter which has previously committed. The most important assumption made is that a majority of the participating voters are fault-free and follow the protocol faithfully (these are called *trustworthy* voters). Increasing the effort required for an attacker to breach security can enforce this assumption. To successfully overtake a majority of voters, each having diverse intrusion detection packages and user interfaces, requires attackers to possess greater experience and ability, and costs them more in terms of both financial cost and elapsed time [41]. No assumptions are made about the remaining voters (the *untrustworthy* voters) – they can refuse to participate, send arbitrary messages, commit incorrect results, etc.; they are not bound in any way.

3.2 Description

Each of the (trustworthy) voters will follow the steps below:

1. If no other voter has committed an answer to the interface module yet, the voter does so with its own vote; it then skips the remaining steps.

2. In the case that another voter has committed, the voter compares the committed value from the other voter with its own vote.

3. If the results agree, the voter does nothing; otherwise it broadcasts its dissenting vote to all the other voters.

4. Once all voters have had a chance to compare their votes with the committed value (this interval would be determined by a timer), the voter analyzes all the dissenting votes to determine if a majority dissenting vote exists.

5. If no majority exists, then the voter does nothing.

6. If a new majority exists (or if another, perhaps faulty, voter commits a new result), then the voter returns to step 1.

The interface module will follow these steps:

1. Once a commit is received, the result is stored in the buffer and the timer is started. The timer is set to allow time for all the voters to check the committed value, dissent, and recommit if necessary.

2. If a new commit is received before the timer runs out, the new result is written over the old in the buffer, and the timer is restarted.

3. If no commit occurs before the timer runs out, then the interface module sends the result in its buffer to the user, and the algorithm is terminated.

3.3 Discussion

TB-DVA corrects errors at the destination of the majority result. This is somewhat akin to using information theory to correct errors induced in a channel that connects a source to a recipient [42]. With the information theoretic approach, redundancy is encoded into the message on the source's side of the channel, so that error-free transmission is accomplished at the expense of time delays for decoding the message after it reaches the recipient's side of the channel. TB-DVA takes advantage of redundancy in the message *sources* and uses *time* to correct a source-induced error that arrives at the recipient's side of the channel.

TB-DVA reverses the 2-phase commit protocol by initiating a commit to a timed buffer and then allowing for a period of dissension in a voting phase. Although conceptually simple, this change forces an attacker to overcome a majority of the voters in order to compromise the system. This has important security implications because it greatly increases the cost of a successful attack.

3.3.1 Authentication

For the correct execution of the voting algorithm it is necessary that the commits sent to the interface module from the various voters be authenticated, as well as the messages between the voters themselves. Any known sophisticated authentication techniques can be used to enforce secure communication, but it should be done without increasing the complexity of the interface module. For authentication between the voters, the particular method used is not as important since there is plenty of computational power available. The interface module, however, should be kept as simple as possible. For illustrative purposes, we describe a simple authentication technique for the interface module that does not employ standard cryptographic methods such as public key encryption. The technique described here is called SKEY authentication [43], which is simple to implement but is capable of strong authentication with minimal communication between the voters and the interface module. This approach allows the implementation of our secure and fault tolerant voting scheme on existing platforms without any modifications to the underlying protocols.

The SKEY authentication is based on a one-way function. The voter and the host on which the interface module is built first agree on a common random number R prior to the start of the voting algorithm. A set of numbers $x_1, x_2, ..., x_n$ is generated at a given voter as well as the host by applying the one-way function f on R as $x_1 = f(R), x_2 = f(f(R))$, and so on. The host also calculates and stores x_{n+1}. The voter sends its commit by appending x_n to its vote. The host will calculate $f(x_n)$ and compare it with x_{n+1}. If these numbers match, the communication is treated as authentic. The voter will delete x_n and use x_{n-1} the next time it has to commit to the interface module.

Since the SKEY method requires only an occasional exchange of a random number between the voters and the host computer in which the timed-buffer resides, a reasonable level of security can be maintained on the exchange of votes.

3.3.2 The Interface Module

The function of the interface module is to record a commit from a voter, set up a timer, wait until the timeout expires, and deliver the result to the user. It is possible that the timer may be reset several times before passing the final result to the user. In addition, the interface module should have the capability to authenticate voters, so that it can track the voters to ensure that each can commit only once in a given voting cycle. In order to reduce the likelihood of attacks on the interface, it should be isolated from the rest of the voter complex and be built to have minimal interaction with the outside world.

Depending upon the level of voting, the design of the interface module may vary. Voting may proceed at either hardware or software levels. It essentially depends on the volume of data, complexity of computation, and the approximation and context dependency of the voting algorithms. If low-level, high-frequency voting is to be done, a hardware implementation might be preferred; if high-level voting with low-frequency is desired, a software implementation of the interface module may be suitable. This is because the voting is generally much more complex at higher levels of abstraction. A software implementation is fairly simple, although the real-time nature of the protocol must be taken into account. We assume low-level, high-frequency voting in hardware and discuss a hardware architecture for the interface module below.

Since only one copy of the vote needs to be buffered before giving it to the user, the amount of memory required is small. The actual size depends on the data that is voted upon. The tracking of voters can be implemented using a flag register. One bit flag per voter is sufficient. The flag will be set as soon as a commit is received from a voter and will be reset after the expiry of the timer. If multiple commits are received from the same voter during the flag set state, they will be ignored.

A small amount of additional memory must be built into the module to support the SKEY authentication of communication between the voters and the interface module as described before. This memory is needed to store a sequence of n numbers for each of the voters as required by the SKEY method of authentication. Control logic must be designed into the interface module to step down the sequence each time a commit is received from a voter. Re-initialization of the sequence for a specific voter is necessary when the sequence reduces to zero, over time. This can be done by requesting the host computer to receive a new random number from the voter and computing a new sequence. Another capability that needs to be built into the interface module is the synchronization of result delivery with the expiry of the timer.

Though the interface module may be viewed as a single point of failure, it is far less vulnerable to failure than a voter would be due to the decreased level of complexity compared to the voter/processor module. The module has no requirement to run any algorithm (code). It is isolated from the voter complex and is designed to have minimal hardware and minimal interaction with the outside world. Thus, it is less vulnerable to attacks as well.

3.4 Simulation

Departure from the 2-phase commit protocol with the adoption of TB-DVA could be reasonably proposed only after affirming that the degree of difficulty in implementing them would not be significantly different. For the purpose of determining this, we implemented DVA1 and TB-DVA in Java. We also used Java to simulate a system architecture consisting of five processors on a LAN. For simulation of TB-DVA, this architecture was augmented with an interface module component as shown in Figure 2.2.

In constructing simulation models of DVA1 and TB-DVA, it became apparent that they both heavily depend upon the notion of a *timeout.* Because a processor may fail to produce any results, timeouts are employed to prevent indefinite postponement of the voting process. Just how long of a waiting period should be before a timeout occurs depends upon estimates of system properties and circumstances that are sources of delay. Unless a system can accept unbounded message delays, a fault-tolerant distributed system must be designed with an upper bound on the timeout period [44]. For this reason, DVA1 assumes that the voter receives all inputs before a majority is determined. As a failed processor may crash and not respond at all, DVA1 assumes the existence of a timeout utility that precludes postponing voting for receipt of an result that may never arrive. Both DVA1 and TB-DVA enforce a waiting limit that is based on the duration of time before processor should respond. DVA1 postpones a majority determination to allow for sufficient time for collecting the votes. TB-DVA aggregates the waiting periods within the interface module by allowing time for dissension among the processors. In both DVA1 and TB-DVA, this limit is based on anticipating the worst-case possible delay. Although the worst-case may rarely occur, in the conservative approach is regarded as far better than risking the inadvertent loss of operating processors by timing out prematurely. The reduced performance by incorporating timeout periods is accepted as a necessary evil in fault-tolerant distributed systems, but we will show later that TB-DVA offers high performance due, in part, to its treatment of messages.

Creation of the simulation models did not show there would be any drastic increase in design complexity in transitioning from DVA1 to TB-DVA. The models were then simulated to realize the payoff potential of TB-DVA. In the simulation of both models, processor failures were induced. These simulations validated TB-DVA by effectively eliminating errors that were catastrophic for DVA1. Traces of the simulations showed the interface module first capturing the erroneous committal (effectively halting the propagation of the error at the interface module). Second, the simulated TB-DVA restored system integrity by allowing the incorrect committals to be overwritten until the interface module reflected the majority outcome. Thirdly, the simulated timer expired allowing for release of the majority result to the user.

Realistically, simulation could only validate TB-DVA. That is, it confirmed for us that we *built the right thing*, but did not tell us if we had *built it right.* Constraints placed in the simulation model, such as the number of processors, prevent using the simulation results for anything but demonstration - they do not *prove* TB-DVA will always meet its goal. Nevertheless, successfully simulating the algorithm showed its potential advantages. We continued developing TB-DVA by formally verifying the algorithm.

3.5 Correctness

In order to prove the correctness of this algorithm, we turn to Lamport's Temporal Logic of Action (TLA) [45]. This temporal logic is ideally suited to proving properties of distributed or concurrent processes, and has been previously used to specify and verify a Byzantine fault-tolerant system [46].

The first step is to utilize TLA+ (a specification language based on TLA; [47]) to write a precise, formal specification for the algorithm which faithfully followed the steps outlined above. This specification can then be used to prove the partial correctness of TB-DVA (i.e., **if** the algorithm terminates, **then** it produces the correct result). Secondly, the specification can be used to prove termination (i.e., the algorithm always terminates). From these two properties, we can conclude that the algorithm is correct (it terminates, and when it terminates it produces the correct result).

The next section briefly describes some properties and terminology of TLA and TLA+. Then the specification is presented and explained. The two sections afterwards discuss the proofs of partial-correctness and termination.

3.5.1 Temporal Logic of Actions

TLA

TLA is a temporal logic specifically designed to reason about concurrent and distributed algorithms. In this section we will discuss the subset of TLA relevant to the specification and proofs below. More complete information can be found in [45, 47, 48].

All TLA formulas can be expressed using the familiar operators of predicate logic (e.g., $\wedge$, $\vee$, $\neg$), in addition to a few new operators such as $'$ (prime) and $\Box$ (read as *always*), which will be discussed below. TLA is a *state-based* logic – we use it to describe states and state-transitions, where a state is simply a mapping from the set of variable names to the collection of possible values. Every state is universal, in the sense that it assigns some value to every possible variable name (although usually, we are only interested in a small subset of the variables and ignore the rest). TLA is typeless, meaning that any variable can assume any value. If type-invariance is desired, that property must be enforced by the formulas used (i.e., the formulas must make certain not to assign more than one type of value to any one variable).

In order to reason about an algorithm, we have to have a precise way of expressing that algorithm. Normally, an algorithm can be described as a sequence of steps that are gone through in some order. In TLA, a particular execution of an algorithm is described by the sequence of states it goes through. A complete sequence of states gone through by the algorithm is called a *behavior* (technically, a behavior is an infinite sequence of states, where termination of the algorithm is represented by the fact that at some point, the relevant variables stop changing values and remain static). An algorithm is uniquely described by the set of all possible behaviors it can exhibit, i.e., the set of all possible sequences of states that it can go through.

An *action* describes a state transition – it represents a relation between an old state and a new state. An action expression is formed using variables, constants, and primed variables, e.g. $x' = x + 1$. The primed variables refer to the variables of the new state, while the unprimed variables refer to the variables of the old state. Therefore, the example $x' = x + 1$ is saying that the value of the variable x in the new state is 1 greater than the value of x in the old state. An action represents one atomic instruction of a concurrent program.

Temporal formulas are created by using the $\Box$ operator, along with a statement in ordinary predicate logic. $E_1 \wedge \Box(E_2)$, $\Box(E_1 \vee E_2)$, and $\neg\Box(E_1 \Rightarrow E_2)$ are all temporal formulas. We can interpret these formulas as an assertion about a behavior. Any lone expression not associated with an $\Box$ is referring to only the first state of the behavior. The $\Box$ operator asserts that the associated expression is true throughout all states in the behavior. For example, $E_1 \wedge \Box(E_2)$ states that E_1 is true in the first state of the behavior, and E_2 is true in all states of the behavior.

An example can help make this clear. Here is a simple program expressed in pseudocode:

```
int x = 0;
while (TRUE) {
    x = x + 1;
}
```

It declares an integer x, which is initialized to zero, and increments it by one in an infinite loop. The equivalent program in TLA would be expressed as:

$$\begin{aligned} \mathit{Init} &\triangleq x = 0 \\ \mathit{Action} &\triangleq x' = x + 1 \\ \Phi &\triangleq \mathit{Init} \land \Box(\mathit{Action}) \end{aligned}$$

The formula Φ states that in the first state x is equal to zero, and it is always true that the value of x in the next state of the sequence is one greater than the value of x in the previous state.

One final concept used in TLA is that of *stuttering steps.* These are steps which leave the variables involved in the algorithm unchanged. The canonical example for explaining the necessity of stuttering steps is a clock. Consider a specification Π that describes a clock which display hours and minutes. Now consider a specification Ψ which describes a clock displaying hours, minutes, and seconds. Intuitively, Ψ should satisfy Π – the specification of a clock displaying hours, minutes, and seconds should satisfy a specification of a clock displaying only hours and minutes (conceptually, we can just cover up the display of the seconds). However, Ψ contains sequences of 59 steps in which the seconds display changes, but hours and minutes remain the same. Therefore, in order for Ψ to satisfy Π, Π must allow for steps in which hours and minutes do not change.

This is easy to accomplish – we simply include a statement which says that either an action is taken, or that the variables do not change their values. In the case of the example given above which increments x infinitely, this would look like:

$$\begin{aligned} \mathit{Init} &\triangleq x = 0 \\ \mathit{Action} &\triangleq x' = x + 1 \\ \Phi &\triangleq \mathit{Init} \land \Box(\mathit{Action} \lor (x' = x)) \end{aligned}$$

As shorthand, TLA introduces the $[N]_v$ notation, where N is an action and v is a variable (or tuple of variables) associated with that action. The statement

$$\Phi \triangleq \mathit{Init} \land \Box[\mathit{Action}]_x$$

is equivalent to

$$\Phi \triangleq \mathit{Init} \land \Box(\mathit{Action} \lor (x' = x))$$

TLA+

TLA+ is a language built atop the foundation of TLA that enables the precise definition of algorithmic specifications. In this section we will introduce some of the TLA+ operators that are used in the specification of the TB-DVA algorithm. More about TLA+ and the TLA+ operators can be found in [47].

Propositional Logic TLA+ uses the standard operators of propositional logic: $\land$, $\lor$, $\neg$ and all the operators that can be derived form those three. The operators have the standard order of precedence.

Predicate Logic TLA+ also uses the operators of predicate logic – the universal quantifier $\forall$ and the existential quantifier $\exists$.

The CHOOSE Operator The CHOOSE operator is also known as Hilbert's ε. If there exists some x which satisfies an expression p, then CHOOSE $x : p$ is defined to be that x. If there is more than one x possible, then the actual x chosen is arbitrary. If no such x exists, then the value of CHOOSE $x : p$ is undefined.

IF...THEN...ELSE The expression **if** p **then** e_1 **else** e_2 is equal to e_1 if p is true, otherwise it is equal to e_2.

LET The **let** construct allows a local definition within an expression. For instance:

$$\begin{array}{ll} \textbf{let} & x \triangleq a * b \\ \textbf{in} & x * x \end{array}$$

is the same as $(a * b) * (a * b)$

EXCEPT If f is a function, then $[f \text{ EXCEPT } ![e_1] = e_2]$ is equal to f, except that in the new function, $f[e_1]$ is replaced by e_2. For instance, if $f[1] = 1$, $f[2] = 2$, and $f[3] = 3$, then $[f \text{ EXCEPT } ![2] = 4]$ equals the function $\hat{f}$ where $\hat{f}[1] = 1$, $\hat{f}[2] = 4$, and $\hat{f}[3] = 3$.

UNCHANGED UNCHANGED v is shorthand for the expression $(v' = v)$. It states that the variable v does not change value from the old state to the new.

DOMAIN DOMAIN f is the domain of function f.

Junction Lists TLA+ uses junction lists and indentation to eliminate parentheses. A list bulleted with $\wedge$ or $\vee$ indicates the conjunction (disjunction) of the elements of that list. For example:

$$\begin{array}{l} \wedge\ A \\ \wedge\ B \\ \wedge\ \vee\ C \\ \ \vee\ D \end{array}$$

is identical to $A \wedge B \wedge (C \vee D)$.

Sets TLA+ uses the traditional set operators $\in$ (element of), $\notin$ (not an element of), $\cup$ (union), $\cap$ (intersection), and $\subseteq$ (subset). The operator $\setminus$ denotes set difference ($A \setminus B$ is the set of elements that are in A but not in B).

Set Constructors Finite sets can be constructed using one of two methods. $\{x \in S : p\}$ denotes the set of all elements of S which satisfy the expression p. $\{e : x \in S\}$ denotes the set of expressions of the form e for all elements of S.

Functions TLA+ defines a function as a set with an associated domain. Function application is expressed using square brackets, so $f[e]$ is the value obtained from function f with argument e.

Tuples A tuple is a function with domain $\{1, ..., n\}$ that maps i to e_i. Therefore, if e has at least i components, then $e[i]$ is the ith component of e.

3.5.2 The Specification

Explanation of the Specification

This section contains the TLA+ specification for the TB-DVA algorithm. More information about TLA+ can be found in [47]. Here we provide a brief explanation of this particular specification. The actual specification is given immediately after the explanation.

The first section of the specification contains the declarations for various constants and variables, and details assumptions that hold for the specification as a whole. Remember that TLA is a typeless language, and therefore these declarations do not specify what kind of constant or variable is being declared. Type-invariance is a property that, if desired, must be enforced by the specification itself (which this specification does).

The constants are: *Voters*, *Answers*, *RightAnswer*, *Safe*, and four distinct Δs. *Voters* is the set of all voters in the system. *Answers* is the set of all possible answers. *RightAnswer* is the particular answer that is the correct result. *Safe* is an array, each element of which corresponds to a voter. The elements of this array will be TRUE or FALSE to represent, respectively, trustworthy and untrustworthy voters. The Δs are time intervals.

The variables are: *buffer*, *user*, *cv*, *votes*, *rcvd_commit*, *dissented*, *analyzed*, *rdy_commit*, four distinct T variables, and *now*. *buffer* represents the buffer in the interface module, which records each committal. *user* represents the end user that receives the final result. *cv* is a tuple that records the voters that have already committed. *votes* is a two-dimensional array, each row and column of which corresponds to a voter – $votes[i][j]$ represents the vote which voter i received from voter j. *rcvd_commit*, *dissented*, *analyzed*, and *rdy_commit* are all boolean arrays, each element of which corresponds to a voter. For *rcvd_commit*, an element is TRUE if that voter has received a commit from another voter, FALSE if it has not; for *dissented*, an element is TRUE if that voter has had a chance to dissent to a committal, FALSE otherwise; for *analyzed*, an element is TRUE if the corresponding voter has analyzed the results of the various dissents from all the voters, FALSE otherwise; and finally for *rdy_commit*, an element is TRUE if the corresponding voter is ready to commit a value, otherwise it is FALSE. The T and *now* variables have to do with the real-time aspect of the algorithm, and will be explained later.

The assumptions state, in order, that: *RightAnswer* is an element of the set *Answers* (i.e., the right answer is one of the set of possible answers); that '?' is not an element of *Answers*; that all elements of *Safe* are either TRUE or FALSE; that the number of TRUE elements in *Safe* is greater the the number of FALSE elements; and finally that the Δ variables are all real numbers greater than zero. The assumptions regarding *Safe* imply the assumptions that the set *Voters* is finite, and that there are a majority of trustworthy voters.

The second section contains two helpful definitions. The *Card* operation is a recursive function that returns the cardinality of the (finite) set given as an argument. The *var* definition collects the various variables into one convenient tuple.

The third section contains the core of the specification. In this section are defined the initial conditions of the system, and the various actions which model the actual algorithm. The definitions are *Init*, *Commit*(v), *Dissent*(v), *Analyze*(v), and *Terminate*.

The initial conditions are an important part of the algorithm – if these conditions are not met, the behavior of the system would be unpredictable, and likely incorrect. *Init* defines these conditions for TB-DVA: *buffer* and *user* are equal to '?' (i.e., they are not recognized as possible answers; see assumptions above); *cv* is empty (no voters have committed yet); for all trustworthy voters i, *votes*[i][i] equals *RightAnswer* and all other elements of *votes*[i] equal '?' (all trustworthy voters have computed the correct answer and have not received any votes from other voters yet); all elements of *dissented* and *analyzed* equal FALSE (no voter has either dissented to a committal or analyzed any votes); and for all trustworthy voters i, *rdy_commit*[i] equals TRUE and *rcvd_commit*[i] equals FALSE (all trustworthy voters are ready to commit an answer, and have not yet received a committal from another voter). Note that the initial conditions make no statements about what the untrustworthy voters have recorded in *votes*, *rdy_commit*, or *rcvd_commit*. Being untrustworthy, we cannot say anything about their state. *dissented* and *analyzed* are special cases – these are global variables, not local to each voter. Also note that the initial conditions require all trustworthy voters to have already calculated a result before the algorithm begins, as described in the assumptions in section 3.1.

Commit(v) takes a voter v as an argument. This voter is the committing voter. *Commit* also has two local definitions – *answer*, which is defined as *RightAnswer* if v is trustworthy, as some random answer otherwise; and CV, which is defined as the set of elements of the tuple *cv*. The enabling conditions for *Commit*(v) (i.e., the conditions under which the action *Commit*(v) is able to be performed) are: *user* is not an element of *Answers* (the algorithm has not yet terminated); v is not an element of CV (v has not committed before); and *rdy_commit*[v] is TRUE (v is ready to commit). If these conditions are met, then v is added to the tuple *cv* (making sure that v cannot commit again); *buffer* is set to *answer*; all elements of *dissented* and *analyzed* are set to FALSE; and for all trustworthy voters i, *rdy_commit*[i] is set to FALSE (once a voter has committed, no other voter should commit until the committal has been analyzed), *rcvd_commit*[*i*] is set to TRUE (the voter has received a commit), and *votes*[*i*][*v*] is set to *answer* (each voter records the committal value). No statement is made as to what the untrustworthy voters may do.

Dissent(*v*) also takes a voter *v* as an argument. Again, there is a local definition of *answer*, defined the same as in *Commit*(*v*). The purpose of *Dissent*(*v*) is for the voter *v* to have a chance to dissent to the committal if it thinks the committed value is incorrect. The enabling conditions are: *user* is not an element of *Answers* (the algorithm has not terminated); if *v* is trustworthy then *rcvd_commit*[*v*] is TRUE (trustworthy voters should only dissent if they have actually received a committal); and *dissented*[*v*] is FALSE (*v* has not already dissented to this committal). If these conditions are met, then *dissented*[*v*] is set to TRUE (ensuring that *v* cannot dissent again until the next committal); and for all trustworthy voters *i*, if *rcvd_commit*[*i*] is TRUE (there actually is a committal to dissent to), then *votes*[*i*][*v*] is set to *answer* (i.e., *v* broadcasts its dissenting vote, and all trustworthy voters record that vote). Again, no statements are made as to what untrustworthy voters may do.

Analyze(*v*), as with the last two, takes a voter *v* as an argument. It also makes two local definitions – *Majority*(*i*) is defined as the particular element of *vote*[*v*] which a majority of other elements agree with (i.e., the majority vote); *maj* is defined as the result of *Majority*(*i*) if it exists (in other words, if there *is* a majority vote), otherwise it is defined as '?'. The enabling conditions for *Analyze*(*v*) are: *user* is not an element of *Answers* (the algorithm has not terminated); all elements of *dissented* are TRUE (every voter has had a chance to dissent); and *analyzed*[*v*] is FALSE (*v* has not already analyzed the votes). If these conditions are met, then *analyzed*[*v*] is set to TRUE; and if *v* is trustworthy and the majority vote is equal to the committed value, then *rdy_commit*[*i*] is set to FALSE, otherwise it is set to TRUE (if the committed value is correct, then do not commit another result, otherwise indicate that *v* is ready to commit its own result).

Finally, *Terminate* is the last action that can be taken – it is the action that terminates the algorithm. It makes the same local definition of *CV* that was made in *Commit*(*v*).

The enabling conditions are: *user* is not an element of *Answers* (the algorithm has not terminated yet); and for all the voters i which have not committed yet, *analyzed*$[i]$ is TRUE and *rdy_commit*$[i]$ is FALSE (the voters have analyzed the votes, and each has decided not to commit another value). When this condition is met, *user* is set to the value of *buffer* (which contains the value which was last committed). This terminates the algorithm, since all actions require that *user* is not an element of *Answers*, and therefore no more actions can be taken.

The fourth section of the specification takes these actions and uses them to define the actual specification. *Next* is the next step function – it states what the permissible steps of the algorithm are. Here, *Next* is defined as either some voter v takes one of the actions $Commit(v)$, $Dissent(v)$, or $Analyze(v)$, or the *Terminate* action is taken. The specification itself, named Φ, is then defined as $Init \wedge \Box[Next]_{var}$, which means that the system starts in a state satisfying *Init*, and each step either leaves all variables in *var* unchanged, or is one of the actions defined in *Next*.

The final section defines a real-time version of the specification. The initial specification, Φ, sets a safety condition on the algorithm – it will only take an allowable step. However, that statement is satisfied by a series of steps which never change the values of the variables (i.e., time stands still). In order to force the behavior of the specification to take some action, we have to enforce some timing constraints. This is what the Δ constants and the T and *now* variables are for. The Φ^t specification definition basically states that each action ($Commit(v)$, $Dissent(v)$, $Analyze(v)$, and *Terminate*), must take place within Δ time units of when they are first enabled. In other words, once it is possible to take a particular action, that action must be taken within some set time limit. For more information about real-time TLA, consult [48].

TB-DVA TLA+ Specification

module *TB-DVA*

$\text{CONSTANTS } \mathit{Voters}, \mathit{Answers}, \mathit{RightAnswer}, \mathit{Safe}, \Delta_{commit}, \Delta_{dissent}, \Delta_{analyze}, \Delta_{terminate}$

$\text{VARIABLES } \mathit{buffer}, \mathit{user}, \mathit{cv}, \mathit{votes}, \mathit{rcvd\_commit}, \mathit{dissented}, \mathit{analyzed}, \mathit{rdy\_commit}, T_{commit}, T_{dissent}, T_{analyze}, T_{terminate}, \mathit{now}$

$$
\begin{aligned}
\text{ASSUME } & \land\ \mathit{RightAnswer} \in \mathit{Answers} \\
& \land\ ? \notin \mathit{Answers} \\
& \land\ \forall i \in \mathit{Voters} : \mathit{Safe}[i] \in \{\text{TRUE}, \text{FALSE}\} \\
& \land\ \mathit{Card}(\{i \in \mathit{Voters} : \mathit{Safe}[i]\}) > \mathit{Card}(\{i \in \mathit{Voters} : \neg \mathit{Safe}[i]\}) \\
& \land\ (\Delta_{commit} \in \mathit{Real}) \land (\Delta_{commit} > 0) \\
& \land\ (\Delta_{dissent} \in \mathit{Real}) \land (\Delta_{dissent} > 0) \\
& \land\ (\Delta_{analyze} \in \mathit{Real}) \land (\Delta_{analyze} > 0) \\
& \land\ (\Delta_{terminate} \in \mathit{Real}) \land (\Delta_{terminate} > 0)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Card}(S) \ \triangleq\ & \textbf{let } c[R \in \text{SUBSET } S] \triangleq \textbf{if } R = \{\} \textbf{ then } 0 \\
& \qquad\qquad\qquad\qquad\quad \textbf{else } 1 + c[R \setminus \{\text{CHOOSE } r \in R\}] \\
& \textbf{in } c[S]
\end{aligned}
$$

$$\mathit{var} \triangleq \langle \mathit{buffer}, \mathit{user}, \mathit{cv}, \mathit{votes}, \mathit{rcvd\_commit}, \mathit{dissented}, \mathit{analyzed}, \mathit{rdy\_commit} \rangle$$

$$
\begin{aligned}
\mathit{Init} \triangleq\ & \land\ \mathit{buffer} = \mathit{user} = ? \\
& \land\ \mathit{cv} = \langle\rangle \\
& \land\ \forall i, j \in \mathit{Voters} : \mathit{Safe}[i] \Rightarrow \mathit{votes}[i][j] = \textbf{if } i = j \textbf{ then } \mathit{RightAnswer} \textbf{ else } ? \\
& \land\ \forall i \in \mathit{Voters} : \land\ \mathit{dissented}[i] = \mathit{analyzed}[i] = \text{FALSE} \\
& \qquad\qquad\qquad\quad \land\ \mathit{Safe}[i] \Rightarrow \land\ \mathit{rcvd\_commit}[i] = \text{FALSE} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \land\ \mathit{rdy\_commit}[i] = \text{TRUE}
\end{aligned}
$$

```
$Commit(v) \triangleq$ let $answer \triangleq$ if $Safe[v]$ then $RightAnswer$
                                              else CHOOSE $i \in Answers$
                         $CV \triangleq \{cv[j] : j \in$ DOMAIN $cv\}$
                     in $\land\ user \notin Answers$
                        $\land\ v \notin CV$
                        $\land\ rdy\_commit[v] =$ TRUE
                        $\land\ cv' = \langle v \rangle \circ cv$
                        $\land\ buffer' = answer$
                        $\land\ \forall i \in Voters : \land\ dissented[i]' =$ FALSE
                                          $\land\ analyzed[i]' =$ FALSE
                                          $\land\ Safe[i] \Rightarrow \land\ rdy\_commit[i]' =$ FALSE
                                                        $\land\ rcvd\_commit[i]' =$ TRUE
                                                        $\land\ \forall j \in Voters \setminus \{i\}$ :
                                                            $votes[i][j]' = answer$
                                                        $\land$ UNCHANGED $\langle votes[i][i] \rangle$
                        $\land$ UNCHANGED $\langle user \rangle$

$Dissent(v) \triangleq$ let $answer \triangleq$ if $Safe[v]$ then $RightAnswer$
                                               else CHOOSE $i \in Answers$
                      in $\land\ user \notin Answers$
                         $\land\ Safe[v] \Rightarrow rcvd\_commit[v] =$ TRUE
                         $\land\ dissented[v] =$ FALSE
                         $\land\ dissented' = [dissented$ EXCEPT $![v] =$ TRUE$]$
                         $\land\ \forall i \in Voters : Safe[i] \Rightarrow votes[i]' =$
                            if $(votes[i][v] \neq answer) \land (rcvd\_commit[i] =$ TRUE$)$
                              then $[votes[i]$ EXCEPT $![v] = answer]$
                              else $votes[i]$
                         $\land$ UNCHANGED $\langle buffer, user, cv, analyzed, rcvd\_commit, rdy\_commit \rangle$
```

```
$Analyze(v) \triangleq$ let $Majority(i) \triangleq Card(\{j \in Voters : votes[v][j] = i\}) >$
                              $Card(\{j \in Voters : votes[v][j] \neq i\})$
                            $maj \triangleq$ if $\exists\, i \in Answers : Majority(i)$
                                        then CHOOSE $i \in Answers : Majority(i)$ else ?
                        in $\wedge\ user \notin Answers$
                            $\wedge\ \forall\, i \in Voters : dissented[i] = \text{TRUE}$
                            $\wedge\ analyzed[v] = \text{FALSE}$
                            $\wedge\ analyzed' = [analyze \text{ EXCEPT } ![v] = \text{TRUE}]$
                            $\wedge\ Safe[v] \Rightarrow rdy\_commit[v]' =$ if $maj \neq buffer$ then TRUE
                                                                                         else FALSE
                            $\wedge\ \forall\, i \in Voters \setminus \{v\} : \text{UNCHANGED } \langle rdy\_commit[i] \rangle$
                            $\wedge\ \text{UNCHANGED } \langle buffer, user, cv, dissented, votes, rcvd\_commit \rangle$

$Terminate \triangleq$ let $CV \triangleq \{cv[j] : j \in \text{DOMAIN } cv\}$
                       in $\wedge\ user \notin Answers$
                          $\wedge\ \forall\, i \in Voters \setminus CV : \wedge\ analyzed[i] = \text{TRUE}$
                                                          $\wedge\ rdy\_commit[i] = \text{FALSE}$
                          $\wedge\ user' = buffer$
                          $\wedge\ \text{UNCHANGED } \langle buffer, cv, dissented, analyzed, votes,$
                            $rcvd\_commit, rdy\_commit \rangle$
```

```
$Next \triangleq \vee\ \exists\, v \in Voters : \vee\ Commit(v)$
                                       $\vee\ Dissent(v)$
                                       $\vee\ Analyze(v)$
                $\vee\ Terminate$

$\Phi \triangleq Init \wedge \Box[Next]_{var}$
```

$$\begin{aligned}
\Phi^t \triangleq\ & \land\ \Phi \land RT_{var} \\
& \land\ \forall\, v \in Voters : \land\ VTimer(T_{commit}[v], Commit(v), \Delta_{commit}, var) \\
& \qquad\qquad\qquad\quad\ \land\ MaxTime(T_{commit}[v]) \\
& \qquad\qquad\qquad\quad\ \land\ VTimer(T_{analyze}[v], Analyze(v), \Delta_{analyze}, var) \\
& \qquad\qquad\qquad\quad\ \land\ MaxTime(T_{analyze}[v]) \\
& \qquad\qquad\qquad\quad\ \land\ VTimer(T_{dissent}[v], Dissent(v), \Delta_{dissent}, var) \\
& \qquad\qquad\qquad\quad\ \land\ MaxTime(T_{dissent}[v]) \\
& \land\ VTimer(T_{terminate}, Terminate, \Delta_{terminate}, var) \\
& \land\ MaxTime(T_{terminate})
\end{aligned}$$

3.5.3 Proof of Partial Correctness

The proof of partial-correctness is long (approximately 13 pages), but in concept it is quite simple. An algorithm is partially-correct if the following statement is true: **if** the algorithm terminates, **then** it produces the correct result. In terms of the specification, we can describe the same statement thusly:

$$\mathit{PartialCorrectness} \triangleq \Phi \Rightarrow \Box(\mathit{user} \in \mathit{Answers} \Rightarrow \mathit{user} = \mathit{RightAnswer})$$

In other words, the specification (Φ) implies that it is always true that **if** $\mathit{user} \in \mathit{Answers}$ (i.e., the buffer has passed an answer to the user and therefore the algorithm has terminated), **then** $\mathit{user} = \mathit{RightAnswer}$ (the algorithm has produced the correct result).

This property is an *invariant* – the statement is true throughout the entire execution of the algorithm. In order to prove this, we will take advantage of one of the rules of TLA, namely the rule INV1:

$$\frac{I \wedge [N]_f \Rightarrow I'}{I \wedge \Box[N]_f \Rightarrow \Box I}$$

This statement says that if I remains true after each and every step of the algorithm, then we can conclude that I is an invariant (i.e., it is always true). In the case of our specification, I is the statement $\mathit{user} \in \mathit{Answers} \Rightarrow \mathit{user} = \mathit{RightAnswer}$, $N = \mathit{Next}$, and $f = \mathit{var}$. However, we cannot prove this statement directly, because the specification is too complicated. We'll need to use an intermediate invariant Inv. Therefore there are three statements we need to prove in order to prove the partial-correctness of TB-DVA:

1. $\mathit{Init} \Rightarrow \mathit{Inv}$
2. $\mathit{Inv} \wedge [\mathit{Next}]_{\mathit{var}} \Rightarrow \mathit{Inv}'$
3. $\mathit{Inv} \Rightarrow (\mathit{user} \in \mathit{Answers} \Rightarrow \mathit{user} = \mathit{RightAnswer})$

From these, we can then conclude:

$$
\begin{aligned}
\Phi \;\Rightarrow\;& \{\text{By definition of } \Phi\}\\
& Init \land \Box[Next]_{var}\\
\Rightarrow\;& \{\text{By (1) above}\}\\
& Inv \land \Box[Next]_{var}\\
\Rightarrow\;& \{\text{By (2) above and rule INV1}\}\\
& \Box Inv\\
\Rightarrow\;& \{\text{by (3) above}\}\\
& \Box(user \in Answers \Rightarrow user = RightAnswer)
\end{aligned}
$$

The complete proof is given in Appendix A.

3.5.4 Proof of Termination

The proof of termination is much shorter and simpler than the proof of partial-correctness. We can prove it using a simple loop termination argument. To show this, Figure 3.1 provides a graphical view of the flow of control of the TB-DVA algorithm. It can be clearly seen that essentially the algorithm loops through the *Commit*, *Dissent*, and *Analyze* functions until the criteria are met for falling through to *Terminate*. The basic argument for the worst-case scenario is that each time the *Commit* action is taken, the number of voters left that are able to commit is decremented by one (since each voter can commit at most once). In the worst case, eventually there will be no voters left able to commit. However, that circumstance satisfies the exit criteria for the loop, and the algorithm terminates. The actual proof is consistent with scenarios in which all the voters attempt to commit (hence exhausting the set of available voters), as well as those in which the remaining voters decide not to commit, thereby leaving some voters able to commit, but not willing to do so. The algorithm terminates in either case.

To formalize this argument, we select a boundary variable t, which has three important properties:

1. Initially, $t > 0$.

2. $t = 0 \Rightarrow$ the algorithm terminates (i.e., formula B is satisfied).

3. t is decremented each time through the loop.

By proving these three properties, we prove that the loop must terminate. For this proof, $t = Card(Voters) - Card(CV)$ (the difference between the cardinality of the set of all voters and the cardinality of the set of voters who have committed – i.e., t equals the cardinality of the set of voters who have not yet committed). The complete proof is given in Appendix B.

3.5.5 Lessons Learned

Using TLA and TLA+, we have formally specified and verified the TB-DVA secure distributed voting algorithm. As expected, the process of formalizing the algorithm led to the discovery of ambiguities or errors in the initial formulation of the algorithm, which were subsequently corrected. As an example, initially the algorithm provided for the case where a voter would commit a value to the user before the other voters had a chance to calculate a result – if the other voters did not have a result, then they did not know whether they should dissent or not. We overcame this difficulty by providing a special 'not ready' vote, which a voter would use for exactly this circumstance. A majority of 'not ready' votes would then force the timer in the interface module to stop, giving the voters time to finish their computations. However, upon closer examination this was not a very effective method. It is possible for each voter to finish its calculations before any of the remaining voters do (e.g., the first voter finishes before the others and commits; the other voters vote 'not ready' and stop the timer; the second voter finishes before the rest and commits; the other voters vote 'not ready' and stop the timer; *et cetera*). In such a situation, a faulty voter could commit an incorrect result, and there may not be a majority of trustworthy voters left that have not committed. We solved this difficulty by getting rid of the 'not ready' votes and establishing a global time threshold, beyond which all functional voters should have calculated a result.

Any voter which tries to commit before this time is ignored; therefore all trustworthy voters are guaranteed to have a result ready when a value is committed.

3.6 Performance

When analyzing the performance of the TB-DVA algorithm, we will restrict ourselves to the same assumptions as those used by DVA1, the 2-phase commit protocol analyzed earlier. The purpose of this is to give a fair comparison of performance between that widely-known and widely-used algorithm and TB-DVA, since under the less restrictive assumptions used by TB-DVA the 2-phase commit protocol would not work correctly, and a comparison would be futile. As a reminder, the DVA1 assumptions were that the processors would only fail in one of two ways – either they would simply halt, or they would follow the protocol but use an incorrect result. They are not allowed to fail arbitrarily, as in the TB-DVA algorithm. Because of this assumption (made because DVA1 is concerned solely with fault-tolerance, not security), another assumption made is that there may be fewer than a majority of correct processors. That is why provision is made for a self-diagnostic routine in the processors, which they can use to determine whether or not they are faulty. Of course, in a security-conscious environment, processors cannot be trusted to tell the truth about whether or not they are faulty, which is why TB-DVA does not make the same provisions. However, again for the sake of making a fair comparison, we will make the assumption for this analysis that there may not be a majority of trustworthy processors (since we are concerned here with *performance*, rather than *correctness*, which was covered in the previous section, this assumption does not really hurt us).

For TB-DVA the total time taken to send the final answer to the user is composed of four parts - the time for each processor to compute its result, the time for any dissenting votes to be broadcast, the time for the processors to analyze the dissenting votes to determine a majority, and the time added by repeating this process as necessary. Therefore the computational complexity for TB-DVA can be expressed as

$$T_{total}(n) \quad = \quad T_{compute}(n) + T_{repeat}(n) \times (T_{vote}(n) + T_{analyze}(n))$$

As with DVA1, since the processors are operating in parallel, the time for them to compute their results is independent of the number of processors and $T_{compute}(n) = \Theta(1)$.

In the worst case, all the processors disagree with the coordinator and broadcast a dissenting vote, meaning that $T_{vote}(n) = O(n)$. Again in the worst case, all these dissenting votes also disagree with each other, meaning that $T_{analyze}(n) = O(n^2)$. However, if all the processors disagree, there can only be $O(1)$ number of repeats since there is not more than one processor with the correct answer to commit. On the other hand, if all the dissenting votes agree with each other, $T_{analyze}(n) = O(n)$ in the first round of the algorithm, and (since now all the dissenting processors have the correct vote) $T_{repeat}(n) = O(n)$. However, remember that previously-committed voters are still allowed to dissent. This means that in later rounds of the algorithm, more and more of the votes will disagree, and $T_{analyze}(n) = O(n^2)$ again, while $T_{repeat}(n) = O(n)$. Therefore, the worst-case running time of the algorithm is

$$T_{total}(n) = O(n^3)$$

This result is worse than the running time of DVA1. However, the above analysis was made assuming that any number of voters can fail, which directly contradicts our initial assumption that a majority of voters will be correct. By violating this assumption, we created a worst-case possible performance scenario. This was done intentionally to compare TB-DVA and DVA1 as they *try* to deliver an unobtainable majority result. Unless recovery of failed processors and their near-immediate restoration to service is assumed, neither algorithm could, in this case, *guarantee* delivery of a correct result. Being an aberration, this case requires external intervention to rescue the voting process. Therefore, any performance penalty TB-DVA would suffer could be made inconsequential by this intervention.

Now looking at the best case, all the processors agree with the original coordinator and there are no dissenting votes. Therefore $T_{vote}(n) = \Omega(1)$, $T_{analyze}(n) = \Omega(1)$ (since each processor only needs to compare its result to the coordinator's), and $T_{repeat}(n) = \Omega(1)$ (since the original committal was correct). Therefore, in the best-case the running time of the algorithm is

$$T_{total}(n) = \Omega(1)$$

Again assuming a low failure rate, this best-case running time is a good estimate of the average performance of the algorithm, which greatly improves on the $\Omega(n)$ performance of DVA1; and while the worst-case running time is worse than that of DVA1, the worst-case scenario only takes place when a basic assumption of TB-DVA is violated, as described above. In addition, as we will see next while the worst-case TB-DVA time is asymptotically greater than DVA1, the quantitative worst-case performance is better than that of DVA1 for all practical purposes.

Of course, any estimation of performance must include the additional latency caused by the timer in the interface module. The number of times that the algorithm repeats in order to correct an incorrect committal can be set by the user. The two extremes would be setting the number of retries to one (meaning that the processors only get one chance to correct a faulty committal) or setting the number of retries to one less than the number of processors (meaning that each processor would get a chance to correct a faulty committal). The benefit in security comes from the fact that even if an attacker manages to compromise the coordinator, an incorrect result cannot be committed without the uncompromised processors correcting it. With the number of retries set to one less than the number of processors, an attacker would have to compromise a majority of the processors before being able to commit an incorrect result. If the objective is better performance, the number of retries should be set to one (setting retries to zero gives optimal performance, but no fault-tolerance at all). With one retry allowed, the algorithm still provides comparable fault coverage and better security than DVA1. We assume for this analysis that we have set the number of recommits to one, as well as assuming that no two faulty votes agree and that the comparison time takes up the bulk of the performance, as in the previous analysis. Again, n is the number of processors and ϕ is the probability of failure of the processors.

The interface module must wait long enough after a result has been committed to see if there are any dissenting votes that might cause a recommittal. This time is approximately the time needed to make one comparison between votes, so that each processor (in parallel) can compare its result with the coordinator's and broadcast a dissenting vote if necessary. Of course, if fewer than a majority of the processors have dissenting votes no recommittal is possible. Therefore, the interface module must wait for the length of time of one comparison (which is application specific) and see if a majority of processors have dissenting votes. If less than a majority disagree, then when the timer expires it can pass the result on to the user. If a majority of processors do disagree, the interface module must reset its timer to allow time for the dissenting votes to be analyzed, and possibly a second result to be committed. Since the interface module only has information on how many dissenting votes there are, not on what is contained in the votes, it must assume the worst case (that all the votes disagree) and set the new timer to the amount of time it would take to compare all the dissenting votes given that assumption (only the dissenting votes need to be compared, since by definition they all disagree with the coordinator, and processors that agreed with the coordinator did not vote). Of course, if there is a recommittal the interface module can pass it straight to the user without waiting any longer – we have set the number of possible recommittals to one, therefore we know that another result will not be recommitted. This chain of events is modeled in Figure 3.2. The initial time is designated t_1, the probability of needing to wait longer because of dissenting votes is σ, and the amount of time needed to wait for analysis and possible recommittal is t_2.

From this model it is clear that the estimated time for this algorithm to complete is

$$E(t) = t_1 + \sigma t_2$$

As mentioned earlier, t_1 is simply the time for one comparison, so $t_1 = 1$. The probability of a majority of dissenting votes (σ) is the probability that the coordinator is wrong (in which case, all the processors, correct or incorrect, will disagree) plus the probability that the coordinator is correct and a majority of other processors are wrong, as given by

$$\begin{aligned}\sigma &= P(p_0 \text{ wrong } \cup\ p_0 \text{ right and majority of others wrong}) \\ &= \phi + (1-\phi)\left(\sum_{i=\lceil\frac{n}{2}\rceil}^{n-1} \binom{n-1}{i} \phi^i (1-\phi)^{n-1-i}\right)\end{aligned}$$

where i is the number of faulty processors. t_2 can be broken up into two cases: either there is a recommit (in which case the interface module passes the result on to the user without waiting for the timer to expire) or there is not a recommit (in which case the timer must expire before the interface module can pass the result on to the user). The first case (designated t_{21}) is the probability of a recommit given that there were a majority of dissenting votes, multiplied by the number of expected comparisons. The expected number of comparisons is calculated in the same way as it was in the previous analysis, so

$$\begin{aligned}E_c(n,i) &= \sum_{j=0}^{n-i} (P(\text{first correct vote is in the } jth \text{ element}) \times (\#\text{ of comparisons})) \\ &= \sum_{j=0}^{n-i}\left(\frac{\binom{n-1-j}{i-1}}{\binom{n}{i}}\left((j+1)n - \frac{(j+1)(j+2)}{2} + \frac{(n-i-j)(n-i-j-1)}{2}\right)\right)\end{aligned}$$

From inspection of the TB-DVA algorithm, there can only be a recommit if the coordinator was wrong and at least one other processor is correct. Therefore, t_{21} can be derived as

$$\begin{aligned}t_{21} &= P(p_0 \text{ wrong } \cap \text{ at least one right} \mid p_0 \text{ wrong } \cup\ p_0 \text{ right, majority wrong}) \\ &\quad \times(\#\text{ comparisons}) \\ &= \left(\frac{P((p_0 \text{ wrong } \cap \text{ at least one right}) \cap (p_0 \text{ wrong } \cup\ p_0 \text{ right, majority wrong}))}{P(\ p_0 \text{ wrong } \cup p_0 \text{ right, majority wrong})}\right) \\ &\quad \times(\#\text{ comparisons}) \\ &= \left(\frac{P(p_0 \text{ wrong } \cap \text{ at least one right})}{\sigma}\right) \times (\#\text{ comparisons}) \\ &= \frac{1}{\sigma}\left(\sum_{i=0}^{n-2}\left(\binom{n-1}{i}\phi^{i+1}(1-\phi)^{n-1-i}E_c(n,i)\right)\right)\end{aligned}$$

The second case (t_{22}) is the probability of no recommit given that there were a majority of dissenting votes. For there to be no recommit, either the coordinator was correct, or all the processors were faulty. The number of comparisons that the timer needs to wait for in this instance is always the worst case - the number of comparisons among all the dissenting votes if all the votes disagree. t_{22} can be derived as follows:

$$\begin{aligned}
t_{22} &= P(p_0 \text{ right} \cup \text{all wrong} \mid p_0 \text{ wrong} \cup p_0 \text{ right, majority wrong}) \times (\#\text{ comparisons}) \\
&= \left(\frac{P((p_0 \text{ right} \cup \text{all wrong}) \cap (p_0 \text{ wrong} \cup p_0 \text{ right, majority wrong}))}{P(p_0 \text{ wrong} \cup p_0 \text{ right, majority wrong})}\right) \\
&\quad \times (\#\text{ comparisons}) \\
&= \left(\frac{P(p_0 \text{ right } \cap \text{ majority wrong})}{\sigma} + \frac{P(\text{all wrong})}{\sigma}\right) \times (\#\text{ comparisons}) \\
&= \frac{1}{\sigma}\left(\left(\sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} \left(\binom{n-1}{i} \phi^i (1-\phi)^{n-i} \left(\frac{i(i-1)}{2}\right)\right)\right) + \left(\phi^n \left(\frac{(n-1)(n-2)}{2}\right)\right)\right)
\end{aligned}$$

t_2 equals $t_{21} + t_{22}$. From these values for t_1, σ, and t_2, we can now calculate $E(t)$. If we graph this function with $2 \leq n \leq 11$ and $.01 \leq \phi \leq 0.5$, we obtain the graph in Figure 3.3. As expected from the asymptotic analysis, at low failure probabilities TB-DVA is essentially constant with respect to the number of processors, giving a better performance than DVA1. As the probability of failure increases, TB-DVA grows faster than DVA1, but still improves on performance even up to a failure probability of 0.5.

Restrictions on Performance Analysis

The probability analysis above was made for the case of only one recommit. Unfortunately, the analysis for an arbitrary number of recommits is much more difficult – so much so as to be impractical. The sequence of events cannot accurately be modeled by a Markov chain, because the probabilities for each state depend on all the previous states that lead to it. This is true because the probabilities depend on the number of non-faulty and faulty voters. Once a voter turns faulty, it remains faulty forever (in terms of the task at hand), and a non-faulty voter can turn faulty at any time. The probabilities for each state then depend on how many voters have turned faulty up to that point.

A concrete example should help make this point clear. Assume a case with five voters, all of which are initially non-faulty. The probability of a recommit is then equal to the probability that the coordinator fails, but at least one voter remains non-faulty. This can be stated in the following equation, where ϕ is the probability that a particular voter fails:

$$P(recommit) = \sum_{i=1}^{4} \left(\phi^i (1-\phi)^{5-i}\right)$$

There are then a total of four different ways that a recommit could happen: the coordinator could fail, but all the other voters remain non-faulty; the coordinator and one other voter fail, but the rest remain non-faulty; the coordinator and two other voters fail, but the rest remain non-faulty; and the coordinator and three other voters fail, but the last voter remains non-faulty.

Now we can try to calculate the probability of a second recommittal. Remember that the first coordinator is now permanently faulty (since it committed an incorrect value), so there are now only four voters to consider. Again, the probability of a recommit is equal to the probability that the coordinator fails, but at least one voter remains non-faulty. However, this probability depends on how many voters failed in the previous step. If only the initial coordinator failed, then the probability would be:

$$P(recommit) = \sum_{i=1}^{3} \left(\phi^i (1-\phi)^{4-i}\right)$$

However, if it was the case that one other voter failed besides the initial coordinator, then that voter is still faulty and the probability would become:

$$P(recommit) = \sum_{i=1}^{2} \left(\phi^i (1-\phi)^{3-i}\right)$$

In fact, for each of the four cases listed above for the first recommit, there is a different probability for the second recommit. And similarly to the first recommit, there are three different ways that a second recommit could happen, each of which entails a different probability for a third recommit; and there are two different ways for a third recommit to happen, each of which entails a different probability for a fourth recommit (where the progression ends, since for five voters there can be a maximum of four recommits). Clearly, when we generalize this to n voters, the problem of analyzing the quantitative performance of the algorithm scales as $O(n!)$.

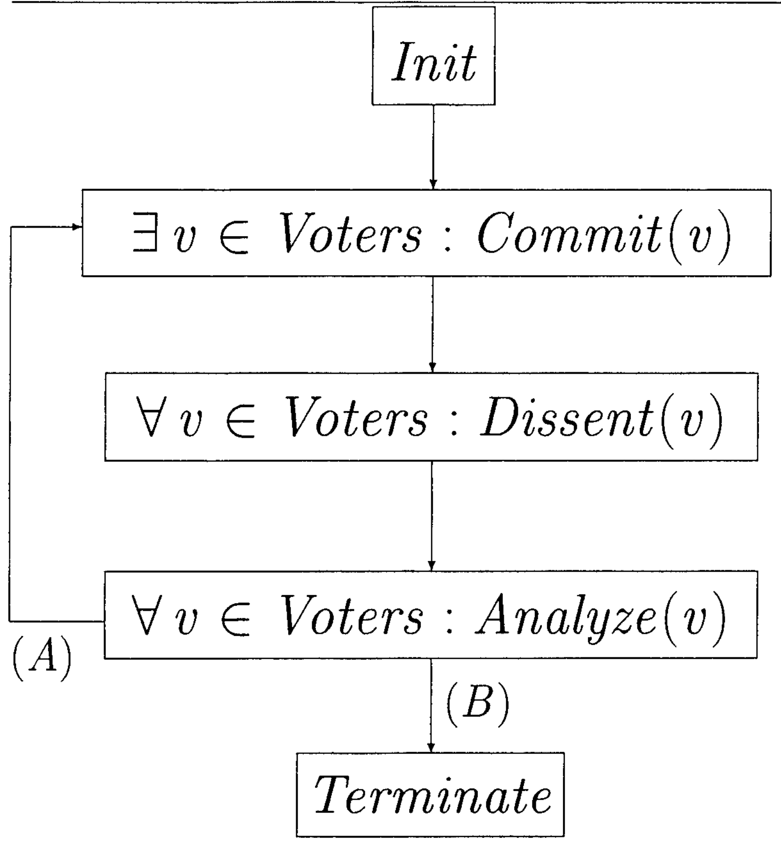
However, the asymptotic analysis of the algorithm can still give us the absolute worst case for performance, which was calculated as $O(n^3)$ above. We can even narrow this down to a more exact value. The worst case arises when each voter remains non-faulty up until the point it commits, when it fails. This would entail a maximum number of recommits (that being $n-1$ total). For each recommit, the voters would need to compare one more faulty vote (since the old coordinators remain faulty, and still vote on subsequent committals once they commit an incorrect value). So the total number of comparisons in the worst case can be calculated by the following formula, where i equals the number of faulty votes:

$$\sum_{i=1}^{n-1}\left(\left(\sum_{j=1}^{i}(n-j)\right)+(n-i-1)\right)$$

which is consistent with the asymptotic analysis. Naturally, the best case remains $\Omega(1)$ no matter how many recommits we take into account, since in the best case no recommits are needed at all.

3.7 Intrusion Tolerance

Another benefit of this algorithm that has yet to be fully explored is its applicability to the problem of intrusion tolerance. Any voter that commits an incorrect value can be partitioned from the network and flagged for review by a higher authority (either automated or human) as a possible security breach. Assuming that all voters are denied access to covert channels, we can also have each voter monitor all other voters, and in a similar manner flag any voter that is releasing confidential information. This could be a valuable property to have, and is a good candidate for future research.


$Init$
$\exists\, v \in Voters : Commit(v)$
$\forall\, v \in Voters : Dissent(v)$
$\forall\, v \in Voters : Analyze(v)$
(A)
(B)
$Terminate$

$$A = \exists\, v \in Voters \setminus CV \ : \ rdy\_commit[v] = \text{TRUE}$$

$$B = \neg A \ (i.e., \forall\, v \in Voters \setminus CV \ : \ rdy\_commit[v] = \text{FALSE})$$

Figure 3.1: TB-DVA Flowchart

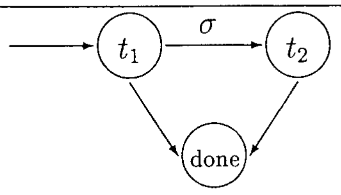


Figure 3.2: State Transition Model of Interface Module's Timer

Wed Jan 12 14:16:43 2000

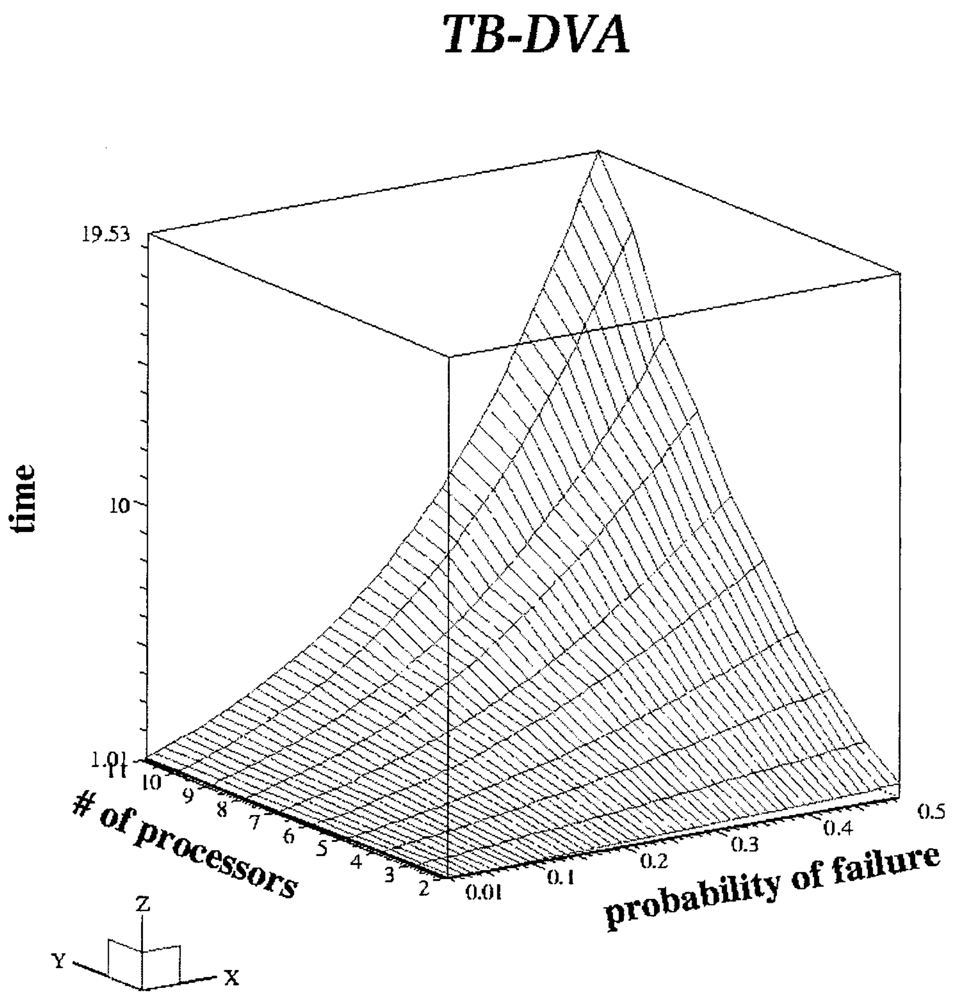


Figure 3.3: Graph of the expected time for TB-DVA versus the number of processors and probability of failure

Chapter 4

Digital Signature Distributed Voting

This chapter discusses some preliminary work on secure voting in a wide-area network, where the assumptions of atomic broadcast and bounded message delay that were made for TB-DVA are not practical. We take advantage of Lamport's results described in [33], where he concludes that Byzantine fault-tolerance can be much simplified through the use of digital signatures. Again, the unique aspect of this algorithm, just as for the previous, is the way it juxtaposes the requirements for security, fault-tolerance, and performance in inexact voting.

4.1 Assumptions

The environment for this algorithm is a wide-area network. While the underlying network itself may be unreliable, we assume that this algorithm operates on top of a reliable transport protocol, guaranteeing eventual delivery of messages (although the messages are not necessarily delivered in the order they were sent). On top of this layer is another layer which guarantees eventual delivery of *valid* messages - messages which have been digitally signed and correctly verified as described in the next paragraph. Messages which cannot be verified are discarded. We assume the presence of a public-key infrastructure ([43]), in which each voter has a private key and each voter knows (or can securely obtain) the public key of every other voter. Each voter knows *a priori* who the other voters are. We further assume that a majority of the voters are are fault-free and will correctly follow the protocol (i.e., they are *trustworthy*). As before, no assumptions are made about the remaining voters. There is no interface module in this system – just the voters and the client. The ultimate goal of the algorithm is to have each trustworthy voter agree with every other trustworthy voter on one final result, and to have proof that its result is that which was agreed on.

Two different functions are employed in the algorithm: *one-way hashes* and *digital signatures.* A one-way hash is a function that maps an argument to a unique fixed-width value in such a way that it is impossible to recover the original argument from that value. This function can be made cryptographically secure, i.e., it is extremely difficult to construct a message that will hash to a given hash value [43]. A digital signature is simply a cryptographic attachment that can be used to validate a message. This can be accomplished in several ways; one mechanism is encrypting a message (or the hash of a message) with a private key. The signature can be verified as valid by decrypting the signature with the corresponding public key. This provides a secure method of authentication. All signatures include a timestamp to guard against replay attacks.

4.2 Description

Each (trustworthy) voter will follow the steps below:

1. Compute a result.
2. Compute the hash of the result and save that value.
3. Sign the result and send it to all the other voters.
4. For all the signed results received from the other voters:
    (a) Make sure that this result is not a repeat (i.e., there is only one result per voter).
    (b) Verify the signature to make sure it is a valid result.

   (c) If the result agrees with this voter's result (using inexact comparison if necessary), then hash the other voter's result, sign the hash, and send it back to the other voter (this signed hash is called an *endorsement*).

5. For all endorsements received from the other voters:

   (a) Make sure the endorsement is not a repeat (i.e., only one endorsement per voter).

   (b) Verify the signature and compare the hash value to the value saved in step 2 in order to make sure it is a valid endorsement.

6. Once a majority of endorsements have been received, the algorithm is terminated.

The voters end up with a majority of endorsements for their result, and once a majority vote has been determined the voters can, if necessary, transmit the result to any interested host along with the relevant endorsements. The host can accept the first such result accompanied by a majority of endorsements which are all verified correctly, knowing that that vote is the result agreed to by a majority of the voters. We are guaranteed that a majority of endorsements will be received by correct voters because of the assumption that a majority of the voters will operate correctly.

4.3 Discussion

The goal of the algorithm, as stated earlier, is to enable voters to agree on a common result *and* provide proof that their result is the one that was agreed on. It must do this in an environment where all messages must pass through unknown (and possible untrustworthy) intermediary nodes, and where all of the voters are not themselves necessarily trustworthy.

The mechanism that makes this possible is the public-key digital signature. With this, voters are able to determine the originator of a message and verify that no-one tampered with the message before it was received. This means that the intermediary nodes cannot influence any of the voters – they can only relay messages. It also means that no voter can masquerade as another voter, nor can any voter fake an endorsement from any other voter.

In the second round of the voting algorithm, signing the hash rather than the result itself is a convenience. The result may be of any size from a simple number to a multi-field record depending on the application, while the hash would always be a constant size (e.g. 160 bits). If the result itself were going to be signed, then there would be one of two options. One would be that the voters could exchange signed votes, in which case each voter would have to store multiple copies of the same vote, each signed by a different voter. In order to prove to a host that the result was correct, a voter would have to transmit each of the votes to the host, which would in turn have to verify and compare them all. The other option would be that the voters could each in turn sign a vote, so that each vote would be signed multiple times. This would necessitate that the vote from each voter be sent to a majority of the other voters, greatly expanding the number of messages necessary.

The requirement for timestamps for each signature is there in order to guard against resend attacks. An attacker could record the messages sent in a previous run of the algorithm and resend them to the voters in a subsequent run. If there was no way of determining that these were old messages, the voters could be fooled into accepting them as valid votes. But since the hashes of these votes would not match the hashes of the voters' results, the votes would be discarded and the voters would not be able to agree on a majority result – even though a majority of them may be functioning correctly.

The distinguishing feature of this algorithm versus the secure voting protocol described in section 2.5, which also used digital signatures, is that this algorithm is capable of inexact voting. The previous protocol used partial signatures, which were then combined into a whole signature before being sent to the client. While this means that the client only has to have one public key in order to verify the result, it also means that all the partial results must be identical in order to be combined together. The algorithm presented here requires that the client possess public keys for all the voters involved, but it does allow inexact comparisons to be used.

4.4 Performance

Performance of the algorithm can be measured by the complexity of the operations required of each voter and the number of messages required to be sent over the network. In the following analysis, n is the number of voters.

The first step for each voter is to calculate its result and sign and hash that result. Since each voter does this only once, and in parallel, this can be taken as a constant. Each voter will then receive one signed vote from every other voter. For each signed vote the voter must verify it and compare it with its own result. Since this may be inexact voting, the comparison may be computationally expensive. If the vote agrees with the voter's result, the voter hashes and signs the vote (a trivial operation relative to the comparison). Each voter will then receive a maximum of one endorsement from every other voter, which they will have to verify and compare with the hash of their own result. The complexity for each voter is therefore $O(n)$. Every voter sends one signed vote to every other voter, resulting in $n(n-1)$ messages. Each voter then sends at a maximum one endorsement to every other voter, causing another $n(n-1)$ messages, for a total of $2n(n-1)$ messages. Therefore the complexity of the algorithm with regards to the number of messages is $O(n^2)$.

Chapter 5

Conclusion

The purpose of this technical report was to present a brief history and explanation of distributed voting; to motivate a search for security-oriented distributed voting algorithms; to analyze the weaknesses of current secure voting algorithms; and to present a new algorithm which overcomes those weaknesses. Portions of this report have been documented elsewhere [49, 50, 51].

In Chapter 1 we argued a case for designing algorithms with both fault-tolerance and security in mind, rather than designing them separately and then combining them at a later time. Security and fault-tolerance were compared and contrasted, and it was pointed out that ignoring one in favor of the other left blind spots in which potential errors were overlooked. Several examples of exactly this phenomenon were illustrated.

In Chapter 2 we presented a brief history of distributed voting and demonstrated the reasons it is an important and relevant topic as the world becomes ever more reliant on distributed technology. We then described and analyzed in detail a well-known and popular distributed voting scheme, the 2-phase commit protocol. We found several weaknesses in this protocol, relating to both security and performance. We then examined several alternatives, including Byzantine algorithms and some proposed secure voting algorithms, but ultimately dismissed them as being unsuitable – they were either to costly, too application specific, or not sufficiently high-performance.

In Chapter 3 we introduced the Timed-Buffer Distributed Voting Algorithm, our own proposed solution for secure, fault-tolerant distributed voting. We discussed the underlying assumptions which the algorithm relied on, and formally specified and verified the algorithm to ensure it was correct and fulfilled our requirements. We next analyzed the performance of the algorithm in order to compare it with the 2-phase commit protocol discussed earlier; our algorithm compared very favorably. We also discussed the possibility of future research in a potential benefit of our algorithm for intrusion tolerance – an unforeseen but very useful capability.

In Chapter 4 we presented some preliminary work on a second distributed voting algorithm, this one growing out of the question of how to do secure voting in an environment where our earlier assumptions for TB-DVA did not hold. This work holds relevance for many wide-area network applications, and also has the potential to bear fruit for future research.

This report delved into using multiple processors to make decisions based on information that could be inaccurate, imperfect, and incomplete. We pointed out that many voting algorithms exist in the literature, but due to the increased hostility in distributed computing systems, this report focused on securing a majority outcome premised on the need for an efficient protocol that deals with uncertainty about a voter's intent. Incidentally, the 2000 presidential election became embroiled with manually discerning partially completed voting ballots (i.e., the so-called "dimples" and "chads" of machine-readable data cards having incomplete punch-throughs). This current event indicates the difficulty that can ensue from having to resolve inexact votes, and the algorithms we developed were premised on the need for inexact voting, but they also accommodate exact voting. Most importantly, we preserve the integrity of the majority result from tampering by a malicious voter. We covered the design, implementation and formal verification of this unique approach towards satisfying the security and fault tolerance needs of distributed information systems.

Bibliography

[1] "Targeting Cyberterrorism: Government Declares War to Protect USA's Infrastructure," *USA Today*, October 20, 1997.

[2] Anderson, T., et al., *Dependability: Basic Concepts and Terminology*, Springer-Verlag/Wien, 1992.

[3] Randell, B., "Dependability - a Unifying Concept," *Conference Proceedings of Computer Security, Dependability, and Assurance: From Needs to Solutions*, IEEE, November 1998.

[4] Orr, A. L., "NASA Denies That Hacking Endangered Shuttle," *Government Computer News*, July 10, 2000.

[5] von Braun, W., Whipple, F. L., and Ley, W., *Conquest of the Moon*, Viking Press, 1953.

[6] Eisner, W., *America's Space Vehicles*, Sterling Publishing, 1962.

[7] Johnson, B., *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley, 1989.

[8] Coulouris, G., and Dollimore, J., *Distributed Systems: Concepts and Design*, Addison-Wesley Publishing, 1988.

[9] Spector, A., and Gifford, D., "The Space Shuttle Primary Computer System," Communications of the ACM, vol 27 No. 9, September 1984.

[10] Herlihy, M. P., and Tygar, J. D., "How to Make Replicated Data Secure," CMU-CS-87-143, August 1987.

[11] Gifford, D. K., "Weighted Voting for Replicated Data," *Proceedings of the Seventh Symposium on Operating Systems Principles*, ACM SIGOPS, December 1979.

[12] Boneh, D., Demillo, R., and Lipton, R., "On the Importance of Checking Cryptographic Protocols for Faults," *Lecture Notes in Computer Science, Advances in Cryptology, Proceedings of EUROCRYPT'97*, pp. 37–51, 1997.

[13] Bao, F., et al. "Breaking Public Key Cryptosystems on Tamper Resistant Devices in the Presence of Transient Faults," *Security Protocol Workshop 1997*, http://www.ens.fr/ vaudenay/spw97/

[14] Amoroso, E., *Fundamentals of Computer Security Technology*, Prentice Hall, 1994.

[15] Losocco. P., Smalley, S., Muckelbauer, P., Taylor, R., Turner, S., and Farrell, J., "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments," *Proceedings of the 21st National Information Systems Security Conference,* October 1998.

[16] Varshney, P. K., *Distributed Detection and Data Fusion*, Springer, 1997.

[17] Pullum, L., "Assessment of the Current State-of-the-Art in Data Diverse Software Fault Tolerance Technology," Rome Laboratory Technical Report, RL-TR-95-15, Vol. 2, February 1995.

[18] Schneider, F. B., "Towards Fault-tolerant and Secure Agentry," *Proceedings of 11th International Workshop of Distributed Algorithms*, September 1997.

[19] Hendler, J., "Unmasking Intelligent Agents," *IEEE Intelligent Systems*, IEEE Computer Society Press, March/April 1999.

[20] Goel, A. L., and Mansour, N., "Software Engineering for Fault-Tolerant Systems," Rome Laboratory Technical Report, RL-TR-91-15, March 1991.

[21] Wensley, J., "SIFT: The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proceedings of the IEEE*, Vol. 66, October 1978, pp. 1240-1255.

[22] Färber, G., "Taskspecific Implementation of Fault Tolerance In Process Automation Systems," *Self-Diagnosis and Fault Tolerance*, M. Dal Cin, E. Dilger (eds), Werkhefte Nr. 4 Attempto Tübingen, 1981.

[23] Ammann, E., Brause, R., Dal Cin, M., Dilger, E., LUTZ, J., Risse, T., "ATTEMPTO: A Fault-Tolerant Multiprocessor Working Station: Design and Concepts," *Digest of Papers FTCS-13*, IEEE Computer Society, 1983, p10-13.

[24] Kieckhafer, R., Walter, C., Finn, A., Thambidurai, P., "The MAFT Architecture for Distributed Fault Tolerance," *IEEE Transactions On Computers*, Vol. 37, No. 4, April 1988, pp. 398-405.

[25] Roda, V., Lin, T., "The Distributed Voting Strategy for Fault Diagnosis and Reconfiguration of Linear Processor Arrays," *Microelectronics Reliability*, Vol. 34, No. 6, June 1994, pp. 955-967.

[26] Gray, J., Reuter, A., *Transaction Processing: Concepts and Techniques,* Morgan Kaufmann, 1993.

[27] Hariri, S., et al., "Architectural Support for Designing Fault-Tolerant Open Distributed Systems," *Computer*, Vol. 25, No. 6, June 1992.

[28] Tanenbaum, Andrew *Computer Networks* Prentice Hall, 1989.

[29] Harper, R. E., Lala, J. H., and Deyst, J. J., "Fault Tolerant Parallel Processor Architecture Overview," *Proceedings of the 18th Fault-Tolerant Computing Symposium,* June, 1988, pp. 252-257.

[30] Palumbo, D. L., Butler, R. W., "A Performance Evaluation of the Software-Implemented Fault-Tolerance Computer," *AIAA Journal of Guidance, Control, and Dynamics,* Vol. 9, No. 2, March-April 1986, pp. 175-180.

[31] Walter, C. J., Kieckhafer, R. M., Finn, A. M., "MAFT: A Multicomputer Architecture for Fault-Tolerance in Real-Time Control Systems," *Proceedings of the IEEE Real Time Systems Symposium,* December 1985.

[32] Goldberg, J., "A History of Research in Fault-Tolerant Computing at SRI International," *The Evolution of Fault-Tolerant Computing*, A. Avizienis, H. Kopetz, J.C. Laprie (eds), Springer-Verlang, Wein, Austria, 1987.

[33] Lamport, L., *et al.*, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, July 1982.

[34] Barborak, M., and Malek, M., "The Consensus Problem in Fault-Tolerant Computing," *ACM Computing Surveys*, Vol. 25, No. 2, June, 1994, pp. 171-220.

[35] Castro, M., Liskov, B. "Practical Byzantine Fault Tolerance", *Proceedings of the Third Symposium on Operating System Design and Implementation*, February 1999.

[36] Reiter, M. "How to Securely Replicate Services," *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 3, May 1994, pp. 986-1009.

[37] Reiter, M., "The Rampart Toolkit for Building High-Integrity Services," *Theory and Practice in Distributed Systems*, Lecture Notes in Computer Science 938, pp. 99-110.

[38] Malkhi, D., Reiter, M., "Byzantine Quorum Systems," *Proceedings of the 29th ACM Symposium on Theory of Computing,* May 1997.

[39] Kihlstrom, K., et al., "The SecureRing Protocols for Securing Group Communication," *Proceedings of the 31st Hawaii International Conference on System Sciences*, Vol. 3, pp. 317-326, January 1998.

[40] Deswarte, Y., et al. "Intrusion Tolerance in Distributed Computing Systems," *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pp. 110-121, May 1991.

[41] Brocklehust, S., Lettlewood, B., Olovsson, T., and Jonsson, E., "On Measurement of Operational Security," *Proceedings of COMPASS*, June 1994.

[42] Jones, D. S., *Elementary Information Theory,* Clarendon Press, 1979.

[43] Schneier, B., *Applied Cryptography*, Second Edition, John Wiley & Sons, 1996.

[44] Cristian, F., "Understanding Fault-Tolerant Distributed Systems," *Communications of the ACM,* Vol. 34, No. 2, February 1991.

[45] Lamport, L., "The Temporal Logic of Actions," *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 3, pp. 872-923, May 1994.

[46] Lamport, L., and Merz S., "Specifying and Verifying Fault-Tolerant Systems," *Proceedings of the Third International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, September, 1994.

[47] Lamport, L., "The Operators of TLA+," *SRC Technical Note 1997-006a*, April 1997.

[48] Abadi, M., and Lamport, L., "An Old-Fashioned Recipe for Real-Time," *ACM Transactions on Programming Languages and Systems*, December 1993.

[49] Hardekopf, B., and Kwiat, K., " Exploiting the Overlap of Security and Fault-Tolerance," *Proceedings of the Academia/Industry Working Conference On Research Challenges (AIWORC)*, April 2000.

[50] Hardekopf, B., and Kwiat K., "Performance Analysis of an Enhanced-Security Distributed Voting Algorithm," *Proceedings of SCS Symposium on Performance of Computer and Telecommunication Systems (SPECTS)*, July 2000.

[51] Hardekopf, B., Kwiat, K., and Upadhyaya, S., " Secure and Fault-Tolerant Voting in Distributed Systems," Proceedings of the IEEE Aerospace Conference, March 2001.

Appendix A

TB-DVA Proof of Partial-Correctness

ASSUME: 1. $RightAnswer \in Answers$

2. $? \notin Answers$

3. $\forall\, i \in Voters \; : \; Safe[i] \in \{\text{TRUE}, \text{FALSE}\}$

4. $Card(\{i \in Voters \; : \; Safe[i]\}) > Card(\{i \in Voters \; : \; \neg Safe[i]\})$

PROVE: $\Phi \Rightarrow \Box(user \in Answers \Rightarrow user = RightAnswer)$

LET: $CV \;\triangleq\; \{cv[j] \; : \; j \in \text{DOMAIN}\; cv\}$

$$
\begin{aligned}
Inv \triangleq\ & 1.\land\ user \in Answers \Rightarrow \land\ user = buffer \\
& \qquad\qquad\qquad\qquad \land\ \forall\, i \in Voters \setminus CV : Safe[i] \Rightarrow \\
& \qquad\qquad\qquad\qquad\quad \land\ analyzed[i] = \text{TRUE} \\
& \qquad\qquad\qquad\qquad\quad \land\ rdy\_commit[i] = \text{FALSE} \\
& 2.\land\ \forall\, i \in \{j \in Voters : Safe[j]\} : analyzed[i] = \text{TRUE} \land rdy\_commit[i] = \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{FALSE} \Rightarrow buffer = RightAnswer \\
& 3.\land\ (\exists\, i \in Voters : analyzed[i] = \text{TRUE}) \Rightarrow \\
& \quad (\forall\, i \in Voters : dissented[i] = \text{TRUE}) \\
& 4.\land\ \forall\, i \in Voters : Safe[i] \land dissented[i] = \text{TRUE} \Rightarrow \forall\, j \in Voters : Safe[j] \Rightarrow \\
& \quad votes[j][i] = RightAnswer \\
& 5.\land\ (\forall\, i \in Voters : analyzed[i] = \text{FALSE}) \land (\exists\, i \in Voters : \\
& \quad Safe[i] \land rcvd\_commit[i] = \text{TRUE}) \Rightarrow \\
& \quad \forall\, i \in Voters : Safe[i] \Rightarrow rcvd\_commit[i] = \text{TRUE} \\
& 6.\land\ buffer = RightAnswer \Rightarrow \forall\, v \in Voters : Safe[v] \Rightarrow \\
& \quad rdy\_commit[v] = \text{FALSE} \\
& 7.\land\ (Card(CV) > 0 \land Safe[Head(cv)]) \Rightarrow buffer = RightAnswer \\
& 8.\land\ (Card(CV) > 1 \land Safe[Head(cv)]) \Rightarrow \neg Safe[Head(Tail(cv))] \\
& 9.\land\ Card(\{i \in Voters \setminus CV : Safe[i]\}) = Card(\{i \in Voters \setminus CV : \\
& \quad \neg Safe[i]\}) \Rightarrow buffer = RightAnswer \\
& 10.\land\ Card(\{i \in Voters \setminus CV : Safe[i]\}) \geq \\
& \quad\ Card(\{i \in Voters \setminus CV : \neg Safe[i]\}) \\
& 11.\land\ Card(Voters \setminus CV) = 0 \Rightarrow buffer = RightAnswer
\end{aligned}
$$

$\langle 1 \rangle 1.\ Init \Rightarrow Inv$

PROOF: By propositional logic, it is sufficient to:

ASSUME: $Init$

PROVE: Inv

$\langle 2\rangle 1.$ 1.$\land$ $user = ?$

2.$\land$ $buffer = ?$

3.$\land$ $analyzed = [\,Voters \mapsto \text{FALSE}]$

4.$\land$ $rcvd\_commit = [\,Voters \mapsto \text{FALSE}]$

5.$\land$ $dissented = [\,Voters \mapsto \text{FALSE}]$

6.$\land$ $cv = \langle\,\rangle$

PROOF: Definition of *Init*.

$\langle 2\rangle 2.$ Q.E.D.

PROOF: $\langle 2\rangle 1$, Assumptions 2, 3, and 4, and definition of *Inv*.

$\langle 1\rangle 2.$ $Inv \land [Next]_{var} \Rightarrow Inv'$

PROOF: By propositional logic, it suffices to:

ASSUME: 1. Inv

2. $[Next]_{var}$

PROVE: Inv'

$\langle 2\rangle 1.$ $Inv.1'$

PROOF: By definition of $Inv.1$ and propositional logic, it suffices to:

ASSUME: $user' \in Answers$

PROVE: $\land\ user' = buffer'$

$\land\ \forall\, i \in Voters \setminus CV' \; : \; Safe[i] \Rightarrow \land\ analyzed[i]' = \text{TRUE}$

$\land\ rdy\_commit[i]' = \text{FALSE}$

$\langle 3\rangle 1.$ CASE: UNCHANGED var

$\langle 4\rangle 1.$ 1.$\land$ $user' = user$

2.$\land$ $buffer' = buffer$

3.$\land$ $cv' = cv$

4.$\land$ $analyzed' = analyzed$

5.$\land$ $rdy\_commit' = rdy\_commit$

PROOF: Definition of UNCHANGED and *var*.

$\langle 4\rangle 2.$ $user \in Answers$

PROOF: $\langle 4\rangle 1.1$ and Assumption $\langle 2\rangle 1$.

$\langle 4 \rangle 3.$ $\wedge$ *user* $=$ *buffer*

$\wedge\ \forall\, i \in \mathit{Voters} \setminus \mathit{CV} \ :\ \mathit{Safe}[i] \Rightarrow \wedge\ \mathit{analyzed}[i] = \text{TRUE}$

$\wedge\ \mathit{rdy\_commit}[i] = \text{FALSE}$

PROOF: $\langle 4 \rangle 2$ and *Inv*.1.

$\langle 4 \rangle 4.$ Q.E.D.

PROOF: $\langle 4 \rangle 1$ and $\langle 4 \rangle 3$.

$\langle 3 \rangle 2.$ CASE: $\exists\, v \in \mathit{Voters} \ :\ \mathit{Commit}(v)$

$\langle 4 \rangle 1.$ CASE: $\mathit{user} \notin \mathit{Answers}$

PROOF: Assumption $\langle 4 \rangle 1$, Assumption $\langle 3 \rangle 2$, and Assumption $\langle 2 \rangle 1$ lead to a contradiction, since $\mathit{Commit}(v) \Rightarrow \text{UNCHANGED } \mathit{user} \Rightarrow \mathit{user}' = \mathit{user}$, by definition of $\mathit{Commit}(v)$ and UNCHANGED .

$\langle 4 \rangle 2.$ CASE: $\mathit{user} \in \mathit{Answers}$

PROOF: Assumption $\langle 4 \rangle 2$ and Assumption $\langle 3 \rangle 2$ lead to a contradiction, since $\mathit{user} \in \mathit{Answers} \Rightarrow \neg\text{ENABLED } \mathit{Commit}(v)$ by definition of $\mathit{Commit}(v)$.

$\langle 4 \rangle 3.$ Q.E.D.

PROOF:$\langle 4 \rangle 1$, $\langle 4 \rangle 2$, and propositional logic.

$\langle 3 \rangle 3.$ CASE: $\exists\, v \in \mathit{Voters} \ :\ \mathit{Dissent}(v)$

$\langle 4 \rangle 1.$ CASE: $\mathit{user} \notin \mathit{Answers}$

PROOF: Assumption $\langle 4 \rangle 1$, Assumption $\langle 3 \rangle 3$, and Assumption $\langle 2 \rangle 1$ lead to a contradiction, since $\mathit{Dissent}(v) \Rightarrow \text{UNCHANGED } \mathit{user} \Rightarrow \mathit{user}' = \mathit{user}$, by definition of $\mathit{Dissent}(v)$ and UNCHANGED .

$\langle 4 \rangle 2.$ CASE: $\mathit{user} \in \mathit{Answers}$

PROOF: Assumption $\langle 4 \rangle 2$ and Assumption $\langle 3 \rangle 3$ lead to a contradiction, since $\mathit{user} \in \mathit{Answers} \Rightarrow \neg\text{ENABLED } \mathit{Dissent}(v)$ by definition of $\mathit{Dissent}(v)$.

$\langle 4 \rangle 3.$ Q.E.D.

PROOF: $\langle 4 \rangle 1$, $\langle 4 \rangle 2$, and propositional logic.

$\langle 3 \rangle 4.$ CASE: $\exists\, v \in \mathit{Voters} \ :\ \mathit{Analyze}(v)$

$\langle 4 \rangle 1.$ CASE: $\mathit{user} \notin \mathit{Answers}$

PROOF: Assumption $\langle 4\rangle 1$, Assumption $\langle 3\rangle 4$, and Assumption $\langle 2\rangle 1$ lead to a contradiction, since $Analyze(v) \Rightarrow$ UNCHANGED $user \Rightarrow user' = user$, by definition of $Analyze(v)$ and UNCHANGED .

$\langle 4\rangle 2$. CASE: $user \in Answers$

PROOF: Assumption $\langle 4\rangle 2$ and Assumption $\langle 3\rangle 4$ lead to a contradiction, since $user \in Answers \Rightarrow \neg$ENABLED $Analyze(v)$ by definition of $Analyze(v)$.

$\langle 4\rangle 3$. Q.E.D.

PROOF: $\langle 4\rangle 1$, $\langle 4\rangle 2$, and propositional logic.

$\langle 3\rangle 5$. CASE: *Terminate*

$\langle 4\rangle 1$. CASE: $user \notin Answers$

$\langle 5\rangle 1$. $user' = buffer$

PROOF: Definition of *Terminate*.

$\langle 5\rangle 2$. $buffer' = buffer$

PROOF: Definition of *Terminate* and UNCHANGED .

$\langle 5\rangle 3$. $user' = buffer'$

PROOF: $\langle 5\rangle 1$, $\langle 5\rangle 2$, and the transitivity of equality.

$\langle 5\rangle 4$. $\forall i \in Voters \setminus CV : Safe[i] \Rightarrow \land\ analyzed[i] = \text{TRUE}$
$\land\ rdy\_commit[i] = \text{FALSE}$

PROOF: Definition of *Terminate* and propositional logic.

$\langle 5\rangle 5$. $\land\ analyzed' = analyzed$
$\land\ rdy\_commit' = rdy\_commit$
$\land\ cv' = cv$

PROOF: Definition of *Terminate* and UNCHANGED .

$\langle 5\rangle 6$. $\forall i \in Voters \setminus CV' : Safe[i] \Rightarrow \land\ analyzed[i]' = \text{TRUE}$
$\land\ rdy\_commit[i]' = \text{FALSE}$

PROOF: $\langle 5\rangle 4$, $\langle 5\rangle 5$, and propositional logic.

$\langle 5\rangle 7$. Q.E.D.

PROOF: $\langle 5\rangle 3$, $\langle 5\rangle 6$, and propositional logic.

$\langle 4\rangle 2$. CASE: $user \in Answers$

PROOF: Assumption $\langle 4\rangle 2$ and Assumption $\langle 3\rangle 5$ lead to a contradiction, since $\mathit{user} \in \mathit{Answers} \Rightarrow \neg$ENABLED *Terminate* by definition of *Terminate*.

$\langle 4\rangle 3$. Q.E.D.

PROOF: $\langle 4\rangle 1$, $\langle 4\rangle 2$, and propositional logic.

$\langle 3\rangle 6$. Q.E.D.

PROOF: $\langle 3\rangle 1$, $\langle 3\rangle 2$, $\langle 3\rangle 3$, $\langle 3\rangle 4$, $\langle 3\rangle 5$, Assumption $\langle 1\rangle 2.2$, and propositional logic.

$\langle 2\rangle 2$. $\mathit{Inv}.2'$

$\langle 3\rangle 1$. CASE: UNCHANGED *var*

$\langle 4\rangle 1$. $\land\ \mathit{analyzed}' = \mathit{analyzed}$
$\land\ \mathit{rdy\_commit}' = \mathit{rdy\_commit}$
$\land\ \mathit{buffer}' = \mathit{buffer}$

PROOF: Definition of UNCHANGED and *var*.

$\langle 4\rangle 2$. Q.E.D.

PROOF: $\langle 4\rangle 1$ and $\mathit{Inv}.2$.

$\langle 3\rangle 2$. CASE: $\exists\, v \in \mathit{Voters} : \mathit{Commit}(v)$

PROOF: $\mathit{Commit}(v)$ implies $\mathit{analyzed}' = [\mathit{Voters} \mapsto \mathit{FALSE}]$ by definition of $\mathit{Commit}(v)$, making $\mathit{Inv}.2'$ true.

$\langle 3\rangle 3$. CASE: $\exists\, v \in \mathit{Voters} : \mathit{Dissent}(v)$

$\langle 4\rangle 1$. $\land\ \mathit{analyzed}' = \mathit{analyzed}$
$\land\ \mathit{rdy\_commit}' = \mathit{rdy\_commit}$
$\land\ \mathit{buffer}' = \mathit{buffer}$

PROOF: Definition of $\mathit{Dissent}(v)$ and UNCHANGED .

$\langle 4\rangle 2$. Q.E.D.

PROOF: $\langle 4\rangle 1$ and $\mathit{Inv}.2$.

$\langle 3\rangle 4$. CASE: $\exists\, v \in \mathit{Voters} : \mathit{Analyze}(v)$

$\langle 4\rangle 1$. CASE: $\mathit{Safe}[v] =$ FALSE

$\langle 5\rangle 1$. $\land\ \mathit{buffer}' = \mathit{buffer}$
$\land\ \forall\, i \in \{j \in \mathit{Voters} : \mathit{Safe}[j]\} : \land\ \mathit{analyzed}[i]' = \mathit{analyzed}[i]$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land\ \mathit{rdy\_commit}[i]' = \mathit{rdy\_commit}[i]$

PROOF: Assumption $\langle 4\rangle 1$ and definition of $Analyze(v)$.

$\langle 5\rangle 2$. Q.E.D.

PROOF: $\langle 5\rangle 1$ and $Inv.2$.

$\langle 4\rangle 2$. CASE: $Safe[v] = \text{TRUE}$

$\langle 5\rangle 1$. $\land\ buffer' = buffer$

$$\land\ \forall\, i \in \{j \in Voters \setminus \{v\} : Safe[j]\} : \land\ analyzed[i]' = analyzed[i]$$
$$\land\ rdy\_commit[i]' = rdy\_commit[i]$$

PROOF: Assumption $\langle 4\rangle 2$ and definition of $Analyze(v)$.

$\langle 5\rangle 2$. $\forall\, i \in Voters : dissented[i] = \text{TRUE}$

PROOF: Definition of $Analyze(v)$.

$\langle 5\rangle 3$. $\forall\, i \in Voters : Safe[i] \Rightarrow votes[v][i] = RightAnswer$

PROOF: $\langle 5\rangle 2$, Assumption $\langle 4\rangle 2$, and $Inv.4$.

$\langle 5\rangle 4$. $maj = RightAnswer$

PROOF: $\langle 5\rangle 3$, Assumptions 3 and 4, and the definition of $Analyze(v)$.

$\langle 5\rangle 5$. $analyzed[v]' = \text{TRUE}$

PROOF: Definition of $Analyze(v)$.

$\langle 5\rangle 6$. $rdy\_commit[v]' = \text{FALSE} \Rightarrow buffer = RightAnswer$

PROOF: $\langle 5\rangle 4$, Assumption $\langle 4\rangle 2$, definition of $Analyze(v)$.

$\langle 5\rangle 7$. Q.E.D.

PROOF: $\langle 5\rangle 1$, $\langle 5\rangle 5$, and $\langle 5\rangle 6$.

$\langle 4\rangle 3$. Q.E.D.

PROOF: $\langle 4\rangle 1$, $\langle 4\rangle 2$, and propositional logic.

$\langle 3\rangle 5$. CASE: $Terminate$

$\langle 4\rangle 1$. $\land\ analyzed' = analyzed$

$\land\ rdy\_commit' = rdy\_commit$

$\land\ buffer' = buffer$

PROOF: Definition of $Terminate$ and UNCHANGED .

$\langle 4\rangle 2$. Q.E.D.

PROOF: $\langle 4\rangle 1$ and $Inv.2$.

$\langle 3\rangle 6$. Q.E.D.

PROOF: $\langle 3\rangle 1$, $\langle 3\rangle 2$, $\langle 3\rangle 3$, $\langle 3\rangle 4$, $\langle 3\rangle 5$, Assumption $\langle 1\rangle 2.2$, and propositional logic.

$\langle 2\rangle 3$. $\mathit{Inv}.3'$

PROOF: By definition of $\mathit{Inv}.3$ and propositional logic, it suffices to:

ASSUME: $\exists\, i \in \mathit{Voters} : \mathit{analyzed}[i]' = \text{TRUE}$

PROVE: $\forall\, i \in \mathit{Voters} : \mathit{dissented}[i]' = \text{TRUE}$

$\langle 3\rangle 1$. CASE: UNCHANGED var

$\langle 4\rangle 1$. 1.$\wedge$ $\mathit{analyzed}' = \mathit{analyzed}$

2.$\wedge$ $\mathit{dissented}' = \mathit{dissented}$

PROOF: Definition of UNCHANGED and var.

$\langle 4\rangle 2$. $\exists\, i \in \mathit{Voters} : \mathit{analyzed}[i] = \text{TRUE}$

PROOF: $\langle 4\rangle 1.1$ and Assumption $\langle 2\rangle 3$.

$\langle 4\rangle 3$. $\forall\, i \in \mathit{Voters} : \mathit{dissented}[i] = \text{TRUE}$

PROOF: $\langle 4\rangle 2$ and $\mathit{Inv}.3$.

$\langle 4\rangle 4$. Q.E.D.

PROOF: $\langle 4\rangle 3$ and $\langle 4\rangle 1.2$.

$\langle 3\rangle 2$. CASE: $\exists\, v \in \mathit{Voters} : \mathit{Commit}(v)$

PROOF: Assumptions $\langle 3\rangle 2$ and $\langle 2\rangle 3$ lead to a contradiction, since $\mathit{Commit}(v)$ implies $\forall\, i \in \mathit{Voters} : \mathit{analyzed}[i]' = \text{FALSE}$ by definition of $\mathit{Commit}(v)$.

$\langle 3\rangle 3$. CASE: $\exists\, v \in \mathit{Voters} : \mathit{Dissent}(v)$

$\langle 4\rangle 1$. $\mathit{analyzed}' = \mathit{analyzed}$

PROOF: Definition of $\mathit{Dissent}(v)$ and UNCHANGED .

$\langle 4\rangle 2$. $\exists\, i \in \mathit{Voters} : \mathit{analyzed}[i] = \text{TRUE}$

PROOF: $\langle 4\rangle 1$ and Assumption $\langle 2\rangle 3$.

$\langle 4\rangle 3$. $\forall\, i \in \mathit{Voters} : \mathit{dissented}[i] = \text{TRUE}$

PROOF: $\langle 4\rangle 2$ and $\mathit{Inv}.3$.

$\langle 4\rangle 4$. Q.E.D.

PROOF: $\langle 4\rangle 3$ and Assumption $\langle 3\rangle 3$ lead to a contradiction, since $\mathit{Dissent}(v)$ implies

$\mathit{dissented}[v] = \text{FALSE}$ by definition of $\mathit{Dissent}(v)$.

⟨3⟩4. CASE: $\exists\, v \in \mathit{Voters} : \mathit{Analyze}(v)$

⟨4⟩1. $\mathit{dissented}' = \mathit{dissented}$

PROOF: Definition of $\mathit{Analyze}(v)$ and UNCHANGED .

⟨4⟩2. $\forall\, i \in \mathit{Voters} : \mathit{dissented}[i] = \text{TRUE}$

PROOF: Definition of $\mathit{Analyze}(v)$.

⟨4⟩3. Q.E.D.

PROOF: ⟨4⟩1 and ⟨4⟩2.

⟨3⟩5. CASE: *Terminate*

⟨4⟩1. 1.$\wedge$ $\mathit{analyzed}' = \mathit{analyzed}$

2.$\wedge$ $\mathit{dissented}' = \mathit{dissented}$

PROOF: Definition of *Terminate* and UNCHANGED .

⟨4⟩2. $\exists\, i \in \mathit{Voters} : \mathit{analyzed}[i] = \text{TRUE}$

PROOF: ⟨4⟩1.1 and Assumption ⟨2⟩3.

⟨4⟩3. $\forall\, i \in \mathit{Voters} : \mathit{dissented}[i] = \text{TRUE}$

PROOF: ⟨4⟩2 and *Inv*.3.

⟨4⟩4. Q.E.D.

PROOF: ⟨4⟩3 and ⟨4⟩1.2.

⟨3⟩6. Q.E.D.

PROOF: ⟨3⟩1, ⟨3⟩2, ⟨3⟩3, ⟨3⟩4, ⟨3⟩5, Assumption ⟨1⟩2.2, and propositional logic.

⟨2⟩4. $\mathit{Inv}.4'$

⟨3⟩1. CASE: UNCHANGED *var*

⟨4⟩1. 1.$\wedge$ $\mathit{dissented}' = \mathit{dissented}$

2.$\wedge$ $\mathit{votes}' = \mathit{votes}$

PROOF: Definition of UNCHANGED and *var*.

⟨4⟩2. Q.E.D.

PROOF: ⟨4⟩1 and *Inv*.4.

⟨3⟩2. CASE: $\exists\, v \in \mathit{Voters} : \mathit{Commit}(v)$

⟨4⟩1. $\forall\, i \in \mathit{Voters} : \mathit{dissented}[i]' = \text{FALSE}$

PROOF: Definition of $Commit(v)$.

$\langle 4 \rangle 2$. Q.E.D.

PROOF: $\langle 4 \rangle 1$, $Inv.4$, and propositional logic.

$\langle 3 \rangle 3$. CASE: $\exists\, v \in Voters : Dissent(v)$

$\langle 4 \rangle 1$. CASE: $Safe[v] = \text{FALSE}$

$\langle 5 \rangle 1$. $\wedge\ \forall\, i \in Voters \setminus \{v\} : dissented[i]' = dissented[i]$

$\wedge\ \forall\, i \in Voters, j \in Voters \setminus \{v\} : Safe[i] \Rightarrow votes[i][j]' = votes[i][j]$

PROOF: Definition of $Dissent(v)$.

$\langle 5 \rangle 2$. Q.E.D.

PROOF: $\langle 5 \rangle 1$, Assumption $\langle 4 \rangle 1$, $Inv.4$, and propositional logic.

$\langle 4 \rangle 2$. CASE: $Safe[v] = \text{TRUE}$

$\langle 5 \rangle 1$. $\wedge\ \forall\, i \in Voters \setminus \{v\} : dissented[i]' = dissented[i]$

$\wedge\ \forall\, i \in Voters, j \in Voters \setminus \{v\} : Safe[i] \Rightarrow votes[i][j]' = votes[i][j]$

PROOF: Definition of $Dissent(v)$.

$\langle 5 \rangle 2$. 1.$\wedge$ $dissented[v] = \text{FALSE}$

2.$\wedge$ $dissented[v]' = \text{TRUE}$

PROOF: Definition of $Dissent(v)$.

$\langle 5 \rangle 3$. $\forall\, i \in Voters : analyzed[i] = \text{FALSE}$

PROOF: $\langle 5 \rangle 2.1$ and $Inv.3$, and propositional logic..

$\langle 5 \rangle 4$. $rcvd\_commit[v] = \text{TRUE}$

PROOF: Assumption $\langle 4 \rangle 2$ and definition of $Dissent(v)$.

$\langle 5 \rangle 5$. $\forall\, i \in Voters : Safe[i] \Rightarrow rcvd\_commit[i] = \text{TRUE}$

PROOF: $\langle 5 \rangle 3$, $\langle 5 \rangle 4$, Assumption $\langle 4 \rangle 2$, $Inv.5$, and propositional logic.

$\langle 5 \rangle 6$. $\forall\, i \in Voters : Safe[i] \Rightarrow votes[i][v]' = RightAnswer$

PROOF: $\langle 5 \rangle 5$, Assumption $\langle 4 \rangle 2$, and definition of $Dissent(v)$.

$\langle 5 \rangle 7$. Q.E.D.

PROOF: $\langle 5 \rangle 6$, $\langle 5 \rangle 2.2$, $\langle 5 \rangle 1$, Assumption $\langle 4 \rangle 2$, and $Inv.4$.

$\langle 4 \rangle 3$. Q.E.D.

PROOF: $\langle 4 \rangle 1$, $\langle 4 \rangle 2$, and propositional logic.

$\langle 3\rangle 4.$ CASE: $\exists\, v \in \mathit{Voters} : \mathit{Analyze}(v)$

$\langle 4\rangle 1.$ $1.\wedge\ \mathit{dissented}' = \mathit{dissented}$

$2.\wedge\ \mathit{votes}' = \mathit{votes}$

PROOF: Definition of $\mathit{Analyze}(v)$ and UNCHANGED .

$\langle 4\rangle 2.$ Q.E.D.

PROOF: $\langle 4\rangle 1$ and $\mathit{Inv}.4$.

$\langle 3\rangle 5.$ CASE: $\mathit{Terminate}$

$\langle 4\rangle 1.$ $1.\wedge\ \mathit{dissented}' = \mathit{dissented}$

$2.\wedge\ \mathit{votes}' = \mathit{votes}$

PROOF: Definition of $\mathit{Terminate}$ and UNCHANGED .

$\langle 4\rangle 2.$ Q.E.D.

PROOF: $\langle 4\rangle 1$ and $\mathit{Inv}.4$.

$\langle 3\rangle 6.$ Q.E.D.

PROOF: $\langle 3\rangle 1$, $\langle 3\rangle 2$, $\langle 3\rangle 3$, $\langle 3\rangle 4$, $\langle 3\rangle 5$, Assumption $\langle 1\rangle 2.2$, and propositional logic.

$\langle 2\rangle 5.$ $\mathit{Inv}.5'$

PROOF: By propositional logic, it suffices to:

ASSUME: 1. $\forall\, i \in \mathit{Voters} : \mathit{analyzed}[i]' = \text{FALSE}$

2. $\exists\, i \in \{j \in \mathit{Voters} : \mathit{Safe}[j]\} : \mathit{rcvd\_commit}[i]' = \text{TRUE}$

PROVE: $\forall\, i \in \{j \in \mathit{Voters} : \mathit{Safe}[j]\} : \mathit{rcvd\_commit}[i]' = \text{TRUE}$

$\langle 3\rangle 1.$ CASE: UNCHANGED var

$\langle 4\rangle 1.$ $\wedge\ \mathit{analyzed}' = \mathit{analyzed}$

$\wedge\ \mathit{rcvd\_commit}' = \mathit{rcvd\_commit}$

PROOF: Definition of UNCHANGED and var.

$\langle 4\rangle 2.$ $\wedge\ \forall\, i \in \mathit{Voters} : \mathit{analyzed}[i] = \text{FALSE}$

$\wedge\ \exists\, i \in \{j \in \mathit{Voters} : \mathit{Safe}[j]\} : \mathit{rcvd\_commit}[i] = \text{TRUE}$

PROOF: $\langle 4\rangle 1$ and Assumption $\langle 2\rangle 5$.

$\langle 4\rangle 3.$ $\forall\, i \in \{j \in \mathit{Voters} : \mathit{Safe}[j]\} : \mathit{rcvd\_commit}[i] = \text{TRUE}$

PROOF: $\langle 4\rangle 2$ and $\mathit{Inv}.5$.

$\langle 4\rangle 4.$ Q.E.D.

PROOF: $\langle 4\rangle 3$ and $\langle 4\rangle 1.2$.

$\langle 3\rangle 2$. CASE: $\exists\, v \in \mathit{Voters} : \mathit{Commit}(v)$

PROOF: Definition of $\mathit{Commit}(v)$.

$\langle 3\rangle 3$. CASE: $\exists\, v \in \mathit{Voters} : \mathit{Dissent}(v)$

$\langle 4\rangle 1$. $\land\ \mathit{analyzed}' = \mathit{analyzed}$

$\land\ \mathit{rcvd\_commit}' = \mathit{rcvd\_commit}$

PROOF: Definition of $\mathit{Dissent}(v)$ and UNCHANGED .

$\langle 4\rangle 2$. $\land\ \forall\, i \in \mathit{Voters} : \mathit{analyzed}[i] = \text{FALSE}$

$\land\ \exists\, i \in \{j \in \mathit{Voters} : \mathit{Safe}[j]\} : \mathit{rcvd\_commit}[i] = \text{TRUE}$

PROOF: $\langle 4\rangle 1$ and Assumption $\langle 2\rangle 5$.

$\langle 4\rangle 3$. $\forall\, i \in \{j \in \mathit{Voters} : \mathit{Safe}[j]\} : \mathit{rcvd\_commit}[i] = \text{TRUE}$

PROOF: $\langle 4\rangle 2$ and *Inv*.5.

$\langle 4\rangle 4$. Q.E.D.

PROOF: $\langle 4\rangle 3$ and $\langle 4\rangle 1.2$.

$\langle 3\rangle 4$. CASE: $\exists\, v \in \mathit{Voters} : \mathit{Analyze}(v)$

PROOF: Assumptions $\langle 3\rangle 4$ and $\langle 2\rangle 5$ lead to a contradiction, since $\mathit{Analyze}(v)$ implies $\mathit{analyzed}[v]' = \text{TRUE}$ by definition of $\mathit{Analyze}(v)$.

$\langle 3\rangle 5$. CASE: *Terminate*

$\langle 4\rangle 1$. $\land\ \mathit{analyzed}' = \mathit{analyzed}$

$\land\ \mathit{rcvd\_commit}' = \mathit{rcvd\_commit}$

PROOF: Definition of *Terminate* and UNCHANGED .

$\langle 4\rangle 2$. $\land\ \forall\, i \in \mathit{Voters} : \mathit{analyzed}[i] = \text{FALSE}$

$\land\ \exists\, i \in \{j \in \mathit{Voters} : \mathit{Safe}[j]\} : \mathit{rcvd\_commit}[i] = \text{TRUE}$

PROOF: $\langle 4\rangle 1$ and Assumption $\langle 2\rangle 5$.

$\langle 4\rangle 3$. $\forall\, i \in \{j \in \mathit{Voters} : \mathit{Safe}[j]\} : \mathit{rcvd\_commit}[i] = \text{TRUE}$

PROOF: $\langle 4\rangle 2$ and *Inv*.5.

$\langle 4\rangle 4$. Q.E.D.

PROOF: $\langle 4\rangle 3$ and $\langle 4\rangle 1.2$.

$\langle 3\rangle 6$. Q.E.D.

PROOF: ⟨3⟩1, ⟨3⟩2, ⟨3⟩3, ⟨3⟩4, ⟨3⟩5, Assumption ⟨1⟩2.2, and propositional logic.

⟨2⟩6. $Inv.6'$

PROOF: By propositional logic, it suffices to:

ASSUME: $buffer' = RightAnswer$

PROVE: $\forall i \in \{j \in Voters : Safe[j]\} : rdy\_commit[i]' = \text{FALSE}$

⟨3⟩1. CASE: UNCHANGED var

⟨4⟩1. 1.$\wedge$ $buffer' = buffer$

2.$\wedge$ $rdy\_commit' = rdy\_commit$

PROOF: Definition of UNCHANGED and var.

⟨4⟩2. $buffer = RightAnswer$

PROOF: ⟨4⟩1.1 and Assumption ⟨2⟩6.

⟨4⟩3. $\forall i \in \{j \in Voters : Safe[j]\} : rdy\_commit[i] = \text{FALSE}$

PROOF: ⟨4⟩2 and $Inv.6$.

⟨4⟩4. Q.E.D.

PROOF: ⟨4⟩3 and ⟨4⟩1.2.

⟨3⟩2. CASE: $\exists v \in Voters : Commit(v)$

PROOF: Definition of $Commit(v)$.

⟨3⟩3. CASE: $\exists v \in Voters : Dissent(v)$

⟨4⟩1. 1.$\wedge$ $buffer' = buffer$

2.$\wedge$ $rdy\_commit' = rdy\_commit$

PROOF: Definition of $Dissent(v)$ and UNCHANGED .

⟨4⟩2. $buffer = RightAnswer$

PROOF: ⟨4⟩1.1 and Assumption ⟨2⟩6.

⟨4⟩3. $\forall i \in \{j \in Voters : Safe[j]\} : rdy\_commit[i] = \text{FALSE}$

PROOF: ⟨4⟩2 and $Inv.6$.

⟨4⟩4. Q.E.D.

PROOF: ⟨4⟩3 and ⟨4⟩1.2.

⟨3⟩4. CASE: $\exists v \in Voters : Analyze(v)$

⟨4⟩1. CASE: $Safe[v] = \text{FALSE}$

⟨5⟩1. $\wedge$ *buffer*$'$ = *buffer*

$\wedge\ \forall\, i \in \{j \in \mathit{Voters} : \mathit{Safe}[j]\} : \mathit{rdy\_commit}[i]' = \mathit{rdy\_commit}[i]$

Proof: Assumption ⟨4⟩1 and definition of $\mathit{Analyze}(v)$.

⟨5⟩2. $\mathit{buffer} = \mathit{RightAnswer}$

Proof: ⟨5⟩1 and Assumption ⟨2⟩6.

⟨5⟩3. $\forall\, i \in \{j \in \mathit{Voters} : \mathit{Safe}[j]\} : \mathit{rdy\_commit}[i] = \mathrm{FALSE}$

Proof: ⟨5⟩2 and *Inv*.6.

⟨5⟩4. Q.E.D.

Proof: ⟨5⟩1 and ⟨5⟩3.

⟨4⟩2. Case: $\mathit{Safe}[v] = \mathrm{TRUE}$

⟨5⟩1. $\wedge$ *buffer*$'$ = *buffer*

$\wedge\ \forall\, i \in \{j \in \mathit{Voters} \setminus \{v\} : \mathit{Safe}[j]\} : \mathit{rdy\_commit}[i]' = \mathit{rdy\_commit}[i]$

Proof: Assumption ⟨4⟩2 and definition of $\mathit{Analyze}(v)$.

⟨5⟩2. $\forall\, i \in \mathit{Voters} : \mathit{dissented}[i] = \mathrm{TRUE}$

Proof: Definition of $\mathit{Analyze}(v)$.

⟨5⟩3. $\forall\, i \in \mathit{Voters} : \mathit{Safe}[i] \Rightarrow \mathit{votes}[v][i] = \mathit{RightAnswer}$

Proof: ⟨5⟩2, Assumption ⟨4⟩2, and *Inv*.4.

⟨5⟩4. $\mathit{maj} = \mathit{RightAnswer}$

Proof: ⟨5⟩3, Assumptions 3 and 4, and the definition of $\mathit{Analyze}(v)$.

⟨5⟩5. $\mathit{buffer} = \mathit{RightAnswer}$

Proof: ⟨5⟩1 and Assumption ⟨2⟩6.

⟨5⟩6. $\mathit{rdy\_commit}[v]' = \mathrm{FALSE}$

Proof: ⟨5⟩4, ⟨5⟩5, and the definition of $\mathit{Analyze}(v)$.

⟨5⟩7. Q.E.D.

Proof: ⟨5⟩1, ⟨5⟩6, and Assumptions ⟨4⟩2 and ⟨2⟩6.

⟨4⟩3. Q.E.D.

Proof: ⟨4⟩1, ⟨4⟩2, and propositional logic.

⟨3⟩5. Case: *Terminate*

$\langle 4 \rangle 1.$ 1.$\wedge$ $\mathit{buffer}' = \mathit{buffer}$

2.$\wedge$ $\mathit{rdy\_commit}' = \mathit{rdy\_commit}$

PROOF: Definition of *Terminate* and UNCHANGED .

$\langle 4 \rangle 2.$ $\mathit{buffer} = \mathit{RightAnswer}$

PROOF: $\langle 4 \rangle 1.1$ and Assumption $\langle 2 \rangle 6$.

$\langle 4 \rangle 3.$ $\forall\, i \in \{j \in \mathit{Voters} : \mathit{Safe}[j]\} : \mathit{rdy\_commit}[i] = \text{FALSE}$

PROOF: $\langle 4 \rangle 2$ and *Inv*.6.

$\langle 4 \rangle 4.$ Q.E.D.

PROOF: $\langle 4 \rangle 3$ and $\langle 4 \rangle 1.2$.

$\langle 3 \rangle 6.$ Q.E.D.

PROOF: $\langle 3 \rangle 1$, $\langle 3 \rangle 2$, $\langle 3 \rangle 3$, $\langle 3 \rangle 4$, $\langle 3 \rangle 5$, Assumption $\langle 1 \rangle 2.2$, and propositional logic.

$\langle 2 \rangle 7.$ $\mathit{Inv}.7'$

PROOF: By propositional logic, it suffices to:

ASSUME: 1. $\mathit{Safe}[\mathit{Head}(cv')] = \text{TRUE}$

2. $\mathit{Card}(CV) > 0$

PROVE: $\mathit{buffer}' = \mathit{RightAnswer}$

$\langle 3 \rangle 1.$ CASE: UNCHANGED *var*

$\langle 4 \rangle 1.$ 1.$\wedge$ $cv' = cv$

2.$\wedge$ $\mathit{buffer}' = \mathit{buffer}$

PROOF: Definition of UNCHANGED and *var*.

$\langle 4 \rangle 2.$ $\mathit{Safe}[\mathit{Head}(cv)] = \text{TRUE}$

PROOF: $\langle 4 \rangle 1.1$ and Assumption $\langle 2 \rangle 7$.

$\langle 4 \rangle 3.$ $\mathit{buffer} = \mathit{RightAnswer}$

PROOF: $\langle 4 \rangle 2$ and *Inv*.7.

$\langle 4 \rangle 4.$ Q.E.D.

PROOF: $\langle 4 \rangle 3$ and $\langle 4 \rangle 1.2$.

$\langle 3 \rangle 2.$ CASE: $\exists\, v \in \mathit{Voters} : \mathit{Commit}(v)$

$\langle 4 \rangle 1.$ $\mathit{Head}(cv') = v$

PROOF: Definition of $\mathit{Commit}(v)$ and *Head*.

$\langle 4 \rangle 2.$ $\mathit{Safe}[v] = \text{TRUE}$

PROOF: $\langle 4 \rangle 1$ and Assumption $\langle 2 \rangle 7$.

$\langle 4 \rangle 3.$ Q.E.D.

PROOF: $\langle 4 \rangle 2$ and definition of $\mathit{Commit}(v)$.

$\langle 3 \rangle 3.$ CASE: $\exists\, v \in \mathit{Voters} : \mathit{Dissent}(v)$

$\langle 4 \rangle 1.$ 1.$\wedge$ $cv' = cv$

2.$\wedge$ $\mathit{buffer}' = \mathit{buffer}$

PROOF: Definition of $\mathit{Dissent}(v)$ and UNCHANGED .

$\langle 4 \rangle 2.$ $\mathit{Safe}[\mathit{Head}(cv)] = \text{TRUE}$

PROOF: $\langle 4 \rangle 1.1$ and Assumption $\langle 2 \rangle 7$.

$\langle 4 \rangle 3.$ $\mathit{buffer} = \mathit{RightAnswer}$

PROOF: $\langle 4 \rangle 2$ and $\mathit{Inv}.7$.

$\langle 4 \rangle 4.$ Q.E.D.

PROOF: $\langle 4 \rangle 3$ and $\langle 4 \rangle 1.2$.

$\langle 3 \rangle 4.$ CASE: $\exists\, v \in \mathit{Voters} : \mathit{Analyze}(v)$

$\langle 4 \rangle 1.$ 1.$\wedge$ $cv' = cv$

2.$\wedge$ $\mathit{buffer}' = \mathit{buffer}$

PROOF: Definition of $\mathit{Analyze}(v)$ and UNCHANGED .

$\langle 4 \rangle 2.$ $\mathit{Safe}[\mathit{Head}(cv)] = \text{TRUE}$

PROOF: $\langle 4 \rangle 1.1$ and Assumption $\langle 2 \rangle 7$.

$\langle 4 \rangle 3.$ $\mathit{buffer} = \mathit{RightAnswer}$

PROOF: $\langle 4 \rangle 2$ and $\mathit{Inv}.7$.

$\langle 4 \rangle 4.$ Q.E.D.

PROOF: $\langle 4 \rangle 3$ and $\langle 4 \rangle 1.2$.

$\langle 3 \rangle 5.$ CASE: $\mathit{Terminate}$

$\langle 4 \rangle 1.$ 1.$\wedge$ $cv' = cv$

2.$\wedge$ $\mathit{buffer}' = \mathit{buffer}$

PROOF: Definition of $\mathit{Terminate}$ and UNCHANGED .

$\langle 4 \rangle 2.$ $\mathit{Safe}[\mathit{Head}(cv)] = \text{TRUE}$

PROOF: $\langle 4\rangle 1.1$ and Assumption $\langle 2\rangle 7$.

$\langle 4\rangle 3.$ $buffer = RightAnswer$

PROOF: $\langle 4\rangle 2$ and $Inv.7$.

$\langle 4\rangle 4.$ Q.E.D.

PROOF: $\langle 4\rangle 3$ and $\langle 4\rangle 1.2$.

$\langle 3\rangle 6.$ Q.E.D.

PROOF: $\langle 3\rangle 1$, $\langle 3\rangle 2$, $\langle 3\rangle 3$, $\langle 3\rangle 4$, $\langle 3\rangle 5$, Assumption $\langle 1\rangle 2.2$, and propositional logic.

$\langle 2\rangle 8.$ $Inv.8'$

PROOF: By propositional logic, it suffices to:

ASSUME: 1. $Safe[Head(cv')] = \text{TRUE}$

2. $Card(CV) > 1$

PROVE: $Safe[Head(Tail(cv'))] = \text{FALSE}$

$\langle 3\rangle 1.$ CASE: UNCHANGED var

$\langle 4\rangle 1.$ $cv' = cv$

PROOF: Definition of UNCHANGED and var.

$\langle 4\rangle 2.$ $Safe[Head(cv)] = \text{TRUE}$

PROOF: $\langle 4\rangle 1$ and Assumption $\langle 2\rangle 8$.

$\langle 4\rangle 3.$ $\neg Safe[Head(Tail(cv))]$

PROOF: $\langle 4\rangle 2$ and $Inv.8$.

$\langle 4\rangle 4.$ Q.E.D.

PROOF: $\langle 4\rangle 3$ and $\langle 4\rangle 1$.

$\langle 3\rangle 2.$ CASE: $\exists\, v \in Voters : Commit(v)$

PROOF: By propositional logic, it suffices to:

ASSUME: $Safe[Head(Tail(cv'))] = \text{TRUE}$

PROVE: a contradiction

$\langle 4\rangle 1.$ $Head(Tail(cv')) = Head(cv)$

PROOF: Definition of $Head$, $Tail$, and $Commit(v)$.

$\langle 4\rangle 2.$ $Safe[Head(cv)] = \text{TRUE}$

PROOF: $\langle 4\rangle 1$ and Assumption $\langle 3\rangle 2$.

$\langle 4\rangle 3$. $buffer = RightAnswer$

PROOF: $\langle 4\rangle 2$ and Inv.7.

$\langle 4\rangle 4$. $Head(cv') = v$

PROOF: Definition of $Head$ and $Commit(v)$.

$\langle 4\rangle 5$. $Safe[v] = \text{TRUE}$

PROOF: $\langle 4\rangle 4$ and Assumption $\langle 2\rangle 8$.

$\langle 4\rangle 6$. $rdy\_commit[v] = \text{FALSE}$

PROOF: $\langle 4\rangle 5$, Inv.6.

$\langle 4\rangle 7$. Q.E.D.

PROOF: $\langle 4\rangle 6$ shows a contradiction, since by definition of $Commit(v)$, $rdy\_commit(v)$ $= \text{TRUE}$. Therefore, Safe[Head(Tail(cv'))] = FALSE.

$\langle 3\rangle 3$. CASE: $\exists\, v \in Voters : Dissent(v)$

$\langle 4\rangle 1$. $cv' = cv$

PROOF: Definition of $Dissent(v)$ and UNCHANGED .

$\langle 4\rangle 2$. $Safe[Head(cv)] = \text{TRUE}$

PROOF: $\langle 4\rangle 1$ and Assumption $\langle 2\rangle 8$.

$\langle 4\rangle 3$. $\neg Safe[Head(Tail(cv))]$

PROOF: $\langle 4\rangle 2$ and Inv.8.

$\langle 4\rangle 4$. Q.E.D.

PROOF: $\langle 4\rangle 3$ and $\langle 4\rangle 1$.

$\langle 3\rangle 4$. CASE: $\exists\, v \in Voters : Analyze(v)$

$\langle 4\rangle 1$. $cv' = cv$

PROOF: Definition of $Analyze(v)$ and UNCHANGED .

$\langle 4\rangle 2$. $Safe[Head(cv)] = \text{TRUE}$

PROOF: $\langle 4\rangle 1$ and Assumption $\langle 2\rangle 8$.

$\langle 4\rangle 3$. $\neg Safe[Head(Tail(cv))]$

PROOF: $\langle 4\rangle 2$ and Inv.8.

$\langle 4\rangle 4$. Q.E.D.

PROOF: $\langle 4\rangle 3$ and $\langle 4\rangle 1$.

$\langle 3\rangle 5$. CASE: *Terminate*

$\langle 4\rangle 1$. $cv' = cv$

PROOF: Definition of *Terminate* and UNCHANGED .

$\langle 4\rangle 2$. $Safe[Head(cv)] = \text{TRUE}$

PROOF: $\langle 4\rangle 1$ and Assumption $\langle 2\rangle 8$.

$\langle 4\rangle 3$. $\neg Safe[Head(Tail(cv))]$

PROOF: $\langle 4\rangle 2$ and *Inv*.8.

$\langle 4\rangle 4$. Q.E.D.

PROOF: $\langle 4\rangle 3$ and $\langle 4\rangle 1$.

$\langle 3\rangle 6$. Q.E.D.

PROOF: $\langle 3\rangle 1$, $\langle 3\rangle 2$, $\langle 3\rangle 3$, $\langle 3\rangle 4$, $\langle 3\rangle 5$, Assumption $\langle 1\rangle 2.2$, and propositional logic.

$\langle 2\rangle 9$. *Inv*.9$'$

PROOF: By propositional logic, it suffices to:

ASSUME: $Card(\{i \in Voters \setminus CV' : Safe[i]\}) = Card(\{i \in Voters \setminus CV' : \neg Safe[i]\})$

PROVE: $buffer' = RightAnswer$

$\langle 3\rangle 1$. CASE: UNCHANGED *var*

$\langle 4\rangle 1$. 1.$\wedge$ $cv' = cv$

2.$\wedge$ $buffer' = buffer$

PROOF: Definition of UNCHANGED and *var*.

$\langle 4\rangle 2$. $Card(\{i \in Voters \setminus CV : Safe[i]\}) = Card(\{i \in Voters \setminus CV : \neg Safe[i]\})$

PROOF: $\langle 4\rangle 1.1$ and Assumption $\langle 2\rangle 9$.

$\langle 4\rangle 3$. $buffer = RightAnswer$

PROOF: $\langle 4\rangle 2$ and *Inv*.9.

$\langle 4\rangle 4$. Q.E.D.

PROOF: $\langle 4\rangle 3$ and $\langle 4\rangle 1.2$.

$\langle 3\rangle 2$. CASE: $\exists\, v \in Voters : Commit(v)$

⟨4⟩1. CASE: $Card(\{i \in Voters \setminus CV : Safe[i]\}) = Card(\{i \in Voters \setminus CV : \neg Safe[i]\}) + 1$

⟨5⟩1. $Safe[v] = \text{TRUE}$

PROOF: Assumptions ⟨4⟩1, ⟨2⟩9, and definitions of *Card* and $Commit(v)$.

⟨5⟩2. Q.E.D.

PROOF: ⟨5⟩1 and definition of $Commit(v)$.

⟨4⟩2. CASE: $Card(\{i \in Voters \setminus CV : Safe[i]\}) + 1 = Card(\{i \in Voters \setminus CV : \neg Safe[i]\})$

⟨5⟩1. $Card(\{i \in CV : Safe[i]\}) \geq Card(\{i \in CV : \neg Safe[i]\}) + 2$

PROOF: Assumptions ⟨4⟩2, 3, and 4.

⟨5⟩2. Q.E.D.

PROOF: Contradiction: a simple counting argument shows that for ⟨5⟩1 to be true, 2 trustworthy voters committed in sequence (i.e. $\exists\, i, j \in Voters; n, m \in Naturals : Safe[i] \wedge Safe[j] \wedge n = m + 1 \wedge cv[n] = i \wedge cv[m] = j$). But *Inv*.8 states that no two trustworthy voters can commit in sequence (i.e. $\Box((Card(CV) > 1 \wedge Safe[Head(cv)]) \Rightarrow \neg Safe[Head(Tail(cv))]]$).

⟨4⟩3. Q.E.D.

PROOF: ⟨4⟩1, ⟨4⟩2, definitions of *Card* and $Commit(v)$, and propositional logic.

⟨3⟩3. CASE: $\exists\, v \in Voters : Dissent(v)$

⟨4⟩1. 1.$\wedge$ $cv' = cv$

2.$\wedge$ $buffer' = buffer$

PROOF: Definition of $Dissent(v)$ and UNCHANGED .

⟨4⟩2. $Card(\{i \in Voters \setminus CV : Safe[i]\}) = Card(\{i \in Voters \setminus CV : \neg Safe[i]\})$

PROOF: ⟨4⟩1.1 and Assumption ⟨2⟩9.

⟨4⟩3. $buffer = RightAnswer$

PROOF: ⟨4⟩2 and *Inv*.9.

⟨4⟩4. Q.E.D.

PROOF: ⟨4⟩3 and ⟨4⟩1.2.

$\langle 3\rangle 4.$ CASE: $\exists\, v \in \mathit{Voters} : \mathit{Analyze}(v)$

$\langle 4\rangle 1.$ 1.$\wedge$ $cv' = cv$

2.$\wedge$ $\mathit{buffer}' = \mathit{buffer}$

PROOF: Definition of $\mathit{Analyze}(v)$ and UNCHANGED .

$\langle 4\rangle 2.$ $\mathit{Card}(\{i \in \mathit{Voters} \setminus CV : \mathit{Safe}[i]\}) = \mathit{Card}(\{i \in \mathit{Voters} \setminus CV : \neg \mathit{Safe}[i]\})$

PROOF: $\langle 4\rangle 1.1$ and Assumption $\langle 2\rangle 9$.

$\langle 4\rangle 3.$ $\mathit{buffer} = \mathit{RightAnswer}$

PROOF: $\langle 4\rangle 2$ and *Inv*.9.

$\langle 4\rangle 4.$ Q.E.D.

PROOF: $\langle 4\rangle 3$ and $\langle 4\rangle 1.2$.

$\langle 3\rangle 5.$ CASE: *Terminate*

$\langle 4\rangle 1.$ 1.$\wedge$ $cv' = cv$

2.$\wedge$ $\mathit{buffer}' = \mathit{buffer}$

PROOF: Definition of *Terminate* and UNCHANGED .

$\langle 4\rangle 2.$ $\mathit{Card}(\{i \in \mathit{Voters} \setminus CV : \mathit{Safe}[i]\}) = \mathit{Card}(\{i \in \mathit{Voters} \setminus CV : \neg \mathit{Safe}[i]\})$

PROOF: $\langle 4\rangle 1.1$ and Assumption $\langle 2\rangle 9$.

$\langle 4\rangle 3.$ $\mathit{buffer} = \mathit{RightAnswer}$

PROOF: $\langle 4\rangle 2$ and *Inv*.9.

$\langle 4\rangle 4.$ Q.E.D.

PROOF: $\langle 4\rangle 3$ and $\langle 4\rangle 1.2$.

$\langle 3\rangle 6.$ Q.E.D.

PROOF: $\langle 3\rangle 1$, $\langle 3\rangle 2$, $\langle 3\rangle 3$, $\langle 3\rangle 4$, $\langle 3\rangle 5$, Assumption $\langle 1\rangle 2.2$, and propositional logic.

$\langle 2\rangle 10.$ $\mathit{Inv}.10'$

$\langle 3\rangle 1.$ CASE: UNCHANGED *var*

$\langle 4\rangle 1.$ $cv' = cv$

PROOF: Definition of UNCHANGED and *var*.

$\langle 4\rangle 2.$ Q.E.D.

PROOF: $\langle 4\rangle 1$ and *Inv*.10.

$\langle 3\rangle 2.$ CASE: $\exists\, v \in \mathit{Voters} : \mathit{Commit}(v)$

$\langle 4\rangle 1$. CASE: $Safe[v] = \text{TRUE}$

$\langle 5\rangle 1$. CASE: $Card(\{i \in Voters \setminus CV : Safe[i]\}) > Card(\{i \in Voters \setminus CV : \neg Safe[i]\})$

$\langle 6\rangle 1$. $\wedge\ Card(\{i \in Voters \setminus CV' : Safe[i]\}) = Card(\{i \in Voters \setminus CV : Safe[i]\}) + 1$

$\wedge\ Card(\{i \in Voters \setminus CV' : \neg Safe[i]\}) = Card(\{i \in Voters \setminus CV : \neg Safe[i]\})$

PROOF: Assumption $\langle 4\rangle 1$, definition of *Card* and $Commit(v)$.

$\langle 6\rangle 2$. Q.E.D.

PROOF: $\langle 6\rangle 1$, Assumption $\langle 5\rangle 1$, definition of *Card*.

$\langle 5\rangle 2$. CASE: $Card(\{i \in Voters \setminus CV : Safe[i]\}) = Card(\{i \in Voters \setminus CV : \neg Safe[i]\})$

$\langle 6\rangle 1$. $buffer = RightAnswer$

PROOF: Assumption $\langle 5\rangle 2$ and *Inv*.9.

$\langle 6\rangle 2$. $rdy\_commit[v] = \text{FALSE}$

PROOF: $\langle 6\rangle 1$ and *Inv*.6.

$\langle 6\rangle 3$. Q.E.D.

PROOF: Assumptions $\langle 5\rangle 2$ and $\langle 4\rangle 1$ lead to a contradiction as shown by $\langle 6\rangle 2$, since $Commit(v)$ implies $rdy\_commit[v] = \text{TRUE}$ by definition of $Commit(v)$.

$\langle 5\rangle 3$. Q.E.D.

PROOF: $\langle 5\rangle 1$, $\langle 5\rangle 2$, and propositional logic.

$\langle 4\rangle 2$. CASE: $Safe[v] = \text{FALSE}$

$\langle 5\rangle 1$. $\wedge\ Card(\{i \in Voters \setminus CV' : \neg Safe[i]\}) < Card(\{i \in Voters \setminus CV : \neg Safe[i]\})$

$\wedge\ Card(\{i \in Voters \setminus CV' : Safe[i]\}) = Card(\{i \in Voters \setminus CV : Safe[i]\})$

PROOF: Assumption $\langle 4\rangle 2$, definition of *Card* and $Commit(v)$.

$\langle 5\rangle 2$. Q.E.D.

PROOF: $\langle 5\rangle 1$ and *Inv*.10.

$\langle 4\rangle 3$. Q.E.D.

PROOF: $\langle 4\rangle 1$, $\langle 4\rangle 2$, and propositional logic.

$\langle 3 \rangle 3$. CASE: $\exists\, v \in \mathit{Voters} : \mathit{Dissent}(v)$

$\langle 4 \rangle 1$. $cv' = cv$

PROOF: Definition of $\mathit{Dissent}(v)$ and UNCHANGED .

$\langle 4 \rangle 2$. Q.E.D.

PROOF: $\langle 4 \rangle 1$ and $\mathit{Inv}.10$.

$\langle 3 \rangle 4$. CASE: $\exists\, v \in \mathit{Voters} : \mathit{Analyze}(v)$

$\langle 4 \rangle 1$. $cv' = cv$

PROOF: Definition of $\mathit{Analyze}(v)$ and UNCHANGED .

$\langle 4 \rangle 2$. Q.E.D.

PROOF: $\langle 4 \rangle 1$ and $\mathit{Inv}.10$.

$\langle 3 \rangle 5$. CASE: $\mathit{Terminate}$

$\langle 4 \rangle 1$. $cv' = cv$

PROOF: Definition of $\mathit{Terminate}$ and UNCHANGED .

$\langle 4 \rangle 2$. Q.E.D.

PROOF: $\langle 4 \rangle 1$ and $\mathit{Inv}.10$.

$\langle 3 \rangle 6$. Q.E.D.

PROOF: $\langle 3 \rangle 1$, $\langle 3 \rangle 2$, $\langle 3 \rangle 3$, $\langle 3 \rangle 4$, $\langle 3 \rangle 5$, Assumption $\langle 1 \rangle 2.2$, and propositional logic.

$\langle 2 \rangle 11$. $\mathit{Inv}.11'$

PROOF: By propositional logic, it suffices to:

ASSUME: $\mathit{Card}(\mathit{Voters} \setminus CV') = 0$

PROVE: $\mathit{buffer}' = \mathit{RightAnswer}$

$\langle 3 \rangle 1$. CASE: UNCHANGED var

$\langle 4 \rangle 1$. $1.\wedge\ cv' = cv$

$2.\wedge\ \mathit{buffer}' = \mathit{buffer}$

PROOF: Definition of UNCHANGED and var.

$\langle 4 \rangle 2$. $\mathit{Card}(\mathit{Voters} \setminus CV) = 0$

PROOF: $\langle 4 \rangle 1.1$ and Assumption $\langle 2 \rangle 11$.

$\langle 4 \rangle 3$. $\mathit{buffer} = \mathit{RightAnswer}$

PROOF: $\langle 4\rangle 2$ and *Inv*.11.

$\langle 4\rangle 4$. Q.E.D.

PROOF: $\langle 4\rangle 3$ and $\langle 4\rangle 1.2$.

$\langle 3\rangle 2$. CASE: $\exists\, v \in \mathit{Voters} : \mathit{Commit}(v)$

$\langle 4\rangle 1$. $\mathit{Card}(\mathit{Voters} \setminus \mathit{CV}) = 1$

PROOF: Assumption $\langle 2\rangle 11$, definition of *Card* and $\mathit{Commit}(v)$.

$\langle 4\rangle 2$. $\{v\} = \mathit{Voters} \setminus \mathit{CV}$

PROOF: $\langle 4\rangle 1$, Assumption $\langle 3\rangle 2$, and definition of *Card*.

$\langle 4\rangle 3$. $\mathit{Safe}[v] = \text{TRUE}$

PROOF: $\langle 4\rangle 1$, $\langle 4\rangle 2$, *Inv*.10.

$\langle 4\rangle 4$. Q.E.D.

PROOF: $\langle 4\rangle 3$ and definition of $\mathit{Commit}(v)$.

$\langle 3\rangle 3$. CASE: $\exists\, v \in \mathit{Voters} : \mathit{Dissent}(v)$

$\langle 4\rangle 1$. $\land\ cv' = cv$

$\land\ \mathit{buffer}' = \mathit{buffer}$

PROOF: Definition of $\mathit{Dissent}(v)$ and UNCHANGED .

$\langle 4\rangle 2$. $\mathit{Card}(\mathit{Voters} \setminus \mathit{CV}) = 0$

PROOF: $\langle 4\rangle 1.1$ and Assumption $\langle 2\rangle 11$.

$\langle 4\rangle 3$. $\mathit{buffer} = \mathit{RightAnswer}$

PROOF: $\langle 4\rangle 2$ and *Inv*.11.

$\langle 4\rangle 4$. Q.E.D.

PROOF: $\langle 4\rangle 3$ and $\langle 4\rangle 1.2$.

$\langle 3\rangle 4$. CASE: $\exists\, v \in \mathit{Voters} : \mathit{Analyze}(v)$

$\langle 4\rangle 1$. $\land\ cv' = cv$

$\land\ \mathit{buffer}' = \mathit{buffer}$

PROOF: Definition of $\mathit{Analyze}(v)$ and UNCHANGED .

$\langle 4\rangle 2$. $\mathit{Card}(\mathit{Voters} \setminus \mathit{CV}) = 0$

PROOF: $\langle 4\rangle 1.1$ and Assumption $\langle 2\rangle 11$.

$\langle 4\rangle 3$. $\mathit{buffer} = \mathit{RightAnswer}$

PROOF: $\langle 4\rangle 2$ and $Inv.11$.

$\langle 4\rangle 4$. Q.E.D.

PROOF: $\langle 4\rangle 3$ and $\langle 4\rangle 1.2$.

$\langle 3\rangle 5$. CASE: $Terminate$

$\langle 4\rangle 1$. $\land\ cv' = cv$

$\land\ buffer' = buffer$

PROOF: Definition of $Terminate$ and UNCHANGED .

$\langle 4\rangle 2$. $Card(Voters \setminus CV) = 0$

PROOF: $\langle 4\rangle 1.1$ and Assumption $\langle 2\rangle 11$.

$\langle 4\rangle 3$. $buffer = RightAnswer$

PROOF: $\langle 4\rangle 2$ and $Inv.11$.

$\langle 4\rangle 4$. Q.E.D.

PROOF: $\langle 4\rangle 3$ and $\langle 4\rangle 1.2$.

$\langle 3\rangle 6$. Q.E.D.

PROOF: $\langle 3\rangle 1$, $\langle 3\rangle 2$, $\langle 3\rangle 3$, $\langle 3\rangle 4$, $\langle 3\rangle 5$, Assumption $\langle 1\rangle 2.2$, and propositional logic.

$\langle 2\rangle 12$. Q.E.D.

PROOF: $\langle 2\rangle 1$, $\langle 2\rangle 2$, $\langle 2\rangle 3$, $\langle 2\rangle 4$, $\langle 2\rangle 5$, $\langle 2\rangle 6$, $\langle 2\rangle 7$, $\langle 2\rangle 8$, $\langle 2\rangle 9$, $\langle 2\rangle 10$, $\langle 2\rangle 11$, and the definition of Inv.

$\langle 1\rangle 3$. $Inv \Rightarrow (user \in Answers \Rightarrow user = RightAnswer)$

PROOF: By propositional logic, it suffices to:

ASSUME: 1. Inv

2. $user \in Answers$

PROVE: $user = RightAnswer$

$\langle 2\rangle 1$. CASE: $CV \subset Voters$

$\langle 3\rangle 1$. $user = buffer$

PROOF: $Inv.1$, Assumption $\langle 1\rangle 3.2$, and propositional logic.

$\langle 3\rangle 2$. $\forall\, i \in Voters \setminus CV\ :\ Safe[i] \Rightarrow \land\ analyzed[i] = \text{TRUE}$

$\land\ rdy\_commit[i] = \text{FALSE}$

PROOF: $Inv.1$, Assumption $\langle 1\rangle 3.2$, and propositional logic.

$\langle 3\rangle 3.\ \exists\, i \in \mathit{Voters} \setminus \mathit{CV} : \mathit{Safe}[i]$

PROOF: Assumption $\langle 2\rangle 1$, $\mathit{Inv}.10$.

$\langle 3\rangle 4.\ \mathit{buffer} = \mathit{RightAnswer}$

PROOF: $\mathit{Inv}.2$, $\langle 3\rangle 3$, $\langle 3\rangle 2$, and propositional logic.

$\langle 3\rangle 5.$ Q.E.D.

PROOF: $\langle 3\rangle 1$, $\langle 3\rangle 4$, Assumption 1, and the transitivity of equality.

$\langle 2\rangle 2.$ CASE: $\mathit{CV} = \mathit{Voters}$

$\langle 3\rangle 1.\ \mathit{Card}(\mathit{Voters} \setminus \mathit{CV}) = 0$

PROOF: Assumption $\langle 2\rangle 2$ and definition of Card.

$\langle 3\rangle 2.\ \mathit{buffer} = \mathit{RightAnswer}$

PROOF: $\langle 3\rangle 1$ and $\mathit{Inv}.11$.

$\langle 3\rangle 3.\ \mathit{user} = \mathit{buffer}$

PROOF: Assumption $\langle 1\rangle 3.2$ and $\mathit{Inv}.2$.

$\langle 3\rangle 4.$ Q.E.D.

PROOF: $\langle 3\rangle 3$, $\langle 3\rangle 2$, and the transitivity of equality.

$\langle 2\rangle 3.$ Q.E.D.

PROOF: $\langle 2\rangle 1$, $\langle 2\rangle 2$, and propositional logic.

$\langle 1\rangle 4.$ Q.E.D.

PROOF:
$$\begin{aligned}
\Phi \ &\Rightarrow \{\text{By definition of } \Phi\}\\
&\quad \mathit{Init} \wedge \Box[\mathit{Next}]_{\mathit{var}}\\
&\Rightarrow \{\langle 1\rangle 1\}\\
&\quad \mathit{Inv} \wedge \Box[\mathit{Next}]_{\mathit{var}}\\
&\Rightarrow \{\langle 1\rangle 2 \text{ and TLA Rule INV1}\}\\
&\quad \Box \mathit{Inv}\\
&\Rightarrow \{\langle 1\rangle 3\}\\
&\quad \Box(\mathit{user} \in \mathit{Answers} \Rightarrow \mathit{user} = \mathit{RightAnswer})
\end{aligned}$$

Appendix B

TB-DVA Proof of Termination

ASSUME: 1. $\forall\, i \in Voters : Safe[i] \in \{\text{TRUE}, \text{FALSE}\}$

2. $Card(\{i \in Voters : Safe[i]\}) > Card(\{i \in Voters : \neg Safe[i]\})$

PROVE: Algorithm TB−DVA terminates

LET: $t \triangleq Card(Voters) - Card(CV)$

$\langle 1 \rangle 1.$ After $Init$, $t > 0$

PROOF: By Assumptions 1 and 2, $Card(Voters) > 0$. By definition of $Init$, $Card(CV) = 0$. Therefore, $Card(Voters) - Card(CV) > 0$.

$\langle 1 \rangle 2.$ $t = 0 \Rightarrow TB{-}DVA\ terminates$

PROOF: If $t = 0$, then $Card(Voters) - Card(CV) = 0$ (by definition of t). Therefore, $A = \text{FALSE}$ and $B = \text{TRUE}$, by definition of A [1] and B [2] and propositional logic. As evidenced by the flowchart in Figure 3.1, the loop will exit and TB-DVA will terminate.

$\langle 1 \rangle 3.$ After each $Commit$, $t' = t - 1$

PROOF: By definition of $Commit(v)$, $CV' = CV \cup \{v\}$. Also by definition of $Commit(v)$, $v \notin CV$. Therefore, $Card(CV') = Card(CV) + 1$ and $Card(Voters) - Card(CV') = Card(Voters) - Card(CV) - 1$.

$\langle 1 \rangle 4.$ Q.E.D.

[1] $\exists\, v \in Voters \setminus CV : rdy\_commit[v] = \text{TRUE}$

[2] $\forall\, v \in Voters \setminus CV : rdy\_commit[v] = \text{FALSE}$

PROOF: $\langle 1\rangle 1$, $\langle 1\rangle 2$, $\langle 1\rangle 3$, and propositional logic.

☆U.S. GOVERNMENT PRINTING OFFICE: 2001-610-055-10088

MISSION OF AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.