An N-version Electronic Voting System.

by

Soyini D. Liburd

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science.

at the Massachusetts Institute of Technology

May 20, 2004

Author _____
Department of Electrical Engineering and Computer Science
May 20, 2004

Certified by _____
Ted Selker, Associate Professor, Program in Media Arts and Sciences
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

An N-version Electronic Voting System.

by

Soyini D. Liburd

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science.

at the
Massachusetts Institute of Technology
May 20, 2004

Abstract

The ballot battles of the 2000 US Presidential Election clearly indicate that existing voting technologies and processes are not sufficient to guarantee that every eligible voter is granted their right to vote and implicitly to have that vote counted, as per the fifteenth, nineteenth, twenty fourth and twenty sixth amendments to the US constitution [1-3]. Developing a voting system that is secure, correct, reliable and trustworthy is a significant challenge to current technology [3, 4]. The Secure Architecture for Voting Electronically (SAVE) demonstrates that N-version programming increases the reliability and security of its systems, and can be used to increase the trustworthiness of systems. Further, SAVE demonstrates how a viable practical approach to voting can be created using N-version programming. SAVE represents a significant contribution to voting technology research because of its design, and also because it demonstrates the benefits of N-version programming and introduces these benefits to the field of voting technology.

Thesis Supervisor: Ted Selker
Title: Associate Professor of Media Arts and Sciences

Acknowledgements

I would like to especially thank Professor Ted Selker for proposing the application of N-version programming to voting and for his invaluable contributions and support throughout the very short and very rewarding semester that I worked on this project.

I sincerely thank Professor Stephen Graves and Professor Ron Rivest for their generous assistance and availability throughout the work on this project. Your suggestions and advice have definitely helped improve the results of this project thus far.

Special thanks to the students, Shawn Sullivan, Arturo Hinojosa, David Chau and Kevin Emery, who helped me implement a prototype for our N-version system. I am glad you are enjoying this project and I look forward to continuing to work with you all.

Lastly, I would like to extend heartfelt thanks to my mother, Yolanda, my sister, Sekayi and my brother, Lasana, as well as to all my friends, for their continued encouragement and support. Your round-the-clock cheers and proddings were quite effective, for as you can see, I am here.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Although, voting systems and protocols have improved since their inception, more must be done to improve their accuracy, reliability, efficiency and security, as well as accessibility and trustworthiness [3, 5, 6]. Paper ballots are subject to loss [5] and may be corrupted in various ways, including accidental or malicious overvoting [7]. Additionally, paper ballots must be securely transported and counted, activities which tend to make the election process slow, labor intensive and costly [3]. Paper ballots also have significant usability limitations which make them generally less accessible to voters who are not comfortable with the languages available at the polling station, voters who are illiterate and voters who are vision impaired or have other disabilities [3, 5]. Many such voters require assistance which compromises their privacy and their trust that their vote was cast as intended [5]. Electronic voting systems have been proposed to address these issues, but they have produced their own problems [8-12, 13-14]. Security flaws, correctness problems and other vulnerabilities in existing electronic voting systems have resulted in flawed elections, where potential votes were refused, and actual votes were discarded or incorrectly counted [8-12, 13]. Electronic voting systems have also suffered from reliability and availability problems [14]. These problems have reduced public acceptance and trust in electronic voting systems [8-12, 13-14].

This thesis develops a secure, reliable, accessible and trustworthy voting system that addresses the problems found in current voting systems. In this chapter, we motivate the use of electronic voting systems and introduce electronic voting criteria that should be satisfied in order to develop a secure, reliable and trustworthy voting system. This chapter then outlines how an N-version voting system, in particular, could satisfy these voting criteria. Chapter 2 provides detailed motivation for our use of N-version programming, including the plausibility and effectiveness of N-version programming as a means of improving reliability. It describes the fundamental concepts in N-version programming and defends the credibility of N-version programming by examining N-version research and results. Chapter 2 concludes by motivating and explaining the N-

version implementation choices made in SAVE and describing SAVE as an N-version structure. Chapter 3 describes the SAVE voting system in detail, including the architecture, voting process and system assumptions made, and chapter 4 analyzes this system. Chapter 5 describes details regarding the implementation of our SAVE prototype, Chapter 6 explains the issues and limitations of the current SAVE system and also discusses plans for future work on the SAVE system. Specially, chapter 6 notes some lessons that would be helpful for future N-version system designers. This thesis concludes in chapter 7 with SAVE's contribution to voting systems research and the voting systems community in general.

1.1 Motivation for Electronic Voting

Recent problems with electronic voting machines have caused a fresh wave of panic and uncertainty about whether electronic voting can indeed improve on paper systems [15, 16]. It is critical to remember however that the problems that occurred in the California Primary Election in March 2004 are representative of a few voting schemes and implementation, not of electronic voting as a whole. We also emphasize that reverting to paper systems is not the long term solution to our voting woes. There are significant problems with paper voting as catastrophically proven in the 2000 Presidential Election, and electronic voting can solve most of these problems [3]. In this section, we briefly recall the problems with paper voting that arose in the 2000 Presidential Election and describe how electronic voting systems can address them. In the next section however, we will address electronic voting concerns by listing criteria for electronic voting systems that address popular worries and explaining how an N-version electronic voting system can counter concerns by satisfying the criteria.

1.1.1 Problems with Paper Voting

The paper voting systems currently used in the United States are: full paper ballot schemes, punch card systems and optical scan systems [3]. All of these schemes face numerous problems that together have disenfranchised millions of voters [3].

Administration Difficulties

All paper ballot schemes face administrative nightmares in order to get the right ballots to the right locations, including the right proportions of ballots in different languages or with other distinguishing features [3]. Further, the paper ballots required by these schemes are expensive to print, secure and distribute correctly [3]. Full paper ballot systems and punch cards face the additional task of securing the ballots after they have been marked and before they can be counted; ballot boxes must be securely stored and transported and subsequent ballot box opening and ballot counting must be carefully monitored [3]. We believe that all of this costs significantly more time and money than would be the case with electronic systems. Further, the mundane, repetitive task of hand-counting is relatively slow, cumbersome, labor-intensive, inefficient and error-prone [3]. Hand-counting the millions of ballots generated in a US presidential election would be quite infeasible, and thus cries to go completely back to full paper ballot systems are unreasonable in practice.

User Interface Problems

User Interface problems are again common to all paper ballot schemes [3]. In many cases paper voting system user interfaces allow voters to make mistakes that ruin their ballot [3]. For example, it is estimated that 1.5 million presidential votes are lost each election and 3.5 million votes for governor and senator are lost each cycle, due to undervoting or overvoting [3]. Undervotes have been known to occur quite frequently in paper systems, where, for example, a circle or arrow is not sufficiently filled in in a full paper ballot scheme, or a punch card machine only dimples the ballot instead of completing the punch in a punch card system [3]. Overvoting occurs in many cases because of stray marks or dimpling at multiple indicators or holes corresponding to candidates [3]. Additionally, in some cases like the famed "butterfly ballots," the ballot layout is just so confusing that a voter completely misrepresents his intentions, by actually voting for candidate A, say, when he thinks he has voted for candidate B [3]. Falloff is another user-interface related problem in voting. Falloff is the name given to the phenomena that candidates near the bottom of a ballot are less likely to be selected than candidates at the top. It is very difficult for paper ballot systems to compensate for

falloff. Attempts to negate the effects of falloff on paper ballots exist, for example, in a few places voting officials rotate the placement of names on the ballot. However, this fix requires the printing of many instances of the same ballot with different name rotations, and this increases the cost of supplying ballots as well as the administrative difficulties associated with transporting and distributing ballots.

Accessibility

Paper voting systems are not accessible to many voters with special needs [5]. Fonts are generally small and ballots are generally crowded with names, for example, and these and other issues make voting difficult for vision impaired voters or voters with some learning disabilities [5, 85]. Further all current paper voting systems require some sort of motor control, which makes secret voting impossible for many paralyzed citizens [5]. Because of these and other user interface problems in paper systems, it is said that approximately 16.4 million disabled and 37 million illiterate American voters are unable to vote in privacy, and millions of other persons with less severe impairments find voting extremely difficult [5].

System Problems

Punch card and optical scanner systems have also had structural problems that have resulted in lost votes [3]. Poorly aligned ballots in punch card systems have caused many votes to be lost as voter punches make holes between candidates, for example [17]. Additionally, punch card systems are prone to unreliable and inconsistent counts, as dimples snag or chads fall out, after multiple passes through the reader [3]. Punched ballots are not guaranteed to be true representatives of the voter's intention or even of what the voter cast, as the ballot is not maintained in its original state.

1.1.2 The Electronic Voting Solution

Modern electronic voting systems can solve the problems with paper systems highlighted in the previous section. The ballots are electronic and so this removes the issues and frustrations with paper administration. User interface and accessibility problems can be solved by flexible font size, coloring and other details, as well as multi media interfaces

and special equipment to translate the more limited signals of paralyzed voters [5]. Disabled persons report significantly preferring electronic voting user interfaces to the paper user interfaces they used previously [5]. Further, it is easy to compensate for the falloff effect; candidate names can be rotated by the software and displayed at the UI, without any additional cost or administrative difficulties.

System failure has been a problem with current electronic voting systems [16]. However, we believe such system failure is a problem with particular systems, particular implementations of those systems, and insufficient testing, rather than a problem with electronic voting in general. Further, we believe that an N-version voting system can be significantly more reliable than existing systems and comparable single-version systems.

1.2 Electronic Voting System Criteria

A reliable, trusted voting system is a vital to communities and countries where matters of importance are decided on by voting. Because of the importance of such systems, as well as the disappointments arising from the use of flawed electronic voting systems, much work has been done in establishing criteria that a sound electronic voting system must necessarily satisfy [18-22]. The major results of such research have been summarized in the following electronic voting system criteria:

- ***System and Data Integrity and Reliability.***
  The behavior and output of the voting system must be correct and must not be altered by tampering with the system or with any data involved in entering and counting votes.

- ***Personnel integrity.***
  The persons involved in developing, operating, and administering an electronic voting system must be of unquestioned integrity.

- ***Operator authentication.***

The persons authorized to administer an election must gain access to the voting system only through nontrivial authentication mechanisms.

- ***System accountability and verifiability.***
  All internal system operations, including testing and modification, must be monitored without violating voter confidentiality. Additionally, the correctness of the election result must be verifiable.

- ***Voter anonymity and data confidentiality.***
  The voting counts must be protected from external reading during the voting process. Also, the association between recorded votes and the identity of the voter must be completely unknown by third parties as well as within the voting system.

- ***System credibility.***
  The system's trustworthiness must be irrevocably established and assured.

- ***System availability.***
  The system must be protected against both accidental and malicious denials of service, and must be available for use whenever it is expected to be operational.

- ***Interface usability.***
  Systems must be amenable to easy use by local election officials, and must not necessitate the on-line control of external personnel (such as vendor-supplied operators). The interface to the system should be inherently fail-safe, fool-proof, and overly cautious in defending against accidental and intentional misuse.

While the above criteria are not provably sufficient, they have generally been agreed upon as necessary [18-22]. As such a sound, reliable and secure voting system must be expected to meet the above requirements.

1.3 Related Work

Research on voting has led to established electronic voting systems [23-26], voting schemes and protocols [27-30] and prototyped voting architectures [31-33] which meet some of voting criteria. However, as single-version systems, they are theoretically unreliable because they contain single points of failure and place an excessive amount of trust in the ability and integrity of programmers [23-35]. Single version systems are extremely vulnerable to bugs, environment and compiler errors, trojan horses and trapdoors. The presence of even one of those faults in critical sections of the code could destroy system reliability and integrity. Single-version systems are also particularly vulnerable to corrupt or careless insiders. These issues make it difficult to guarantee that the data confidentiality, integrity, and reliability criteria will be met. In fact faults and vulnerabilities have been found in some of the established voting systems [8-12, 36]. Popular cryptographic voting schemes, such as the mix-net model, the blind-signatures model and the homomorphic encryption model [27-30, 34] are promising but, as presented, they still suffer from the problems typical to single-version systems. Also, these cryptographic models are much more complex than our system, and as such would face the disadvantages associated with relative complexity [37]. In particular, they are likely to be more difficult to correctly implement and review than our simpler system [37]. These issues with current systems and protocols have motivated us to research an N-version voting system. We believe that our voting system is an improvement, in practice, over current schemes.

1.4 N-version Programming

An N-version System (NVS) consists of (1) several software modules - developed in controlled isolation - which implement an identical function, and (2) a decision algorithm for determining the system consensus result as a function of each software module's result. The process by which the N-version Software modules are produced is called N-version Programming and is the focus of much research [38-43]. The key advantage of such N-version design, implementation and execution is that structurally there is no way the whole system can be compromised without compromising a significant number of the parts. The modules are simple and they corroborate each other, so we can fairly certain

about the accuracy of the results that they elect together. Further, we can ignore the results that are not corroborated by the other modules. These properties make an N-version system more robust and trustworthy than corresponding single-version architectures – that is, the single-version system that would result when N is 1.

N-version Programming assumes that a majority of the components implementing the same function fail at different points, if at all, so that the failures can be detected and corrected [43]. The N-version community believes that this can be achieved by introducing diversity [38-43, 44]. In particular, diversity may be introduced in the following elements of the NVP process: (1) training, experience, and location of developers; (2) algorithms and data structures; (3) programming languages; (4) software development methods; (5) programming tools and environments (including compilers); (6) testing methods and tools [38, 44]. Diversity may also be introduced in the NVS execution environment, by running the N-version software modules on multiple computers and communicating with them via multiple channels [45].

Software and environment testing can help ascertain that there is a sufficient amount of diversity in the system. Already, there has been significant research done on measuring diversity [43] and commonality checkers exist which can be used to test how similar two software modules are [46].

1.4.1 N-version Performance on Electronic Voting Criteria

This section sketches the ways in which the N-version system is, by design, able to meet each criterion.

System and Data Integrity and Reliability

The N-version System is resistant to tampering: a majority of software modules must be corrupt or receive corrupted data, in order for the behavior or output of the voting system to be affected. Additionally, because of the separation of knowledge in the SAVE system, many modules from different stages must collude for significant voting secrets to be revealed. If each module is compiled on a different compiler, the system as a whole

can resist mistakes introduced by Trojan horses and compiler bugs. Additionally, traditional steps can be taken to improve the quality of each module. In particular, with proper testing and quality assurance techniques such as software versioning and certification, the system can make guarantees about which code is currently a part of the system and how established the code's correctness is. Read-only software executables, run from once-writable memory, can prevent the modification of software at run-time.

Personnel integrity

As with any important system, special attention should be paid to acquiring developers and administrators with established integrity. An N-version system however further decreases the motivation for dishonesty, because each worker is made aware than he would have to corrupt a majority of other well-isolated components or workers in order to affect the overall behavior of the system.

Operator authentication

Operator authentication can be achieved in a variety of ways, including biometrics, non-trivial proofs and dynamic passwords. Additionally, with N-version software, operators have access to only a few modules or components within the system, making the system overall less susceptible to misuse.

Voter anonymity and data confidentiality

With suitable encryption of ballots and messages, separation of ballot encryption from identifying information encryption, module authentication and blind-signatures, it is possible to maintain both voter anonymity and data confidentiality.

System accountability and verifiability

As demonstrated by our own N-version voting system, an N-version system can be designed to provide audit trails that correctly reflect the behavior and output of the system. The audit trails will reflect the state of the majority of modules, or more generally, the decision made by the N-version decision algorithm at each stage. The audit

trails, combined with the system integrity described previously, provide accountability and verifiability, both in implementation and execution.

System credibility

The system's trustworthiness must be irrevocably established and assured. Popular methods of establishing trust include allowing all or part of the system to be open source and having the entire system certified by trusted, expert persons or groups. These methods are suitable for N-version systems as well. To support such disclosure, the entire N-version process must be clearly and consistently documented, including the assurance and testing measures taken for each module. Additionally, we are investigating the feasibility of promoting trust by allowing the public to contribute specially designed, rigorously tested modules that can reassure contributors without compromising the system. N-version systems are not dependent on any one of its modules, and this mitigates the risk of including a publicly contributed module. The SAVE system could still function correctly, even if some of the publicly contributed modules were faulty or malicious. But public trust would be higher since individuals themselves, or persons that individuals respect and trust, contributed to the SAVE functionality.

System availability

An N-version Voting Architecture is well suited to execution on a distributed system which makes system availability much likelier. With time limits on module operations, we can treat computers or channels that fail, just as we would treat any corrupt or erroneous module. In this way, voting can continue as long as a majority of modules do not stop or otherwise fail. An N-version system is therefore likely to be more available than a single-version electronic voting system which could be brought down by bugs or hacks at critical system points or denial of service attacks at its communication channels.

Interface usability

We can leverage the extreme modularity of N-version programming to create an interface that is easy for voters to use. SAVE has numerous measures in place to ensure that the interface is easy to understand and use. At the point of voting, the most suitable user

interface display style can be selected dynamically and swapped in without modifying or affecting the rest of the system. Our research group is doing a lot of significant work on user interface design to meet the varying needs of the voting public [47]. This work will be leveraged by the SAVE system.

1.5 Introduction to SAVE - A Secure Architecture for Voting Electronically

While there has been research which approached various voting problems and led to systems which meet some of voting criteria, our system is more comprehensive. We have developed a practical, simple voting architecture which we believe meets voting system security and soundness criteria while addressing the voting community's concerns about electronic voting [48-50]. We have named this voting architecture a Secure Architecture for Voting Electronically (SAVE). The SAVE concept, including its fundamental design and principles, was developed by our research group and introduced in [45]. This thesis refines and extends that research, by creating an end-to-end secure, robust and accurate voting architecture.

1.5.1 SAVE Objectives

The following are key objectives of the SAVE Architecture. These objectives follow closely the voting system criteria established by the voting community [18-22].

Minimal Trust; No Single Point of Failure

We recognize that minimizing the trust placed in individual components of the system will increase the robustness and security of the system. To this end, we trust no single component; each component assumes that any other component, whether human or software, could be malicious or make errors. The SAVE system is made N-version, where each component is suitably diverse, so a significant number of components must be malicious, erroneous, or successfully attacked at any stage, in order to corrupt that stage. No single portion of software, and no single programmer, can compromise the system, that is, there is no single point of failure. This trust model distinguishes our voting research from the rest of research on electronic voting, where some component of

the system must be trusted [35] – generally the programmers who contribute to the system implementation or some central administrator or authority.

Redundancy

Another key attribute of the SAVE system is redundancy. As an N-version system, we have n versions of each type of software module, and they all implement the same function. This redundancy facilitates the N-version error detection and correction discussed earlier. More dramatically, the ballots themselves are made redundant: when a vote is cast, n copies of the ballot are immediately created to form part of the initial inputs to each of the redundant components at stage 1 of the process. Thus the SAVE system is able to recover if some of these copies become corrupted or are lost – as long as a majority of the ballot copies arrive successfully at the stage 1 components.

Robustness

The N-version foundation gives SAVE the ability to detect and correct stage result differences which are likely errors. Additionally, the software components that make up the system are run from different computer hosts and have access to separate and secure storage. These features result in increased availability of the system during execution. Failures of a few software components do not cause the whole system to fail. Also, attacks on a particular host or channel will not be able to disrupt the system because non-responsive components are treated as faulty and ignored, allowing execution to continue. This makes the SAVE system well able to deal with unreliable networks in a distributed environment, such as the internet. These features combine to create a robust system.

End-To-End Security

Many schemes and protocols have been presented that achieve security in some parts of the system while ignoring others [27-30, 34]. These can be useful, but must be included in an end-to-end scheme to be truly and practically usable. Security goals must be established and maintained regarding the hardware that these systems are executed on, the compilers that are used, the human and other forms of secret storage, etc. The SAVE

architecture is an end-to-end system and particular attention has been paid to system security.

Verifiability

Verifiability is crucial to assure correctness of the system, promote public trust in the system and facilitate recounts. SAVE establishes electronic audit trails to audit each vote cast. The electronic audit trails are generated separately, by independent modules, and are stored on different computers, in such a way that they are not modifiable. Also, a majority of the audit trails extracted from these computers must agree before the audit trail is accepted. These n-version electronic audit trails are more reliable than a redundant audit trail made by copying the single result of a single-version system. Also, these electronic audit trails are far more robust than a paper audit trail which may become lost, destroyed or corrupted by one malicious act or error. We note however, that the SAVE system is capable of incorporating paper audit trails if social policy demands them.

Simplicity

One of the tenets of the SAVE system is simplicity. The benefits of minimizing complexity are well documented: minimizing complexity in a system tends to also minimize bugs, while making the system more testable and more easily documented and maintained [37]. We have taken steps make the design and implementation of SAVE as simple as possible. The system is designed such that each module has a small and relatively simple task to perform. This means that the module can generally be implemented in a few hundred lines of code at most. Small modules implementing simple tasks are generally easier to review, test and maintain than complex ones. These features also make the modules easier to certify by independent groups. Also for simplicity, we use a minimal operating system with a trusted computing base, rather than a large, complex operating system whose correctness and security would be more difficult to attest.

Accessibility

Many persons are unwilling or unable to vote because current systems are not sufficiently accessible [5]. Additionally, user interface problems are arguably responsible for most of the spoilt ballots that are generated during elections [2-3, 7]. The modularity of the SAVE design is particularly useful and appropriate at the user interface because it allows the Module to be updated easily. Another advantage of the SAVE architecture is that the User Interface Module completely separates content from presentation style. Thus the same ballot can be presented in any appropriate style without having to recertify software, reinitialize voting machines, or undergo any of the lengthy and cumbersome processes that would be necessary in monolithic systems. This separation, as well as the general modularity, allows the User Interface to keep pace with human factors' research and create the best possible voting experience. Already, our User Interface contains many features, such as textural and audio cues to important voter actions, which make the User Interface more accessible than paper ballots. The flexibility of the SAVE user interface, as well as the results of our research in ballot and user interface design, should increase the number of voters who are able to vote independently.

As indicated in this chapter, N-version programming increases the reliability, security and trustworthiness of systems. These effects are particularly useful for voting systems where high reliability is required and trustworthiness is demanded. As such, we feel that an N-version system such as SAVE, can improve voting. In general, our research is an important contribution to voting because it highlights the applicability of N-version programming to voting.

Chapter 2

N-Version Programming

N-version programming (NVP), as it is applied to software, was introduced in technical literature in 1977 [38, 40]. The definition and constraints of N-version programming have evolved over the years but can be stated, in general, as the independent generation of N functionally equivalent programs, possessing all the necessary attributes for concurrent execution, with specified state to be compared at expressed points along the execution. The action to be taken at the each comparison point is also specified, but minimally involves the election of some subset of "valid" states from among the states produced by the N versions at that comparison point [38]. At each comparison point, a decision algorithm responsible for that election is executed from within the N-version execution environment (NVX). N-version researchers theorize that independent software generation and further diversity introducing techniques will lead to the creation of software versions which contain significantly different faults that are unlikely to cause the same failures at a comparison point [38, 40, 51]. These researchers therefore conclude that if a majority of versions agree on some output, that common output is likely to be correct [38, 40, 52, 51]. The result of the N-version Programming process is an N-version software (NVS) unit [38]. Though there has been significant research questioning the validity of the N-version assumption [53-54], recent research has been much more positive, in particular, several research groups have affirmed the significantly superior correctness and availability that N-version systems are able to achieve relative to single-version systems [55-58]. The challenges and benefits of N-version programming, and the justification of the N-version assumption as applied to the SAVE system, are discussed here.

2.1 N-version Questions and Research

The fact that non-trivial systems cannot be implemented without faults is at the foundation of N-version programming [37, 59, 60]. Single version software is particularly vulnerable to the errors that result from these faults because they do not generally have means of error detection and correction. Further, these faults and their

failures persist in software; the errors caused by a fault are guaranteed to happen at any time that the environment meet's the fault's failure requirements [59]. N-version programming was established in an attempt to reduce the effect of these faults on the containing system. N-version programming uses functional redundancy to generate, for each input, many possibly correct states at each comparison point. The redundancy ensures that states being compared are equivalent and correct in versions that have not failed on the input. Additionally, N-version programming uses diversity to minimize both the number of errors for a particular input and the number of similar errors for that input. If the incorporated redundancy and diversity indeed achieve these tasks, then their combination implies that larger sets of equivalent states are more likely to be correct.

2.1.1 The Question of Correctness

The decision algorithm is critical to N-version software since its ability to elect correct states from the modules directly determines the reliability of the N-version software unit as a whole. It is therefore necessary to be clear about exactly which parameters and considerations affect the decision algorithm and its output. For each of the states at a comparison point, the proportion of versions that share that state corresponds to the likeliness that the versions with that state are correct. The simplest decision algorithm therefore merely elects the state that occurs most frequently among the versions at the comparison point – if more than some minimum proportion of versions share that state. N-version programmers assume that sufficient versions are near enough to correct that some minimum threshold of them will give the correct output for each input in the input space with high probability. The optimal threshold value varies for each system and is difficult to quantify, but it is often set as the absolute majority with respect to N, $\lceil N/2 \rceil$. In fact, the threshold's value is a function of the confidence required in the correctness of the elected state, the diversity of the N modules, as well as the correctness of each of the versions and of any intermediary communication channels that are used. The threshold for a N-version system is difficult to quantify because the exact function relating the dependant variables is unknown and some of the dependant variables are themselves difficult to quantify; quantitative measurements for diversity is non-trivial, and similarly, quantitative measurements for correctness can be difficult for large input spaces. More

complex election algorithms may require that only pre-specified aspects of the states at the comparison point be equivalent, or may have other base or "sanity" checks that each state must fulfill in order to be considered among the potentially correct. Such extensions do not fundamentally modify the N-version paradigm.

Is it reasonable to expect that the typical "consensus-seeking" decision algorithm would lead to election of the correct state with high probability? Generally, N-version researchers accept that the output elected by such a decision algorithm in an N-version system, is theoretically more likely to be correct than the output of a single version system [53, 61, 59, 52]. There is debate however on the extent of the improvement in reliability that N-version systems achieve compared with their single-version counterparts, and whether that improvement is significant enough to warrant the additional requirements and costs of N-version system development [53, 59, 52]. Significant version diversity is necessary to make both the number of errors for a particular input and the number of similar errors for that input small enough to validate the election process and, consequently, the reliability improvement. However, little is known about how to quantify the effects of particular methods for adding diversity on the distribution and type of errors in a system [38, 40-41, 51, 62], making it difficult to determine whether diversity sufficient to validate the election process can be achieved in practice. The methods to be used for increasing diversity and the nature in which said methods will affect system failures are still largely decided by intuition and qualitative prediction [38, 40, 41, 62], but research geared towards developing rigorous, testable, quantitative methods are being developed [51, 61-65].

2.1.2 N-version Research

Research has led to significant breakthroughs in the study and defense of N-version programming [38-42, 58, 61-62]. Avizienes, Popov, Strigini, et al have contributed with methodologies of N-version programming, describing processes and tools for the specification, implementation, testing, execution and maintenance of N-version software [38, 40, 51]. This and other research have contributed towards maximizing diversity and thereby minimizing coincident failures and have led to favorable results in practical N-

version applications [38-42, 51-52, 58-59, 60-61, 67-72]. Further, there has been some helpful developments in the areas of modeling and quantification of diversity and reliability in N-version systems [51, 61-65].

Knight and Leveson demonstrated that independent development of versions is not sufficient to cause independence of failures [53]. Their study was important to N-version programming because it motivated researchers to look more carefully at how to achieve diversity and quantify its effect [38-42, 51-52, 58-59, 60-62, 67-72]. Many subsequent researchers misunderstood these results to be a definitive statement against N-version programming [63]; however this is not the case [53]. Knight and Leveson warned that independence of failures, though mathematically convenient in theoretical work, could not be assumed in reality [53]. As Avizienis asserts however, independent failure is merely an ideological objective, and is not a basis for, or even an assumption of, N-version programming [38, 58]. The presence of correlated failures in no way destroys the feasibility of the N-version programming since coincident failures will only cause system failure if a majority of versions are faulty [38]. Good quality control can reduce the risk that so many modules are faulty significantly [38]. Additionally, researchers have found that negatively correlated failures are possible and these may lead to even higher reliability than simply independent failures [58, 61, 66]. Researchers, including Partridge and Krzanowski, argue that negative correlation can be achieved by "forcing diversity;" that is, by introducing artificial differences in such a way that the developers find different aspects of the problem difficult and as such multiple versions are less likely to fail together [58, 61]. Knight and Leveson themselves assert that neither they nor their results imply that N-version programming does not work or should never be used [53].

N-version programming has been widely used for improving reliability and the resulting N-version applications, in both research and industry, have produced favorable results in terms of increased reliability [38, 57, 67-72, 52, 86]. Faults, expressed as significant disagreements between modules, were discovered and tolerated in the applications created [38, 57, 59, 67-72, 86]. Although it is impossible to generalize from relatively few and specific experiments, this does indicate that N-version programming improves

reliability. In fact, more and more researchers are establishing themselves as definitely in favor of n-version programming for improving reliability, especially when the cost of failure is high [57, 67-72, 52]. As an indication, an average reliability improvement of an n-version software unit over its single version counterpart of up to a factor of 58 has been found in some practical applications [66].

The premise that the money spent in N-version programming could instead be spent on developing a single version that is N times more reliable is false [59]. Experimental evidence has corroborated the law of diminishing returns for debugging software - as programs become more reliable, it becomes harder to find faults [59]. Similarly allocating more money and resources do not reduce the number of faults introduced into software after a point [37]. No combination of quality assurance methods is perfect, thus even if enough money is spent to acquire the best development and QA techniques, there are still likely to be faults embedded in the resulting system [37]. This negates the argument that single version software should simply be made "reliable enough." Further, as mentioned in [41], N-version programming aims to ensure that the system could recover, *if* the modules contain faults. Since it is the case that non-trivial software must be expected to contain some faults despite best efforts to prevent this [37, 59, 60], a user should not believe that a single version will never fail simply because it has not yet done so. In the case of such failure, N-version programming becomes a useful tool for preventing system failure [41].

2.2 SAVE as N-version Software

The SAVE N-version system is carefully designed to achieve, as much as possible, significant increase in reliability over single-version systems, according to the results and conclusions of the major researchers in the N-version area [38, 53, 51, 62]. Implementations of the SAVE architecture follow the advice and considerations of published N-version programming methodologies [38-41, 62] very closely. Further, the SAVE system, by design, avoids many of the diversity and reliability limiting factors [53] that have thus far been discovered.

Researchers hypothesize that programmers tend to make more faults on difficult problems or subproblems, which may lead to multiple common failures across different modules for particular inputs [41, 53, 61, 66]. The SAVE modules are intentionally functionally simple and implementable in a few hundred lines of source code. In particular, each module's functionality is relatively simple, compared to the functionality described in the N-version experiments reviewed [38, 57, 59, 67-72]. We thus expect fewer failures, and fewer common failures in particular, in accordance with the hypothesized relationship between functional difficulty and fault and failure production.

It has been asserted that the required reliability might be achieved, for systems of sufficient quality, by using a larger value for N [53]. The relative simplicity of SAVE module functions will make it relatively easy for implementers of a SAVE system to produce a large number of modules quickly. Further, SAVE's communication protocol allows modules to be easily distributed across many computers or processors, so that the computational work of many modules can be managed. Both features facilitates a large value of N for the SAVE system and the associated increase in reliability.

Traditional factors that tend to be correlated with failures [37], including large software size, as measured by number of lines of code [53]. The modularity of the SAVE software as well as the relatively small size of each module, as measured by lines of code, suggest that SAVE should be able to generally avoid the coincident failures that are generally caused by those factors.

SAVE's careful development adherence to established N-version programming methodologies as well as its inherent features make us fairly certain that the SAVE system can derive maximum benefit from the application of the N-version concept. With the use of N-version programming, we believe that SAVE can achieve greater reliability and trustworthiness than current voting systems [8, 13-14, 16].

2.3 SAVE's N-version Methodology and Concepts

N-version programming is similar to traditional software development, but features and processes are added to combine individual modules into a fault-tolerant unit and maximize diversity given development costs and other constraints [62]. The N-version Execution Environment (NVX) is added and tasked with creating a fault tolerant N-version Software (NVS) unit from the individual modules. Special restrictions are added to the functional specification design to ensure that they do not limit diversity and that they facilitate the mediatory work done by the NVX. Also, a special coordinating team (c-team) is established to ensure that maximum diversity is injected into the modules and that this diversity is not reduced by communication across module development teams. The c-team is also responsible for maintaining the overall system's quality. The key concepts, features and processes related to N-version programming are discussed here.

2.3.1 Diversity

A fundamental conjecture of N-version programming is that the diversity across the modules will greatly reduce the probability of identical software failures occurring in multiple modules at any comparison point [38, 51]. Diversity may be incorporated into the software modules via: (1) training, experience, and location of personnel; (2) algorithms and data structures; (3) programming languages; (4) software development methods; (5) programming tools and environments; (6) validation and verification methods and tools [38, 59, 62]. This diversity can be applied at all stages of the N-version programming process, including at the design, specification, implementation and testing stages.

The incorporated diversity may be random or may be forced. Random diversity is diversity achieved in an uncontrolled manner, relying on dissimilarity between the individual's training and thinking processes to generate significant differences [38, 39]. Forced diversity, on the other hand, is introduced in a deliberate, calculated manner [38, 40, 51]. At a basic level, forced diversity involves locating points where diversity could be inserted into the modules directly or into the software development process, determining the best diverse approaches that could be applied at those points, and

requiring that those approaches be uniformly distributed at random amongst the software development teams. Forced diversity has been showed experimentally to be more effective than random diversity in reducing the correlation between software failures [38, 58]. Further, it has been theorized that forced diversity could lead to negative correlation between software failures; this is better than the uncorrelated failures that would be expected of independence [41, 58].

A contribution of diversity to the N-version programming process is that the developers implement very different approaches to the specification. With different initial implementation choices, particularly when they are prescribed by forced diversity elements, the developers likely face different problems, challenges and difficulties while implementing the modules [38, 41, 58]. Thus the developers of different modules are less likely to introduce the common software faults which may lead to identical software failures during execution [38, 41, 58]. The amount of diversity chosen is dependent on the costs and the funds available, the time constraints and various deadlines for project progress and completion, and the required dependability of the system [38, 62]. It has been suggested by Popov and Strigini that potential sources of system failures should first be identified, and diversity chosen to best eliminate or remove the effect of these sources, within the constraints mentioned [62]. The amount of diversity present in a system is expected to vary from implementation to implementation, as different development teams are used; however certifiable SAVE systems will have some minimum required diversity associated with the collection of modules at each stage.

2.3.2 The N-version Execution Environment (NVX)

The N-version Execution Environment (NVX) is another fundamental of N-version Programming. The NVX is the software and/or hardware component that manages the N individual software modules, constructing an N-version Software Unit (NVS) from the inputs, outputs and behavior of the individual modules. Specifically, the NVX is the execution environment of the system, containing the set of functions that are needed to support the creation of a fault-tolerant N-version Software Unit (NVS) from the concurrent execution of N member modules. These functions can be applied to any set of

software modules generated from any pre-approved, pre-specified module functional specification (VS). SAVE features, such as mutually distrusting software components, a distributable execution environment, and unreliable channel communication, add special constraints to the NVX. In particular, the NVX must not transfer problems like hacks and viruses from one module to others, the NVX must maintain authenticated addressing to locate all the modules under its control, the NVX must recognize that the number of correct states that it compares might be less than the number of correct modules at that stage because of channel corruption, and the NVX cannot assume that it is safe from attack. For these and other reasons, the typical single NVX is not used in the SAVE system. A single NVX is unable to provide the availability, correctness and trust assurances that our voting system demands. We therefore use multiple NVXs, which meet the general NVX responsibilities discussed here and also satisfy the special constraints of our SAVE system.

2.3.2.1 Diverse, Redundant NVX implementations

Software modules do not trust other software under SAVE's trust model. Thus, no software is given access to a software module's memory or allowed to directly call one of a software module's functions. Also, the SAVE model allows software modules to be distributed across different computers at different locations. Standardized XML message protocols were developed to allow software modules to communicate with each other in the face of these constraints. These message protocols were built over the authenticated SSL protocol for secrecy and authentication. Note however, that the SAVE model allows for some unreliability in the communication channels across which these XML messages are sent and this unreliability could lead to message corruption. This relatively harsh environment is unsuited to traditional single NVX models [38]. A single NVX's decision may be severely compromised if its communication channels are error prone so that many correct input messages are corrupted before they are considered by the decision function. If too many correct input messages are corrupted, a majority consensus would likely become impossible and the NVX would be unable to elect an official input for use by the modules at future stages. This would be a critical failure since the modules at all future stages would have no access to correct inputs and thus the system would produce

an incorrect result overall or be unable to return a result at all. The same type of system failure could result if the NVX is itself malicious or faulty. A voting system must be available and must be extremely reliable. Therefore steps must be taken to avoid such system failures and prevent overdependence on a single NVX or any other piece of software. SAVE removes the single point of failure of the typical NVX by including multiple, diverse NVXs in its design. The SAVE system provides a functional specification (XS) for the NVX that is independently implemented within each module by the module developers. In this way, a problem with the implementation of one NVX does not compromise the entire SAVE system. Additionally, there is no single NVX to become the focus of hackers or malicious insiders.

2.3.2.2 NVX responsibilities

The NVX must support the N-version Software execution. Among other duties, the NVX must: facilitate any necessary inter-version communication; manage any module synchronization and enforce any timing constraints; provide and execute the decision algorithm(s) for electing "likeliest correct" inputs; perform error-masking for each stage at the inter-stage comparison points; execute any decision functions for error-correction or other treatment of faulty modules and, in general, manage system correctness and efficiency [38].

Inter-version communication

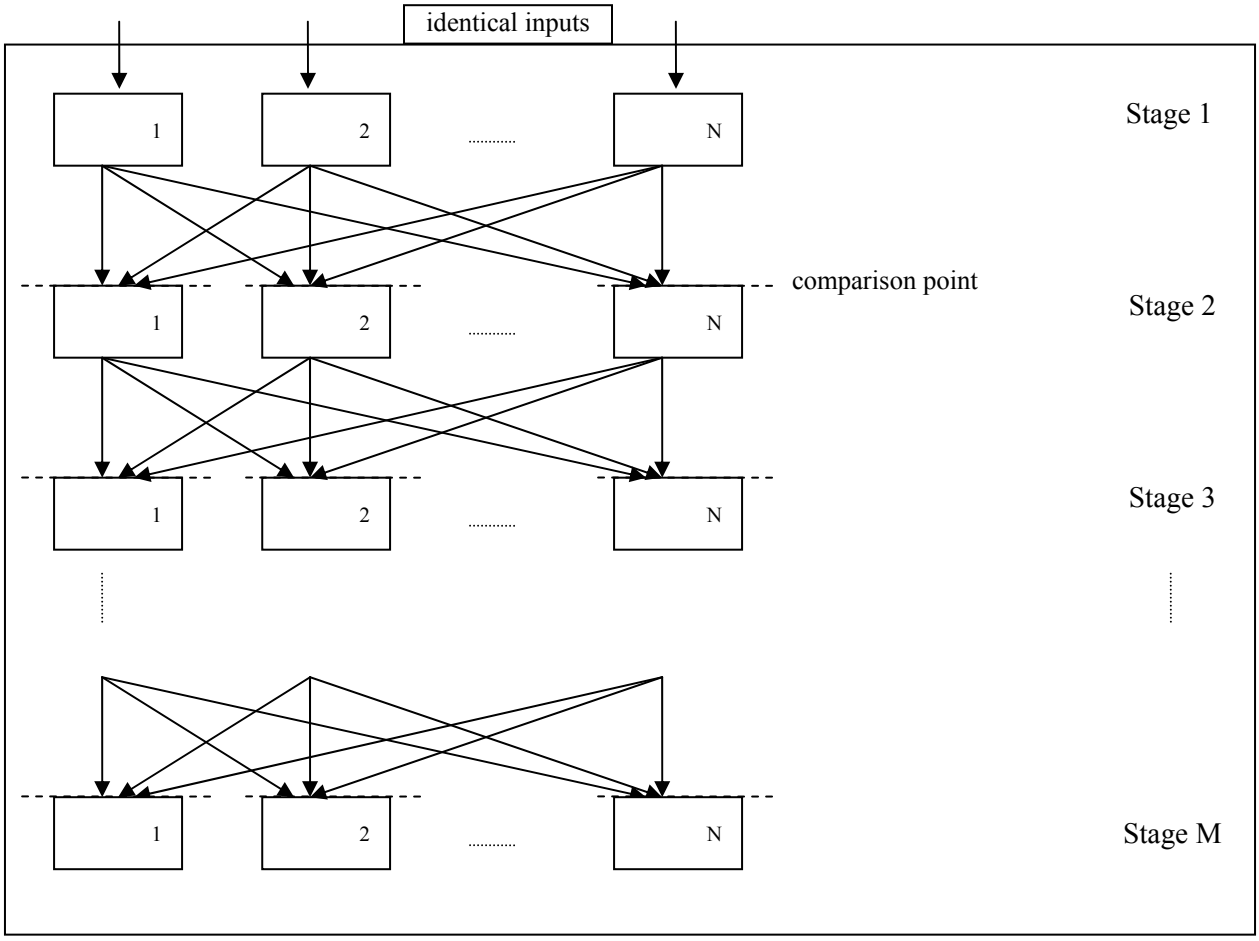

Figure 2.1: Generalization of SAVE communication network

Figure 2.1 depicts a generalization of the SAVE communication network. There are N modules at each stage of the voting process and there are M major stages. Each module is both a server and a client and can send, process, respond to and receive messages from other modules. Each module has an identical directory, containing the addresses and functions of all the modules. The directory is digitally signed on creation, so that any modifications to the file can be detected. This prevents malicious alterations to the file and preserves its authenticity. The modules manage their own communication with other modules. This communication is constrained to XML messages, each of which has a

specified format and results in a specified computation at the server module and, if specified, also results in the generation of a standard XML message response.

The principle comparison points occur at the points between SAVE stages; outputs from one stage are compared and their errors masked before they become the inputs to the next stage. Modules therefore typically do not communicate with other modules at the same stage. Instead, modules typically communicate their output to the NVXs of modules in the subsequent stage. Each module decides, on its own, when to send its output – but typically does so as soon as it completes its function. This relative independence is unlike traditional NVX models, where module computation might be forcibly interrupted by external NVX implementations. Despite this lack of interruption, each module knows that it must execute its function within a pre-specified amount of time otherwise its output is ignored. Therefore system performance time is constrained and some performance guarantees can be made.

Each module at a particular stage receives messages from any of the modules at the previous stage, at random, until messages have been received from all modules or the waiting period has ended. At this point, if possible, the module elects an official message from the ones it has receives and operates on it; then sends the output randomly to modules at the next stage until it has sent an output message to all the modules at the next stage. All communication between modules occur in this format, with the exception of the witness modules, which perform no election, but operate on each message they receive and respond directly to the module that sent the message, and the User Interface modules which communicate with each other, at the same stage, so that they can collectively elect what should be displayed to the voter.

Note that we must also assume that a large enough number of communication channels correctly relay their message each round that a majority of correct messages are received by correctly functioning modules in the receiving stage. This is necessary so that error-masking can occur at the receiving stage and the receiving stage can thus recover from errors at previous stages.

The Decision Algorithm and Error Masking

Each module, n, implements the decision algorithm specified in the XS. The parameters of the decision algorithm are extracted from the messages sent by the modules at the previous stage. In particular, the modules at the previous stage all send a specified XML message to module n. That XML message indicates to the server module that a particular function should be executed, and the contents of the XML tags in the message provide parameter values for the function to be executed. Specified XML attribute and element values must be equivalent for the XML messages to be considered equivalent. These attributes and elements constitute *matching features* of that input. If the *matching features* in the XML messages provided by two modules are the same, the XML messages are considered equivalent and their contents equally likely to be the correct input to the server function to be executed. The decision algorithm is used to decide which of the inputs to a function is "likeliest correct" before the function is applied. This decision is based on the size of the largest consensus group among input XML messages. If the largest consensus group has more than some *k* input XML members, then the *matching features* that describe that consensus group are chosen by the decision algorithm to be "likeliest correct." *k* may be set, for example, as $\lceil N/2 \rceil$ where N is the number of modules in the previous stage, as listed in the module address-book. For this value of *k*, a majority of the modules in the previous stage would have to succeed in sending the same *matching features* in their XML message for those *matching features* to be elected "likeliest correct." If the size of largest consensus group is less than or equal to *k*, no "likeliest correct" message is elected; the server does not perform any further operations, in particular it does not execute the function indicated by the XML messages. Otherwise, the relevant *matching features* from the elected consensus group is used as the input to the server function to be executed. In this way, error-masking occurs before execution of the server function; the function is applied only once, to the input deemed most likely to be correct. The decision algorithm is executed only if enough input messages were received from the previous stage to possibly constitute a consensus by the end of the waiting period. During its wait period, each module collects as many inputs as possible from modules at the previous stage, so that it can make the best possible choice of which input message is most likely to be correct. Note that input messages that are

incorrectly formatted or unauthenticated are immediately discarded. Also, repeat related messages from the same client module are immediately discarded.

Treatment of Faulty modules

The SAVE NVX performs error-masking, but it does not seek to immediately correct or abandon modules that provide an incorrect input message. The SAVE threat model allows channel failure, and so a server module can not determine for certain whether the client module that sent the incorrect message is corrupt or whether the client's message was corrupted later, in the communication channel. Correspondingly a client module that sends no output at all could be faulty or its message could have been lost in the communication channel. If the client module repeatedly sends incorrect messages, this increases the likelihood that the client module is itself faulty, but still does not guarantee this. The server could arbitrarily ask the client to repeat itself, and this could eventually prove that the client is capable of providing the right output. However this still could not prove definitively that the client is, or is not, faulty or malicious. A series of transient failures – or even permanent failure – within the channel, could make a correct module look faulty, or a malicious module could provide incorrect results at first, and correct results later as desired. Further, arbitrary requests would arbitrarily increase the number of messages passed in the system, as well as the time until a decision can be made; none of this loss of efficiency is desirable. Thus, the server simply notes client modules that repeatedly send incorrect messages and may take steps to reduce their credibility.

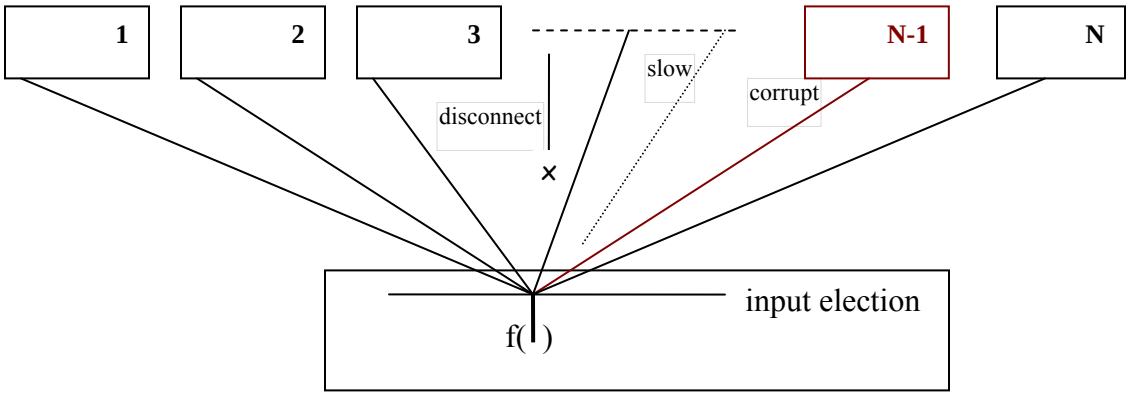

Figure 2.2: SAVE cross check point

As is the case with many N-version systems, the SAVE system is semi-synchronized. Each module compares the outputs of the modules from the previous stage and elects one such output as the correct one, if possible. Figure 2.2 depicts the events that might occur at such a comparison point. Each module must wait for more than k modules from the previous stage to send a particular input message before it can perform the operation associated with that message. The receiving module does not wait overlong for input from slow, corrupted or disconnected modules, however the module must wait long enough to receive at least a majority of related messages so that it can be sufficiently certain that a consensus was achieved. This tradeoff is weighed carefully before deciding on an appropriate wait period for each stage of each implementation of a SAVE system. The wait period specifies how long each server module should wait for inputs at any particular round. The wait period begins only after some significant fraction of the expected messages arrives. This prevents a few malicious modules at the previous stage from sending messages exceedingly early to force the server module to exhaust its wait period before receiving sufficient inputs. After the specified wait period, the module compares the inputs it has received and elects, if possible, an official input from these as specified by the decision algorithm. If no consensus input is discovered by the end of the

maximum waiting time, the module discards the related inputs it has received thus far and performs no computation on them. Since the module does not perform its function, it "fails" this round and, in particular, sends no output to modules at the next stage.

All modules at the same stage perform the same function, though they are diversely implemented. Additionally, the modules at the first stage all receive the same input message. Therefore it is expected, by our assumptions, that all functioning modules in the previous stage will eventually send along a message, M_i, or some variant of that message, which corresponds to the original input message after it has been operated on at all the stages it passes through. Since we assume that a majority of modules at each stage are correct, and that a sufficient majority of channels are also correct, we can assume that there are not sufficient slow or faulty modules or channels to delay a receiving module past its waiting period. That is, the expected time to receipt of a consensus is well within the waiting period of the modules at each stage.

Each module at a stage must wait for some majority ($k > \lceil N/2 \rceil$) modules from the previous stage to complete their operation and send outputs to the module. Each module at a stage decides on a random recipient order before sending its output to the modules at the next stage; thus each module at the next stage has the same chance of receiving a validating majority (ceil N/2) of input messages during its waiting period. This also means that the modules are given a better chance of completing their function near the same time and before the end of the specified time allotted. Thus, the actions of the modules at each stage are typically loosely synchronized. Also, because of the ability of the system to continue even if a minority of modules is disrupted at each stage, we do not expect an entire stage to be slowed down by a few slow modules or channels. Deadlocks are not possible because the unidirectional communication flow ensures that each stage must wait only on stages before it, and that earlier stage operation happens first.

System Correctness and Efficiency

The NVX is responsible for managing the correctness and efficiency of the N-version software. As such, the number of comparison points must be large enough that the

correctness of the system is sufficiently high, yet small enough that the system is not delayed too long by the time used by the NVX to perform its computation. Each module is slowed somewhat by waiting to collect its input messages and then comparing them. The extent of this delay is calculated and managed so that the probability of correctness of the elected *matching features* is sufficiently high. In particular, the length of the wait period for new inputs at a module, and the return time constraints of the modules at each stage is carefully optimized and planned. The actual times would vary with the actual software specifications, the number of modules at each stage, the number of stages in an execution of the system, the number of modules run per processor, the network speeds, the extent of the distribution of the modules on different computers throughout the network, and other such constraints.

2.3.3 N-version Software Development

At every stage of development, the needs and requirements of the N-version process must be made to coincide with the needs and requirements of the software development process. In particular, the N-version goal of diversity maximization must be targeted without overly reducing the achievability of the traditional quality goals of software development. The N-version programming process refines the nature of the functional specification and greatly influences subsequent software development. Additionally, the N-version programming paradigm directs the testing, deployment and maintenance of the N-version software developed. Directing all aspects of the software creation as specified by the N-version paradigm [38] is necessary to establish and maintain the diversity that is so critical to N-version software, while maintaining software quality. We discuss additional constraints and modifications to traditional software development.

2.3.3.1 Functional Specification

The functional specification is designed to meet user requirements and is generally the introductory point for the N-version software implementers. We describe two sets of functional specifications: the module specification (VS) which describes the functionality of the individual software modules that will comprise the N-version software unit, and the N-version execution environment specification (XS) which describes the functionality

of the N-version execution environment (NVX). It is critical that the specifications be as error-free as possible. This is because experiments have shown that error in the specification is one of the predominant causes of identical software failures across modules [53, 71, 61, 66]. It has been indicated experimentally that formal specification languages, which offer verification of completeness and correctness, may help reduce the number of occurrences of specification faults and their related software failures [38]. The specifications must also be written carefully to avoid unnecessary constraints, examples, or other guidelines which may be followed by multiple developers and thus reduce diversity across the modules [38]. The specifications set the baseline for the diversity and functional equivalence of the software that will be implemented from them.

The N-version execution environment specification must be developed in conjunction with the module specification as the interaction between the NVX and the software modules is critical to the N-version software unit. Care must therefore be taken to optimize their interaction by specifying the interface points, parameters and behaviors most likely to lead to correct and efficient execution of the N-version software. With SAVE, the NVX functionality and thus specification is integrated with that of the software modules. Thus the simultaneous mutual specification generation is inherent in our design.

The state to be compared at each comparison point must be made explicit in the functional specification (VS) of the software modules and also in the functional specification of the NVX (XS). Each of the NVX implementations in a stage must have the same set of comparison points and *matching features*, so that the NVX can support the output of all software modules from the previous stage, regardless of the non-functional ways in which it differs from other modules. Conversely, the VS must explicitly state the same comparison points and *matching features* as the XS so that the software modules can provide the data that the NVXs expect to find in the XML messages. The XS must specify the same compare method and decision algorithm for all NVXs at the same stage so that the definition of correctness can be consistently maintained and enforced. The number of comparison points and the amount of state

required to match at each point must also be carefully considered. While a large number of comparison points and a large *matching features* set enhances error detection and recovery, extensive common constraints such as these may limit diversity [38].

Multiple distinct specifications, derived from the same set of user requirements, may also be designed and deployed, perhaps in different specification languages or using effects on different properties to describe the same functional behavior [38, 59, 62]. Multiple specifications would increase the cost of the system as they require independent designers and must be rigorously tested to ensure that they are equivalent [38]. However, diverse specifications are theoretically expected to increase software diversity and also mitigate the impact of single specification error on software failures, and identical failures in particular [59, 62].

2.3.3.2 The Coordinating Team (c-team)

Correct specification is crucial to the success of NVS software. However software implementation, testing and maintenance must also be guided by the N-version paradigm to maintain diversity and prevent similar faults from being introduced into the software. Software development is similar to traditional methods from the perspective of each development team. However strict guidelines must be implemented and followed to maintain diversity; the creation, implementation and maintenance of these guidelines are the main task of the N-version *coordinating team* (c-team). The c-team is responsible for managing the general isolation of the development teams as well as the limited communication that these development teams are allowed, in such a way as to maximize the diversity of the resulting software modules. In addition to looking after the diversity needs of the N-version software, the c-team maximizes software quality by managing the functional specifications, as well as the progress - in terms of source code as well as documentation and test sets - of all module development teams and of the system overall.

Careful isolation and independence of software developers and their software is important to ensure that similar ideas and techniques do not spread across development teams and their modules [38, 40, 51]. Potential “fault leak” links along which such

similarity may spread include casual conversations or mail exchanges between developers, common flaws in training or manuals, use of the same development tools like compilers [38]. In general, communication between development teams is strictly limited, if allowed at all, to prevent the permeation of ideas that would limit diversity. Communication between the c-team and the development teams are carefully structured to prevent the c-team from unduly influencing the development teams and from inadvertently spreading ideas from one development team to another thus limiting diversity. The c-team is tasked with identifying and avoiding the potential fault leaks that may arise through such communication. Several measures are taken to help reduce the risk of inadvertent communication that may reduce diversity. These include clear expression and enforcement of the rules of isolation and their purpose; physical isolation of the developers, such as separate working spaces and separate computers for software development; as well as authentication-schemes and access control lists for each module's software and other files [38].

Isolation and independence, while important, must however be balanced with avenues for feedback, questions and error, bug or general problem reporting. Such communication is vital for the module developer and cannot be ignored if quality modules are to be developed. The c-team is responsible for creating and maintaining the communication protocol. This includes setting up the communication infrastructure, tackling such issues as which email or other addresses or phone numbers are to be used for communication; what format the communication should take and what are the time lines for responses; as well as what hardware, software or other materials or assistance is needed to create, dispatch and receive communications. The c-team must then decide whether and how to respond to each query and whether a response should be made just to the development team making the query or whether it should be a broadcast to all teams. Queries may include comments, bug reports, or questions regarding the specification or other issues such as funding or deadlines. All queries and communications, as well as all source code, documentation and other files, are carefully monitored so as to maximize the realized diversity and trace the progression of ideas in the system so that diversity reducing information leaks can be detected and, if possible, corrected. All members of the c-team

must be aware of all correspondence and a significant group of them must agree on each communication they send out or other action to prevent corruption at this level.

The c-team cannot be disbanded as soon as the N-version software unit is compiled however. Diversity must be monitored, and added where possible, throughout the testing and maintenance of the N-version software. Diverse testing and maintenance methodologies can contribute to this as different schemes tend to have different focuses and are unlikely to detect, solve or create the same problems [62]. For example, "operational" testing tends to find faults with higher contributions to unreliability first, as these are the faults that produce operational failures quickly, but "coverage" testing is not biased in this way and is thus likely to have a different fault discovery distribution [62]. Additionally, testing and maintenance methods that tend to reduce diversity, such as back-to-back testing, should be used carefully [62]. Regardless of the methodology that is used, faults detected by the testing teams must be specified clearly, without diversity reducing suggestions, and fixes to different modules must be implemented independently. Similarly the software's upkeep throughout its lifetime must be carried out by independent maintenance teams who follow the N-version programming processes for isolation, monitoring and restricted communication. Adhering to the N-version development paradigm is necessary to prevent the modules from becoming more and more similar over time and thus destroying the system's diversity.

Though the c-team's responsibilities are many and complex, it should be observed that, from the perspective of the developer, the relationship with the c-team closely approximates the traditional relationship with the specification team and the project management team.

Chapter 3

SAVE - A Secure Architecture for Voting Electronically

The SAVE N-version system is a redundant, distributed architecture for voting electronically. The SAVE architecture consists of redundant and diverse modules arranged in a distributed execution environment, such that it is unlikely that a significant number of modules will fail in a way that results in an overall incorrect output. A key aim driving the SAVE Architecture is the elimination of single points of failure. No component, human or software, is completely trusted and this is true at all stages of the development and execution process. Further, as an N-version system, all components are selected and specified so that the likelihood of similar errors are minimized. In particular, software is implemented independently by different programmers using different programming languages, compilers and tools. Also, the system is installed across different computers, using different hardware, software and operating systems, so that common flaws in the underlying machinery can be avoided. Thus the system will continue to function correctly, even if there are failures at a minority of components at each stage, because minority disagreements can be masked and need not corrupt the output of the stage. The system is therefore more reliable than comparable single version systems, which have no means of error detection and masking.

The SAVE system is run from a trusted computing base, like that specified by the Trusted Computing Group [73], which provides secure private storage, process isolation and attestation. These features allow each SAVE module to prove its own identity and protect its secrets including its private key and data. Attestation of each module allows other modules to be relatively secure in their expectations about the module's function as well as in their grants of protected bits of knowledge to the module. This is necessary for correct and verifiable SAVE operation.

3.1 SAVE Components

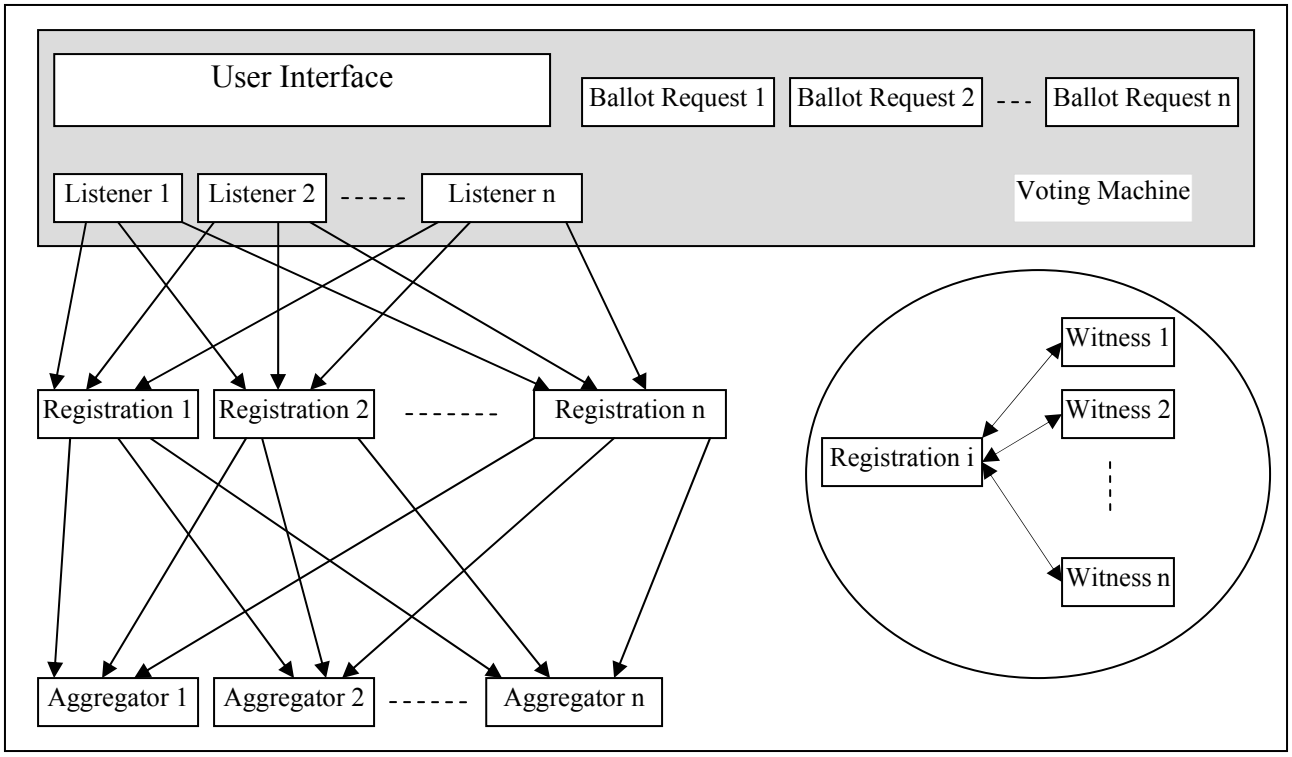

Figure 3.1: SAVE System

There are six main types of Modules, each responsible for performing some part of the voting system's function. Figure 3.1 depicts the relationship of these modules within the SAVE system. The **Ballot Request** Module uses information about the precinct and the election to request a suitable ballot and HAVA [74] compliant user interface display specification from designated ballot servers. The **User Interface** Module is responsible for displaying all screens necessary to inform the voter and facilitate voter input. The **Listener** Module's task is to capture voter input from the User Interface display, and transmit the collected information to all relevant parts of the system. The **Registration Module** is responsible for attesting valid voters and their ballots while rejecting invalid voters. The **Witness** Module creates an auditable and secure record of each vote. Finally, the **Aggregator** Module is responsible for establishing the vote recorded by the system for each ballot cast, and securely transmitting that anonymous, completed ballot

to any applicable third party storage. The Aggregator Modules may also establish an actual outcome for the election.

3.1.1 Ballot Request Module

The Ballot Request Module is a simple, but important Module, responsible for requesting the blank ballot and user interface display specification for its associated voting machine. The HAVA [74] compliant user interface display specification specifies the style in which instances of the ballot should be displayed at the User Interface. Each user interface display specification is carefully designed to best accommodate the special needs that some voters may have. Further, the user interface display specification is designed to satisfy any usability restrictions imposed by the jurisdiction in which the voting machine is being operated. The Ballot Request Module uses the election and precinct information, provided to the voting machine, to request the ballot and user interface display specification from designated ballot servers. Both the Ballot Request Module and the ballot server are authenticated, using their public keys, before the ballot and user interface display specification are transferred.

3.1.2 User Interface Module

The User Interface is vital to any voting architecture. The relatively poor user interface historically provided by paper ballots has caused confidential ballot casting to be inaccessible to many voters with disabilities [5]. Additionally, user interface problems are arguably responsible for most of the spoilt ballots that are generated during elections [2-3, 7]. The modularity of the SAVE design is particularly useful and appropriate at the user interface because it allows the Module to be updated easily. Another advantage of the SAVE architecture is that the User Interface Module completely separates content from presentation style. Thus the same ballot can be presented in the best style available for a voter, without having to recertify software, reinitialize voting machines, or undergo any of the lengthy and cumbersome processes that would be necessary in monolithic systems.

The SAVE User Interface Module is responsible for communicating relevant information to the voter in the most accessible way possible [74, 5], so that the voter is able to understand, and contribute accurately to, the entire voting process. Specifically, the User Interface Module is responsible for prompting the voter for relevant input and generally guiding voter interaction with the system. On initialization of a voting machine, the associated User Interface receives, from the designated ballot servers, independent copies of the blank ballot and the user interface display specification that will be used for the election. The official blank ballot and user interface display specification is elected from these inputs. The elected input is used by that voting machine throughout the election. The User Interface Module will generate a unique instance of the official blank ballot, for each voter that interacts with that voting machine during the election. The blank ballot is displayed or otherwise communicated to voters, as specified by the user interface display specification.

The User Interface Module must also facilitate ballot completion. The User Interface Module ensures that the mechanism by which the voter makes her selections and submits them is clear and obvious. Additionally, the User Interface Module must ensure that the user is informed if her ballot is not cast, for example because the Registration Modules have determined that the user is not eligible to vote. The User Interface Module is also responsible for facilitating user verification of the ballot the system has received, before it is officially cast, as well as user correction of that ballot, including obtaining a new blank ballot as per section 301 (a)(1)(A) of HAVA [74]. The User Interface receives versions of the ballots tentatively stored by the Aggregator Modules and chooses an official ballot from among them. The official ballot represents the ballot that the system believes the user intends to cast, as interpreted by the Aggregator Modules. The User Interface displays this ballot and allows the voter to indicate whether or not it is correct. If the voter verifies that the displayed ballot is correct, this approval is relayed to the system and the ballot is cast. If the voter indicates that the displayed ballot is incorrect, the User Interface asks the voter to indicate whether she would like to begin the process again, with a new blank ballot, and displays that new ballot if requested. In this way the SAVE

system provides voter verification as well as second-chance voting as required under HAVA [74].

A single User Interface Module is not allowed to decide which content is displayed to the voter; this would be a critical point of failure. Instead, each User Interface module must communicate with the others so that they can collectively agree on the content that should be displayed. The collective agreement is done based on one of the typical distributed consensus algorithm chosen by the developers [75]. More than some specified majority of User Interface modules must sign the agreed-upon content to indicate their agreement, before that content is displayed to the user. Though it has not yet been implemented, we intend to implement a system that can detect display errors. For example, we might implement a system of multiple diverse display driver monitors that would analyze the display to make sure that what is displayed is actually what was elected for display by the User Interface modules. Then, if more than some specified majority of driver monitors indicate that the display is incorrect, we would announce that there is a problem with the driver or display and steps can be taken to, for example, abandon the use of that voting machine. We plan to add redundant drivers in the future as we seek to increase the redundancy at lower levels of the system, including the operating system and hardware levels. Unfortunately there is currently no way to correct a problem with the single display, but this system is still an improvement over systems with no error masking. Display errors, like content or format errors, can be detected with high probability, according to the N-version model.

3.1.4 Listener Module

The Listener Module listens for user activity at the machine's I/O interfaces, captures all user input, and sends the user input to all relevant parts of the system. The Listener Module's first duty is to capture the ballot selections the voter makes. When the Listener Module captures a voter's filled-in ballot, it must next verify that the registration token presented by the user is authentic and meant for use at the precinct in which the voting machine operates. The token's content is digitally signed by its manufacturers and certifiers, and this signature is verified. Additionally, the election id and precinct id

provided on the token is checked to make sure that it is consistent with the election and precinct id that was provided to the voting machine on initialization. This ensures that the voter is at the correct precinct and will vote in the correct election. Once the token is authenticated, the Listener Module extracts the user's encrypted identifying information from it. Note that the user's identifying information is encrypted with the public keys of the Registration Modules. This precaution ensures that the Listener Module does not have access to both the clear-text completed ballot and the voter's identity; therefore the Listener Module is unable to create a receipt. The Listener Module encrypts the completed XML ballot using the public keys of the Aggregator Modules and sends this, along with the encrypted voter registration id information, to the Registration Modules for further processing. The Registration Modules check to see whether the voter is valid and, if the voter is authorized, they send the ballot along for further processing.

When the User Interface Module presents a validated ballot to the voter for verification, the Listener Module is responsible for communicating to the rest of the system whether the voter confirms that the presented ballot is completed as intended, or reports an error. If the voter verifies that the ballot displayed is what was intended, the Listener Module communicates this approval to the Aggregator Modules, which then officially acknowledge and store the ballot. The Listener Module also communicates this approval to the Registration Modules so that they can record that the voter has completed her interaction with the system. If however, the ballot is not what the voter intended to cast, the Listener Module captures the type of error indicated by the voter, and sends the report to be stored for audit purposes. The possible errors are human error due to mistakes by the voter or system error. The Listener Module sends the faulty ballot, along with indication of its faulty status, to the Aggregator Modules so that they can invalidate the faulty ballot. If the voter requests a new ballot, the Listener Module relays this request to the User Interface Module.

3.1.5 Registration Module

The Registration Module has access to the roster of all registered voters and manages the registration data and check-in procedures for the election. The Registration Module

receives versions of the encrypted ballot and voter identification information as input from Listener Modules. The Registration Module selects an official version, and examines the voter identification information from that version to see whether the voter is valid. If the voter is valid the Registration Module signs the elected encrypted ballot. A valid voter minimally is listed in the registration roster and has not already cast a valid ballot in the election at the time when the check is performed. The Registration Module relies on the Listener Modules to relay when the voter has cast her vote by approving a ballot presented for voter-verification. The Registration Module is also responsible for checking that any other requirement for validity is fulfilled before authorizing the voter and signing the ballot. The Registration Module's signature represents its attestation that the voter was authorized to vote and that the ballot was valid, at the time of signing. Once the voter is authorized, the voter's registration id and any other identifying information are completely severed from any association with the encrypted ballot. If the authorized voter has not yet been entered into the check-in database, the Registration Module checks-in the voter by making an entry in the check-in database. This is necessary only on the voter's first attempt to cast a ballot, as the Registration Module would not have previously encountered that voter

Only the encrypted ballot, devoid of any voter identifying information, is sent further into the system for processing. The Registration Module sends the encrypted ballot and its digital signature to the Witness Modules. Each Witness Module checks the Registration Module's signature and, only if the signature is valid, responds to the Registration Module with its own Witness signature. When the Witnesses return their signatures, the Registration Module appends their signatures to the encrypted ballot. Finally, the encrypted ballot and all appended signatures are sent to the Aggregator Modules. As indicated, the Aggregator Module receives no voter identifying information.

If the voter approves a completed ballot presented to him for verification, then the Registration Module receives notice from the Listener Modules, containing the voter's identifying information – encrypted with the Registration Module's public key – and a confirmation that the voter should be finalized. If a more than some specified majority of

Listener's claim that a particular voter should be finalized, the Registration Module records that voter as "finalized" in its databases, and the voter is no longer able to vote.

3.1.6 Witness Module

The Witness Module is a simple Module that takes as input a signed encrypted ballot from a Registration Module, attempts to verify that the signature indeed belongs to the sender Registration Module and, if successful, hashes the encrypted ballot and produces a digital signature using its private key. Witnesses do not maintain a record of the ballots coming through them, as they are meant to be lightweight implementations. The Witness Module signs the encrypted ballot to attest that the ballot, as well as the voter who cast it, have been deemed valid by a Registration Module. More than some specified majority of Witness Modules must sign every ballot in order for the ballot to be deemed valid. Thus the Witness Module provides additional verifiability, if the Registration module later becomes corrupted for example. Witness Modules may be provided by independent organizations such as political parties and watchdog organizations. Witness modules would then help increase the trust that those organizations place in SAVE.

3.1.7 Aggregator Module

The Aggregator Module has the important task of making the final decision about which ballots should potentially be counted. The Aggregator Module receives encrypted ballot packages from Registration Modules. Each encrypted ballot package contains an encrypted ballot, as well as the Registration Module signature and Witness Module signatures of the encrypted ballot. The Aggregator Module selects the encrypted ballot that occurs an absolute majority of times among these ballot packages as the potentially official encrypted ballot. All the encrypted ballot packages containing that encrypted ballot are collected and examined to determine whether or not the ballot should be accepted and tentatively stored. The ballot is accepted if set thresholds of Registration Module signatures and Witness signatures are valid. The thresholds must be such that the Aggregator ensures that there are valid signatures for at least a majority of the Registration Modules and Witness Modules listed in the Aggregator's copy of the directory. If the ballot is deemed valid, the Aggregator Module decrypts the ballot,

parses the plain text of the ballot and tentatively records the selections. Those selections represent the vote tentatively recorded by the system for that voter

At this point, the Aggregator Module must send the ballot back to the User Interface so that the voter can verify the vote, recast the vote, or cancel the voting process as necessary. This feature is required for the SAVE system to be HAVA compliant [74]. To ensure that the ballot presented to the voter is indeed the ballot that would be stored, the Aggregator Module queries its own storage for the relevant selections and reconstructs the ballot from this information. The reconstructed ballot is then encrypted with the User Interface Module's public key and sent to the User Interface Module. The Listener Modules would then capture the voter's response to the presented ballot and relay this response to the Aggregator Module along with the ballot itself, again encrypted with the Aggregator Module's public key. The Aggregator elects one of the Listener messages as the official input. If the official input indicates that the voter has rejected the ballot, the Aggregator Module discards the ballot and rolls back its tentative storage of the corresponding selections. If the official input indicates that the voter approved of the ballot and the ballot attached is identical to the ballot that that Aggregator Module sent out, then the tentative storage is committed. Additionally, both the encrypted and plain text versions of the ballot are transmitted to designated authenticated counting or storage servers as well as stored in the SAVE system's read-only repository for auditing purposes.

3.2 SAVE Architecture

The SAVE Architecture is designed to be both complete and viable, so that it is directly usable for practical purposes. We now list the assumptions that SAVE depends on, and analyze the SAVE system given these assumptions.

3.2.1 Design Assumptions

A major assumption made is the N-version assumption that, for an N-version system, it is highly unlikely that most of the redundant versions of a link or module will fail in the same way in any particular situation. Given this assumption, it is likely that the majority

of modules are correct if they all return the same result. Given this, SAVE can achieve greater availability, correctness and security than single version systems, because a few faulty modules or links are not enough to affect the overall result. This robustness has both reliability and security implications because the corrected faults may be due to errors, external hacks or malicious programming.

The SAVE system also assumes that a trusted computing base, such as that specified by the Trusted Computing Group [73], can be securely implemented and used in reality. This assumption allows the SAVE modules to depend on the TCB to provide secure private storage, process isolation and attestation. Because of these features, modules can confidently declare their identity and keep their identity safe from fraud, and can also protect their secrets, including their private keys and the sensitive data that they operate on. Maintaining module identity and secrets are crucial to the operation of the SAVE system.

Additionally, the SAVE system assumes that external parts of the voting process are correct and secure. This includes the assumption that the registration database is correct, as well as the assumption that the registration tokens have been delivered safely to voters and have not been stolen.

SAVE also assumes that it can detect whether a malicious module is capable of sending messages to other modules that contain stage secrets. We assume that no such malicious module becomes certified. Thus we assume that vertical collusion involving the exchange of secrets within messages does not occur during SAVE execution.

SAVE assumes that cryptographic systems are hard to break. We base our confidence in the attestability of our modules and the secrecy of our messages, for example, on this assumption.

3.2.2 The Extent of Redundancy

It is useful to consider just how much of SAVE is redundant and where the limitations are. The voting machine typically has single I/O devices, and this is important for providing a simple, familiar user interface for voters. For example, there is a single keyboard and single display screen. We are able to detect errors with these, however, by using a combination of driver monitors and user verification. Additionally, if an error is detected with the screen or keyboard, it may be possible to exchange these devices for other, certified replacements. Currently, the voting machine's software is not redundant, except for the voting modules described previously. Many software errors here would not be detectable under the current conditions. Future work will involve creating a redundant trusted computing base that could eliminate more of these points of failure.

Note that the SAVE system is capable of counting ballots and presenting an election result, however this is not part of the typical SAVE function. Thus one need not be concerned about the single point of failure that would potentially arise from the reporting of such a result. The ballots are stored independently in diverse and separated Aggregator modules, and counts may be extracted simultaneously from all the Aggregators, with as many observers as desired. In this way, the ballot storage, and resulting election result calculation, need not be a single point of failure.

3.2.3 System Initialization

Poor system initialization can make it impossible for a system to function accurately or securely. As such, correct system initialization is very important. A major component of SAVE initialization is establishing each module's critical data. A SAVE module's critical data consists primarily of its private key and its directory of other authentic SAVE modules and their corresponding functions, addresses and public keys. The directory's correctness and authenticity is essential to system integrity. The directory contains no secrets, so directory exposure is not critical. However, it is important to minimize the likelihood that the eventual directory is corrupted, by being flooded with fraudulent module descriptions for example. Such fraudulent modules reduce the reliability of the

system, and could cause even more damage, by using their access to the system to launch attacks.

Modules cannot construct their directories for themselves. This is because software cannot trust remote software, communicating through XML messages, without some sort of authentication, such as a signature that proves current possession of some pre-determined secret. Without such a proof, any remote interaction sequence is forgeable. However, if a signature is used, the public key must be known to the distrusting software. There must be some root-of-trust that provides the first public keys to the distrusting software. To avoid placing our trust in few external roots-of-trust which we do not know very well, we create our own root-of-trust consisting of any suitably sized subset of N human principles. The public keys of these human principles are known to all modules and the modules will trust any majority of them. Modules that have been deemed correct and have been designated for inclusion in the final SAVE system have "approval" certificates to this effect, signed with the private keys of these trusted human principles. The authenticity of a module and its messages is attested by the Trusted Computing Base, which knows the module by its source code and public keys. These human principles also independently verify and endorse special Directory Generation (DirG) modules, whose function is to collect the public keys, function, and other information associated with each module and compiling that information into a directory. Each endorsement takes the form of a signed certificate containing that Directory Generation module's public key.

Each SAVE module contributes its own information to the generation of the collective directory. When a SAVE module is started for the first time, it generates its private keys and reports the corresponding public keys to the Directory Generation modules that a majority of human principles have collectively told it about. If the SAVE module can prove that it has been granted an "approval" certificate, and the DirG module can prove that it has also been endorsed, the module presents its directory information to the DirG module and the DirG module approves the SAVE module and its information. A DirG module approves a SAVE module by generating a certificate, if none exists, which

includes that DirG's signature of the module's public keys and other information, or by adding it's signature of the module's public keys and information to the already existing "directory" certificate. A "directory" certificate is deemed valid once it contains signatures from more than some specified majority of DirG module signatories. In addition to reporting public keys, each module may also report, for example, its host and ports the module wishes to associate with itself. A DirG module tests such ports by sending "proof-of-association" requests to the claimed ports. To verify association with that port, the module must respond to a "proof-of-association" challenge with a live signed message which contains the challenge and an appended time stamp.

The DirG modules then compare the directory information they have collected and merge their collections into a single address-book. Any typical distributed consensus algorithm may used [75]. Note that only information that was independently collected by more than some specified majority of DirG modules is included in the official directory. This prevents fraudulent or faulty DirG modules from including modules, or module information that should not be included. Note that the DirG modules gain very little secret information, even if they are fraudulent, because each module uses fresh keys for its proofs and interactions with the DirG modules; these keys are attested by the TCB. The official directory is digitally signed by all the DirG modules that have agreed to it, so that it cannot be modified without detection, and then distributed to the SAVE modules listed in the directory.

3.2.4 The Voting Process

The voter's interaction with the system begins when the voter receives a registration token by secure mail. This token contains the *precinct id* of the precinct that the voter is registered to vote at, the *election id* of the election that the voter is registered to vote in, as well as encrypted voter registration information. The token may be, for example, a read-only, copy-resistant compact disc. When the voter arrives at the polling station, a poll worker verifies that the voter is who she claims to be and that the registration token is valid and is associated with that voter's identity. If the voter is valid, the person is allowed to enter a polling booth with the registration token. There are further schemes

available to reduce the opportunity for fraud with regards to the token. These include sending only part of the voter's token by secure mail, and having the voter collect the other part at the polling station, after proving his or her identity. The parts are fragmented so that the information on each part is unusable unless it is combined with the other. This scheme would however increase the inconvenience faced by the voter and the administrative difficulties faced by the poll workers.

On startup, each voting machine requests a blank ballot and interface definition from ballot servers, through its Ballot Request modules. The ballot servers only service these requests during the official voting period. The voting machine then awaits the arrival of voters. The voting process depicted in Figure 3.2 is carried out for each voter.

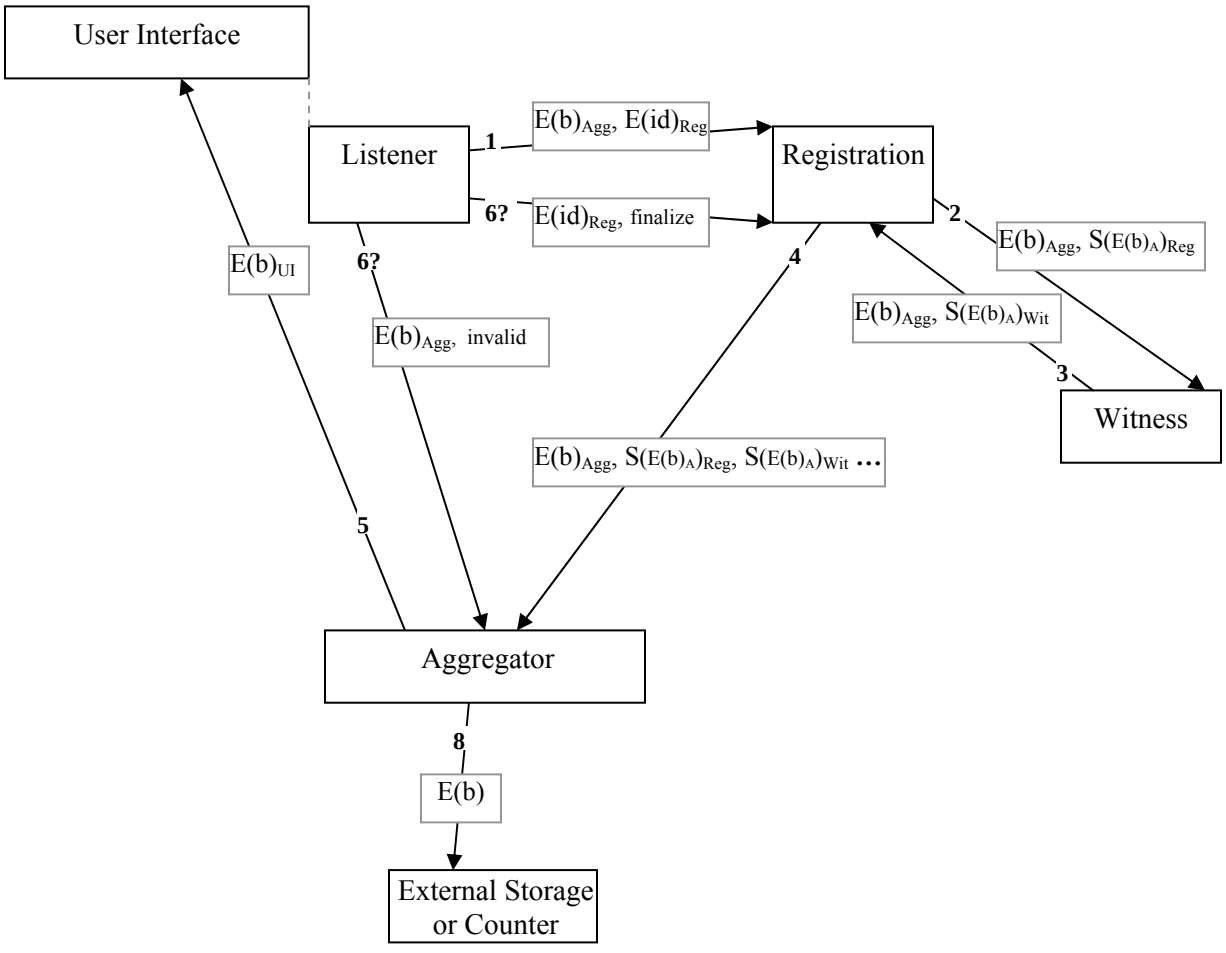

Figure 3.2: Voting Process

When a voter places his registration token into the voting machine running SAVE, the User Interface module displays an instance of the blank ballot to the voter. The instance is distinguishable from other instances by a ballot id generated by the User Interface module. The voter makes his desired selections on the ballot, submits the ballot and waits to verify the ballot actually recorded by the system. The completed ballot is compiled and encrypted by Listener modules, and the encrypted data is sent to the Registration module along with the voter's encrypted registration information as extracted from the registration token. Once the ballot is encrypted, its processing is mingled with the processing of other ballots, from other User Interface modules on other voting machines. Only the User Interface, Listener and Ballot Request modules need reside on the voting machine; other modules are distributed among other connected but distinct computers.

Each Registration module receives messages from Listener modules. Each such message contains encrypted voter registration information and the separately encrypted secret ballot. The Registration module elects an official message from these. The Registration module then decrypts the voter's registration information contained in the official message and checks that the voter is authorized to vote and has not yet cast a valid ballot. If this is the case, the Registration module checks-in the voter, if the voter has not yet been checked-in, and signs the encrypted ballot. The Registration module then sends the encrypted ballot and signature off to Witness modules to be digitally signed. Finally, the Registration module sends the encrypted ballot and all signatures to the Aggregator module. The Registration module never knows the contents of the encrypted ballot, since it is encrypted with an Aggregator module's public key.

Each Aggregator module determines the validity of the encrypted ballots it receives based on the proportion of valid signatures appended to that ballot. If the ballot is deemed valid, it is decrypted and both the encrypted ballot and the voter's selections as expressed in clear-text version of the ballot are tentatively stored in the Aggregator module's database. To facilitate complete voter verification, the Aggregator module then extracts the voter's selections from the database and recompiles them into an XML ballot. This

reconstituted ballot is encrypted with the User Interface module's public key and sent to the User Interface. In this way, the ballot that the user verifies is exactly the ballot that would be cast by the system; errors at any level of the system, including at the database level, will be detected.

The ballot tentatively stored by the system is presented to the voter. The voter may indicate that the displayed ballot is correct. In this case, the Listener modules relay the approval to the Aggregator modules, who permanently store the ballot, and the Registration modules, who record that the voter has completed voting. The voter's approval at this stage ends the voter's interaction with the system; the vote is officially cast. The voter may alternatively indicate that the displayed ballot is not the ballot that the voter intended to cast, that is, indicate that either the voter made a mistake or that there was an error in the system. If the voter indicates that the displayed ballot is not what was intended, the Listener modules relay this disapproval to the Aggregator modules who rollback their tentative ballot storage and discard the ballot. The type of error, as perceived and indicated by the voter, as well as the faulty ballots may be stored for audit purposes. If the voter indicates that there was a problem with the ballot, the voter may terminate the voting process without voting, or may request a new blank ballot. If the voter chooses to terminate the voting process, the voter's interaction with the system ends. On the other hand, if the voter requests a new ballot, the Listeners relay this request to the User Interface module and the voting process is repeated.

3.2.5 Audit Trails & Recounting

For each ballot cast, the n Aggregator modules store the original ballot package, including the encrypted ballot and digital signatures, as well as the clear-text of the ballot on read-only media. Additionally, this data may be stored on authenticated, external read-only storage set aside for maintaining audits. The ballot packages are stored on different computers by different Aggregator modules. Thus it is expected that there will be a majority of accurate, signed (and thus non-modifiable) electronic copies of each ballot cast, excluding those stored at faulty storage or by faulty or malicious Aggregator modules. When an official audit is requested, the audit common to the absolute majority

of storage devices is chosen, if such a majority exists. Consequently, the errors of a few will not affect the resulting official audit ballot. This form of auditing is highly robust because it can withstand attacks, errors or losses at many of the Aggregator modules or storage points, while maintaining the validity of the result. Conversely, a single-version paper audit would not be able to withstand such failures; if such a failure was to occur, the audit trail would be completely lost. Additionally, this form of electronic auditing is verifiable because the state that led to the Aggregator modules final approval of the ballot, i.e. the digital signatures of the Registration and Witness modules, are also stored. Assuming that no registration tokens are stolen, it is impossible to cast a fraudulent ballot without corrupting a majority of Registration modules. Further, the Registration and Witness modules responsible for certifying each ballot can be traced from their digital signatures and investigated if necessary.

With electronic ballot verification, we can communicate the ballot that the system has stored for a voter in ways that are accessible to all persons, regardless of language or disability. So, after ballot verification by the voter, it is exceedingly likely that the electronic ballot stored as an audit is what the voter intended. This is not the case with paper verification, which many disabled voters would be unable to interpret on their own. Additionally, there are no transport costs associated with storing or accumulating the audit trails. Counting or other checks on the audits can be done quickly and efficiently using software. Further, the accuracy and reliability of these audit checks are greatly improved by the use of N-version software, as opposed to single-version software or manual checks.

True recounting is possible with these audit trails. The count can actually be redone, rather than just re-reported, because each ballot is stored individually. We have the actual selections that each voter made, so we can recount the ballots in a number of ways: including passing the encrypted ballots and their associated signatures back through our SAVE counting system or through another diverse implementation of the SAVE counting system or hand-counting the actual selections made on each ballot, as reported by a majority of Aggregator modules.

3.2.6 HAVA Act compliancy

The Save Architecture complies with the Help America Vote Act (HAVA) of 2002 [74]. Multimedia communication is used to ensure that the system is accessible to all voters, regardless of their language or special needs, in such a way that the voter's entire voting experience is private and independent and the voter's ballot is confidential. Also, the voter is provided with the opportunity and the information needed to verify and correct his ballot before the ballot is cast. In particular: the voter is allowed to modify his selections on his existing ballot at any time before it is entered into the system; the voter is allowed to verify the exact ballot that will be cast, after it has passed through the system; and the voter is allowed to void any ballot that has not yet been cast and receive a new ballot for a "second chance" at voting. The User Interface also restricts the errors that the voter can make by preventing overvoting. Each ballot and its related signatures are stored and can be used as electronic audit trails or printed as paper audit trails. The error rate of the SAVE system is measured and will be made to comply with the error rates standards issued by the Federal Election Commission. Also, the Aggregators are able to incorporate any standard definition of what constitutes a vote in its decision on whether each ballot is valid.

3.3 SAVE Security Features

The SAVE system is centered on redundancy and diversity. These principles also contribute to the security of the SAVE system. While there has been some application of diversity and redundancy to security [76, 80], our system, as an N-version voting architecture, is one of the first voting systems to use it so fundamentally and extensively. There is some cryptography involved in keeping SAVE safe from attacks. In particular signing and verification of digital signatures for authentication, as well as encryption for confidentiality play critical roles in SAVE security. However SAVE relies much less on cryptography than other voting schemes [28-32]. Reduced reliance on cryptography and complex algorithms make SAVE simpler to implement and review, which in turn reduces the likelihood of bugs and increases the likelihood that bugs and maliciousness can be detected. The modularity of SAVE allows the system to take advantage of a combination of methods for review that establishes system security, including limited expert review

and open-source review. Each module can be reviewed separately, in the style more suited to that module, without compromising the integrity of other modules. SAVE can therefore benefit from some of the advantages of both methods [77-78]. This is in keeping with N-version testing notions [38].

3.3.1 The Trust Model

The trust model is small and simple to describe, yet its implications are powerful. Principally, each module trusts itself and trusts that no majority of modules which implement the same function shall collude, fail, become disconnected or express an inaccurate result. A module places no trust in any minority of its counterparts; a minority group may consist of erroneous, malicious, hacked or slow software. A module does not trust another module to respond correctly, quickly, or even to respond at all.

3.3.2 The Threat Model

As with any voting system, the SAVE system must be able to make certain essential guarantees. In particular, we must guarantee that no one is able to produce a receipt that ties a voter to the ballot he cast, so that no one can be forced or enticed into revealing this information. We must also ensure that each voter that casts a ballot actually has cast the ballot that the voter intended and that the ballot is counted. We must ensure also that our system produces correct results overall at every stage. We assume that a variety of serious attacks are possible from internal developers, election officials, voters or external hackers. We have taken steps to prevent these.

Insider Attacks

Module developers may attempt to introduce malicious code that might, for example, modify, delay or delete votes, incorrectly report tallies, or flood the system so that valid operations are delayed or unable to take place. Such a threat could also come from any underlying hardware or software libraries used by the system. Additionally, module developers or other persons might seek to attack the voting machine's hardware or software.

Each module is compiled on several different compilers with the majority result chosen as the official executable. This mitigates the possibility of bugs or Trojan horses from a few compilers compromising the election. Additionally, by the N-version assumption, a few modules producing an incorrect result, because of maliciousness or error, has no effect on the overall results of the system. Further, modules that produce minority results have their result ignored and their faultiness noted. We also assume that it is highly unlikely that a majority of module developers, implementing the same function, will collude in such a way that their modules produce the same incorrect result. Thus it is unlikely that the behavior of the system as a whole would be swayed by malicious modules. Also, the modules at each stage do not, by themselves, have any useful information, and also are not guaranteed that their information will influence the final result. This reduces the incentive of malicious developers to expose their module's data. Additionally, only some fraction of the modules developed for each stage are actually used in the SAVE system. Therefore a module developer never knows for sure whether his module will be used.

No module can bring down the system by corrupting or otherwise attacking critical data. Each module has its own copy of the critical data it needs to execute correctly and securely and recover from errors. This copy is well protected by the secure and curtained memory that the TCB provides to each module. Further, the TCB protects its own critical data. Thus the only critical data that a module has access to is its own and the system does not depend on a single module's operation. The other modules' behavior is not affected and thus the system can recover.

Each module must receive a command from a majority of modules in the previous stage before that command is carried out. Thus it is impossible for a few modules to spoof the system, by causing other modules to perform tasks that they would not have done if the execution was proceeding correctly. Similarly, it is impossible for a module to learn another's vulnerabilities by sending arbitrary requests to that module and observing responses. Those requests will be ignored because they are in the minority.

Since a module cannot access another's memory, or cause another module to perform operations, it would be very difficult for a malicious module to get a correct module to behave maliciously. Even if a module sent an incorrect input to another, that input is not likely to be elected and operated on, and therefore the receiving module should be able to recover from that incorrect input. The system therefore has the "fault containment" property.

It is therefore very difficult for malicious modules to affect the system. This fact reduces a developer's incentive to introduce malicious code.

System installers may attempt to steal module secrets or provide modules with misleading and malicious startup information. For example, a system installer might attempt to steal a module's private key or add fraudulent information to a module's directory. We attempt to counter this threat in various ways. Firstly, the modules never reveal their secret keys, not even to system installers. We trust that the Trusted Computing Base is able to correctly generate key-pairs and protect the private key of each module. As such, it is very unlikely that a system installer would gain access to a module's private key. Secondly, we require the agreement of many system installers before any bit of information is approved for inclusion in the modules' critical data. Thirdly, the module versions its critical data, for example its directory, as soon as it is installed to prevent modification.

Voting machines and their installed software are also vulnerable, both during development and during the actual election. We keep the voting machines as physically secure as possible at all times, to prevent attack to the hardware. Also, we sign and version all the module executables, and protect software using access control lists, and data encryption. In this way, we can verify what software actually runs on the machine; we can limit who has access to software and data; and we can record who exercises their authority to access. These steps help protect the system from both internal and external hackers.

External Hackers

We assume that with enough experience and time, outside hackers could attempt to gain access to messages between system components, impersonate system components, or flood system components so that they are unable to service valid clients.

The network operates behind a firewall, and modules service requests from pre-determined, authenticated clients only. Further, the modules are distributed, operating from many different hosts and ports; it is therefore unlikely that an attacker will be able to attack and bring down enough modules to affect the system overall. Registration and Aggregator databases are also backed up independently so that recovery, in the case of crashes or hacks, is possible. In this way, no votes that have already been cast, are lost if the system goes down. This helps mitigate the effects of denial of service attacks.

Communication between modules is done using authenticated SSL, so an external hacker is effectively unable to view messages or send impersonating messages or successfully replay old messages. Thus an external hacker is unable to control the messages that arrive at a host or impersonate system components. Only static data is passed between modules. This minimizes the possibility of external hackers gaining control of system components that are appropriately defended against buffer overflows and the like.

Attacks that cause a module to become malicious still have little effect on the system. This is because the system is resistant to malicious modules, as described in the previous section.

Sensitive decisions, for example which modules will be included into the final SAVE system, are made as close as possible to the election time, so that outsiders will have less time to discover them. Additionally, the software is reviewed by trusted experts, but kept secret from outsiders, for the short time between System completion and the election. We do this because we do not currently trust all members of the public who might find exploits to appropriately report it. The System is not completely dependent on obscurity

however. Though exposure of the source code to trusted experts may (or may not) reveal vulnerabilities, it does not automatically give those experts any control over the system.

Corrupt Election Officials

We assume that a corrupt election official could attempt to gain access to a voting machine's data or software and use this access to discard or modify valid ballots or insert fraudulent ballots into the system.

Multiple election officials are at hand to observe each other during every task. This minimizes the chance of an attack, especially since it is unlikely that all of the observing election officials would collude in such a way that allows illegal attempts to tamper with the voting machine. Additionally, any attempted modification of a ballot before it is cast will be detected and corrected because of the voter verification of each ballot. Also, any attempt to modify a ballot after it has been signed by the Registration and Witness modules would only render the signatures invalid and hence invalidate the ballot itself. Ballots can not easily be modified after they are cast because ballots are stored on read-only media. Also, because each official ballot must be elected by majority from multiple ballots at different computers, the compromise of the ballots on a few computers will not affect the official ballot corresponding to any particular vote. Discarding stored ballots at a few storage repositories, even if this could be accomplished, would not truly remove the ballot from the system or stop it from being counted. The system is thus safe from the threat of data or functional integrity compromise.

The modules are run from a trusted computing base, and they never share their secrets. Thus no introduced software can impersonate a module because it would not have access to the private key of any of the pre-listed authenticated modules. It is virtually impossible to insert fraudulent ballots because the intruder would have to corrupt or steal the private keys of a majority of Registration modules in order to get their signatures to validate the fraudulent ballot. Ballots cannot simply be copied because each ballot has a unique identifier.

Finally, there is no difference between a test vote and a real vote, as far as the software is concerned. So there are no back-doors, or other problems arising from such distinctions, that can be exploited.

It is therefore highly unlikely that votes can be removed, modified or forged.

Malicious Voters

We assume that malicious voters may attempt to vote multiple times, or vote as another person, or sell their votes.

The SAVE system prevents multiple votes because the Registration modules keep track of all voters who have cast their ballots and would not sign a ballot for someone who has already voted. Registration module signatures are needed for any ballot to become official. A malicious voter cannot vote as another person unless he has stolen that person's registration token and managed to convince the authenticating poll worker that he is that other person. The registration tokens are securely sent to the correct persons and the poll workers must satisfy the jurisdiction's authentication requirements before allowing the voter into the voting booth. We are thus satisfied that it is unlikely that a voter can commit the identity theft necessary to vote as another person. No module can impersonate a voter or his choices unless a majority of modules at that stage collude. No module can expose voter's identifying information to another because a majority of UI modules have to agree to display that secret, and the correct modules will not do so. None of the internal modules have any control over the I/O and no voter can access the machine's file system or memory, so that it is impossible for a voter and a module to exchange secrets.

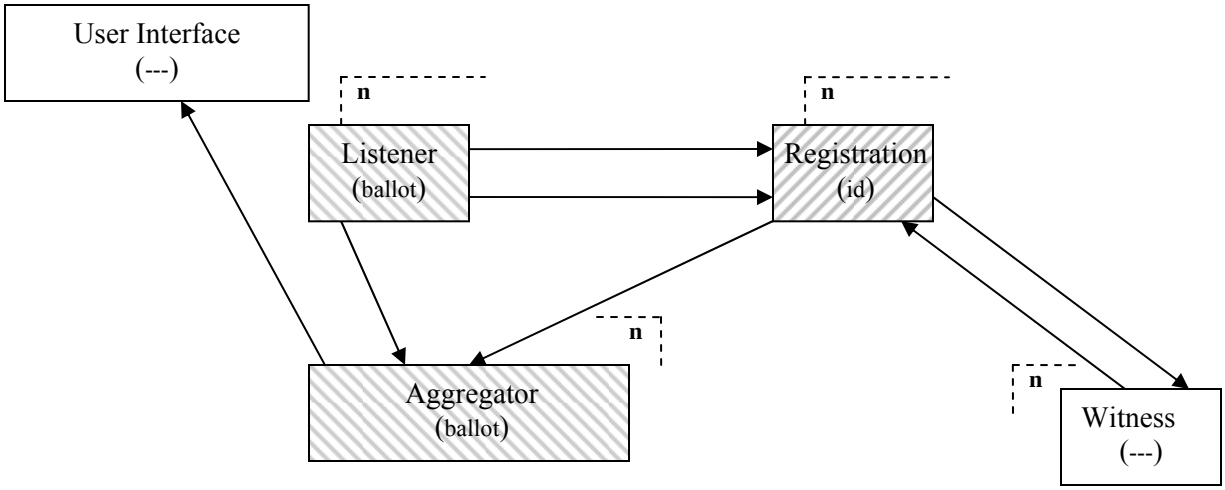

Figure 3.3: Knowledge Separation

The SAVE system uses encryption and blind-signatures to avoid receipt-creation. There is also clear separation of knowledge within the SAVE system, as indicated in Figure 3.3. The User Interface module and the Ballot Request module do not know the identity of the voter or the voter's selections. The Listener modules have access to the voter's selections, but do not know the identity of the voter. The Registration modules know the voter's identity but do not have access to the clear-text ballot. Witness module does not have access to the clear-text ballot or the voter's identity. Finally, the Aggregator modules have access to the clear-text ballot but not the voter's identity. The Registration, Witness and Aggregator modules are shared by multiple User Interface modules at different voting machines, so they may encounter encrypted ballots from any of several different voters at any time. The encrypted ballots can be mixed, using a mix-net scheme [79], to further disassociate voter-identity from ballot. This prevents the creation of a receipt or violation of voter privacy, unless there is collusion by multiple, functionally distinct modules. Additionally, these steps help protect the secret information in the system in general.

3.3.3 Security improvement through N-version programming

The architecture of this system uses diversity, redundancy and threshold agreement for fault and hack tolerance. By N-version programming principles, the modules do not rely

on any particular module, and do trust any particular module. Also, the modules have the "self-checks" of single-version systems, and also have the "neighbor-checks" of N-version systems. Thus, intuitively, it is expected that an N-version system should be more secure than its corresponding single-version system. Though we have mentioned the security benefits of N-version programming throughout the thesis and this chapter, we combine them here to make a definitive statement that an N-version system is inherently more secure than its single version counterpart. That is, the security of any single version system can be improved by adding diversity and redundancy to that system, thus creating a corresponding N-version system. N-version principles help the SAVE system to resist the following major attacks: corruption of critical data or state, compromise of module data or functional integrity, compromise of system functional integrity, malicious logic insertion, denial of service and spoofing. It also protects SAVE modules from malicious modules or outsiders that are not currently aware of vulnerability in the module, but hope that by making requests and viewing module's responses, they would learn information that can be used against the module or the system.

Corruption of critical data or state

All system critical data is redundant – each module owns a copy of all the information it needs to execute correctly and securely and recover from errors – and all module critical data is well protected by the curtained storage and secure memory provided by the TCB. Additionally, the TCB protects its own critical data and other secrets. Thus no module or software component can bring down the system by corrupting or otherwise attacking critical data.

Compromise of Module Data or Functional Integrity

Redundant modules perform the same operation on the same input, and so the correct output can be determined by majority, even if a few modules are hacked, produce an incorrect result, or otherwise fail. Also, multiple channels will communicate the same majority result, so that the receiving module is able to recover the correct input even if a minority of channels has been maliciously or otherwise disconnected. Redundant audit trails provide security, verifiability and reliability by making the audits more resistant to

tampering. Modifying or destroying the audit trails stored at a few storage repositories will not affect the overall audit, which is decided my majority on extraction from all repositories. The redundancy and diversity features of SAVE thus contribute significantly to the security of the SAVE system.

Compromise of System Functional Integrity

It is extremely difficult for a hacker or malicious insider to sway the actual output of an N-version system [80]. It is almost impossible to change a person's vote, for example. Changing the vote cast would require careful insertion of faults that create identical failures at a majority of modules' comparison points at every stage after the voter has reviewed his ballot. Further, to successfully change the vote cast, those faults cannot cause those failures at those stages before the voter has reviewed the ballot. The faults for each of the majority compromised modules would have to be triggered after the voter has reviewed the ballot so that the modified ballot will not be detected. It is very unlikely that a malicious insider will have access to a majority of modules. The module source code is carefully controlled, both physically by securing the machine it is on, and virtually with access control lists. Further, it is very unlikely that an external hacker would be able to modify the operation of a majority of modules. Even if the enemy was able to gain access to a majority of modules, experiments have indicated that it is very difficult to force coincident errors to occur in a majority of diverse modules by injecting faults [80]. Thus it is very likely that the SAVE system will cast the vote that the voter intended.

Fault / Malicious Code Insertion

The SAVE system consists of numerous small and simple modules. Each module is a few hundred lines of code at the most, and performs very few functions. As such, validation, verification and review are much easier, and any bugs, malicious code or vulnerabilities are consequently much easier to detect and correct. Each module can be tested and certified independently, and modifications to one module do not affect any other module. Additionally, the security of each stage can be analyzed separately, allowing greater focus which in turn leads to a greater problem detection rate and more

completeness of review in general. The system is in general very maintainable and can keep pace with advances in social and technological research as well as government policy. This has obvious implications for security as malicious code is more likely to be detected and removed, with suitable punishments for the developer. Additionally, the more credible threat of detection reduces the incentive of the developer to insert malicious code. Note that even though the interactions between the various SAVE stages appear relatively complex, the rules governing the interaction of modules need only be certified once. The cost of validating the SAVE process becomes a less significant part of the overall cost as more and more modules are developed. Though the SAVE process becomes relatively stable, new modules may be made as frequently as desired, to increase diversity and minimize the chance that any module becomes compromised as well as minimize the time that any corrupt module, already included, would have the chance to harm the system.

Denial of Service

The modules are distributed, operating from many different hosts and ports. Also, each module receives inputs, across independent channels, from several diverse but functionally equivalent modules. It is therefore unlikely that an attacker will be able to attack and bring down enough channels or modules to prevent modules at subsequent stages from making input decisions; thus subsequent stages are likely to recover. Because of these features, SAVE is particularly resistant to denial-of-service attacks.

Spoofing

No module performs a function unless it has received requests from a majority of modules asking it to do so. A few modules therefore cannot spoof the system; a majority of modules would have to collude to cause the server module to perform a valid task and no amount of modules can cause a correctly implemented module to perform a task that is not listed in the module's list of services. Consequently, it is impossible for a module to learn another's vulnerabilities by sending arbitrary requests to that module and observing responses. Those requests will be ignored because they are in the minority.

3.3.4 Cryptographic Security

A number of cryptographic algorithms, protocols and concepts aid in the security of the SAVE system. All modules are issued their own private keys and are able to keep them safe using a trusted computing platform [73]. Modules use signing key pairs for signing, sealing key pairs for sealing, and communication key pairs for their transmissions. All communication between modules is done using authenticated SSL, so data transmissions are protected from exposure by wiretapping and modules are protected from man-in-the-middle attacks or general replay attacks. Authentication also helps protect against external chosen-plaintext and chosen-ciphertext attacks. Note that, for sealing, message blocks of insufficient size are padded before encryption. Adding random bits to the messages help guard against déjà vu attacks, where dictionaries of plaintext<->ciphertext are accumulated and used to extract secret information about the keys. Also, cryptographic hashing is applied to the message to be signed, before signing, to guard against unintended signatures generated using blinding-based attacks.

SAVE also utilized blind-signatures [81] to maintain privacy and ensure receipt-freeness. Each Registration module receives voter identifying information encrypted with its own encryption public key and completed ballots encrypted with the public keys of Aggregator modules. Thus the Registration module can decrypt the voter's identification, but the contents of the encrypted ballot remains secret. If the Registration module verifies that the voter is authorized, the Registration module signs the encrypted ballot without ever knowing its contents. Thus the voter's ballot is completely separated from voter authorization. The Aggregator module can verify the signatures of the encrypted ballot, before decrypting the ballot and observing the clear-text. The Registration module's signature is known as a blind signature and it allows the system to maintain the privacy and security of the ballots.

The cryptography described is crucial to SAVE's security module, but SAVE also enjoys the benefits of the simplicity that arises from the use of relatively few cryptographic elements and algorithms [35, 37].

Chapter 4

Modeling SAVE's Probability of Failure

N-version software performance and reliability have been traditionally hard to quantify [38, 40-41, 51, 62]. The modules within each n-version software unit differ from each other in particular ways so diversity itself is nonstandard across systems. Experimental results from particular systems are therefore particularly hard to generalize upon. Additionally, it is hard to experimentally measure the extent of diversity between modules and the effect of this particular diversity on the correlation of failures, especially for highly reliable software where very few failures are available for analysis [40-41, 82, 51, 62]. Modeling is especially important to N-version programming because of the limitations of experimental inference and extrapolation of such systems.

We know that failures do not necessarily occur independently [53, 60], so that simpler models based on such independence are not practical. Several more sophisticated models of n-version software diversity, performance and reliability have been constructed, which move away from independent failure assumptions [59]. Early models by Eckhardt and Lee [83] and Littlewood and Miller [55] provided some foundations for n-version modeling. However, these models were designed to measure the mean probability of failure taken over all possible versions that could have been constructed from a particular set of specifications and they are therefore unsuitable for modeling particular instances of n-version software. Since then, Popov, Strigini, et al, have introduced models for particular instances of n-version software, derived from particular component software versions, including a Markov chain model and a Bayesian Inference model [63, 61, 64, 65]. Unfortunately, both schemes are not particularly suited to quick prediction of the reliability of a system before experimental results are available. Such prediction is useful to help guide the actual system implementation. For example, the number of versions that should be implemented per stage could be arranged apriori to maximize the estimates of reliability of the system provided by a suitable model, given cost and performance constraints, if such a model was available. We extend the work of Popov and Strigini [51], so that we can model particular instances of our multi-stage, multi-version SAVE system in a way that facilitates prediction.

4.1 Abstraction of the SAVE System

The SAVE system consists of N versions per stage and M stages. All versions at a stage m implement the same functional specification and are thus expected to perform the same function. However each version is implemented independently, by different developers using diverse development methods, so it is expected that the versions will differ. In particular it is expected that different versions will contain different sets of faults.

The versions at the first stage receive input from outside the system. We assume no lateral communication; versions from the same stage do not communicate directly. Versions from stage m-1 communicate with versions from stage m using imperfect channels. Thus there is a positive probability that a message from a version at stage m-1 may become corrupted in the channel, before reaching the intended recipient version at stage m. In particular, messages may be lost, modified so that they are changed into another legal message, or modified so that they become an illegal message. The outputs of each version at stage m-1, possibly modified by the communication channels, form the inputs to each version at stage m. Each version at stage m attempts to elect an official input from the inputs presented to it, and if successful, performs its specified operation on the elected input. Please see Figure 2.1 for a depiction of input flow through the SAVE system.

A version's decision algorithm function, d(…), is responsible for choosing as the official input for that version, the input that is most likely to be correct. We assume that an input that occurs an absolute majority, $\lceil N/2 \rceil$, of times, is more likely to be correct that any other input present. N here is the number of versions from the previous stage, each of which sends at most one input each round. Thus if an absolute majority input is found, that input is elected as the version's official input and the version's specified operation is then performed on that input. If no absolute majority input is received then no official input is elected and the version performs no operation. The SAVE system versions are expected to provide a correct response to each specified demand. Thus a version "fails" if it produces no output, or an incorrect output, in response to a specified, and hence legal, demand. See Figure 2.2 for a diagram of the abstracted tasks of each version: those

of applying the decision function to its inputs, d(…), then, if the decision function provides a legal, non-null decision, f(d(…)), applying its specified function to the elected input and sending its output to the modules of the next level.

4.2 SAVE Fault and Failure Model

The SAVE fault and failure model applies and extends the fault modeling work done by Popov, et al [51]. In particular, relevant assumptions are added to the model so that the simple "1-out-of-2" multi-version system with its single election, analyzed in [51], can be extended to a full n-version, multi-stage system where each version in a stage faces different inputs and makes its own election decision.

4.2.1 Model Definitions

The following terms are adopted from [51] and help describe the model. Their use also directly helps us to analyze the system.

The *Demand Space* is the set of all possible demands on the system. Demands are relevant, properly formatted inputs that are considered by the decision algorithm as possible candidates for the official elected input. Inputs that are redundant, irrelevant or improperly formatted are discarded and never presented to the decision algorithm for consideration.

A *Design Fault* in a version is an arrangement within the version that causes the version to fail when faced with some set of demands.

A *Failure Point* for a version is a demand that causes that version to fail. A *Version Failure Region* for a version is the set of *Failure Points* for that version.

Additionally, we extend the System Failure Region from so that it is applicable to the SAVE system. The *System Failure Region* is defined as the set of regions where at least some $k \geq \lceil N/2 \rceil$ *Version Failure Regions* overlap; that is, the set of regions where a majority of versions fail.

The Probability of Failure per Demand (PFD) for a version is the probability of the version failing on a demand. Similarly, the Probability of Failure per Demand (PFD) for a system is the probability of the system failing on a demand. This is the probability of receiving a demand in the failure region of the version or system respectively.

4.2.2 Model Assumptions

As with the model definitions, several assumptions are adopted from [51], both for simplicity and for convenience. Also, we again offer extensions or modifications that are necessary in order to apply the model to the SAVE system.

The following assumptions are extracted directly from [51]:

1. *There are a fixed set of possible faults associated with each specification to be implemented. These faults correspond to mistakes that could be made by the specification designers or writers or the software implementers.*
2. *Each fault has an associated failure region.*
3. *Failure regions of different faults do not overlap.*
4. *Each fault F_i has a certain probability p_i of being actually produced in a software version.*
5. *Each fault F_i also has a probability q_i of being 'hit' during operation of the system; q_i is the probability that the system receives a demand in the failure region associated with the fault F_i.*
6. *Mistakes are independent of each other. It is as if each version's team tosses a fair coin to decide whether or not to insert each particular fault so that developers choose, randomly and independently, subsets from the set of possible faults.*

Popov and Strigini are careful to note in [51] that their assumption of independent fault inclusion does not imply that the versions will fail independently or that there are no common factors affecting the mistakes in separate version developments. They clarify

that failure correlation and fault similarities are possible and are modeled by the probabilities of the various sets of faults [51].

In addition to the assumptions listed in [51] and the assumptions added previously, we add the following further assumptions to make the model relevant to the SAVE system:

7. Each fault F_i has a certain probability $p_{i,m}$ of being actually produced in a stage m software version.
8. We modify the assumption regarding the total set of possible faults so that: the total set of possible faults associated with all specifications that describe the SAVE system is $\{F_1, F_2, \ldots, F_I\}$.
9. Failures in versions from different stages are independent.
10. A version that elects a faulty input fails, even if it performs the correct computation. That is, we assume that a version cannot recover a correct output from a faulty input. Note that a version may elect a faulty input because it received a majority input that was faulty.
11. Similarly we assume that if a faulty message is inserted into a channel, the message that reaches the recipient will also be faulty.
12. We assume for convenience that all stages have the same number of versions N.
13. We assume that there is a probability, c of channel failure.
14. Channels become corrupt independently.
15. Though this is implied in the original model, we make explicit the assumption that the likelihood of existence of a particular fault, F_i, in a version is independent of the likelihood of the system receiving a demand in that fault's failure region.
16. We assume fault containment; that one module failing cannot cause another module in the same stage to fail. This implies, in particular, that one module can't cause another module in its stage to become malicious.
17. We assume that module failure is independent of channel failure.

Before continuing our calculations, we briefly indicate justifications for the major assumptions added to the original model.

Version failure at different stages independent.
Since the versions at each stage implement a different function from the version at any other stage, and the versions are implemented by randomly chosen developers, we feel that it is reasonable to assume that the versions from different stages would fail independently.

A Version cannot generate a correct output in a round where it elects a bad input.
It is very unlikely that a version can generate a correct output from an incorrect input when the output is a direct function of the input. It is therefore reasonable to assume that a version that elects an incorrect input produces an incorrect output.

Correct input to stage 1.
For simplicity, we also assume that the versions at the first stage all receive the (same) correct input from outside the system. This assumption is not critical, but it reduces the complexity of our resulting expressions. For the SAVE system, the input at the first stage would be the information that the voter provides to the system through the User Interface and the information provided by the authorization token. The user enters information at a single UI and there is a single authorization token. Also, the stage 1 Modules are Listeners which read the user information directly as it is entered and extract the information directly from the authorization token. Thus it is reasonable to assume that the Listeners all receive the same information, and that that it is the information directly entered by the user or contained in the authorization token. The information entered by the user and read from the authorization token is taken by the system to as "correct" by definition.

For brevity, several symbols are used in our calculations. An attempt was made to keep them consistent with the symbols used in [51]. The symbols used are outlined in Table 4.1 below.

PFD	"Probability of failure per demand"
Δ	PFD of a generic SAVE system, seen as a random variable
θ_m	PFD of stage m of a generic SAVE system
N	Number of software versions at a stage.
M	Number of stages
I	Number of potential faults (and failure regions) in a software version.
$p_{i,m}$	Probability of the i-th potential fault being present in a randomly chosen version
q_i	Probability of a demand which is part of the failure region corresponding to the i-th potential fault being presented to the system during its operation. (PFD associated with the i-th potential fault (and failure region).
c	Probability that a channel fails.

Table 4.1: Mathematical symbols and abbreviations

4.2.3 Model of SAVE System PFDs

As in [51], the PFD of a version or system here is the sum of the contributions of the individual faults. Each fault contributes q_i with probability p_i and 0 with probability $(1-p_i)$ to a single version. We extend the model presented in [51] by incorporating the effect of the possibility of channel failure after each stage. As we will show, the possibility of channel failure increases the PFDs of each stage and of the system. As we also show, the amount of the PFD increase is relatively larger for smaller N.

PFD estimate

Stage 1 Probabilities.

$E[\theta_1|$correct input$]$: Mean value of the PFD of stage 1 of a generic SAVE System

Fault Fi contributes to stage failure if the demand received by the system is a failure point for $n \geq \lceil N/2 \rceil$ versions in stage 1, i.e. if some $n \geq \lceil N/2 \rceil$ versions in stage 1 fail given correct inputs. Since the fault p_i is assumed to occur independently among the modules, we can use the binomial distribution.

$$E[\theta_1|\text{correct input}] = \sum_{i=1}^{I} q_i \sum_{n=\lceil N/2 \rceil}^{N} \binom{N}{n} (p_{i,1})^n (1 - p_{i,1})^{N-n}$$

We assume that the initial input to the system is correct.

$\therefore$

$$E[\theta_1] = \sum_{i=1}^{I} q_i \sum_{n=\lceil N/2 \rceil}^{N} \binom{N}{n} (p_{i,1})^n (1 - p_{i,1})^{N-n}$$

Stage m Probabilities.

$E[\theta_m|$correct input$]$: Mean value of the PFD of stage m of a generic SAVE System, given majority of its inputs to each module correct.

Fault Fi contributes to stage failure if the demand received by the system is a failure point for $n \geq \lceil N/2 \rceil$ versions in stage m, i.e. if some $n \geq \lceil N/2 \rceil$ versions in stage m fail given correct inputs. Since the fault p_i is assumed to occur independently among the modules, we can use the binomial distribution.

$$E[\theta_m|\text{correct input}] = \sum_{i=1}^{I} q_i \sum_{n=\lceil N2 \rceil}^{N} \binom{N}{n} (p_{i,m})^n (1 - p_{i,m})^{N-n}$$

$E[\theta_m]$: Mean value of the PFD of stage m of a generic SAVE System

Pr(stage m fails) =
Pr (stage m-1 fails) +
(1 – P(stage m-1 fails)) ×
[P(enough correct stage m modules have enough good input corrupted by bad channels to invalidate majority | stage m-1 correct) +
(1 - P(enough correct stage m modules have enough good input corrupted by bad channels to invalidate majority | stage m-1 correct)) ×
P(stage m fails | majority of inputs to each of its modules correct)]

Let R be the number of correct modules in stage m-1 | stage m-1 is correct.

P(a channel is corrupt ⋂ its sender is a correct module)
= P(a channel is corrupt | its sender is a correct module) ×
P(its sender is a correct module)
= P(a channel is corrupt) × P(its sender is a correct module)
= c × R/N = (cR)/N

Let V be P(a stage m module has enough good input corrupted from bad channels that it cannot recover | stage m-1 correct).

V = P(≥ R – ⌊N/2⌋ channels corrupt good input | stage m-1 correct)

Since we assume that channel failures are independent, we can use the binomial distribution.

$$V = \sum_{R=\lceil \lfloor N/2 \rfloor + 1 \rceil}^{N} \left(\sum_{n=R-\lfloor N/2 \rfloor}^{R} \binom{N}{n} (cR/N)^{n} \left(1 - cR/N\right)^{R-n} \right) * P(R \mid \text{stage m-1 correct})$$

Let S be P(enough correct stage m modules receive input from corrupted channels to invalidate majority | stage m-1 correct).

We assume that channel failure is independent, and therefore the modules with input corrupted by bad channels are also independent. Therefore we can use the binomial distribution.

$$S = \sum_{n=\lceil N/2 \rceil}^{N} \binom{N}{n} V^{n} \left(1 - V\right)^{N-n}$$

$\therefore$

$$E[\theta_m] = E[\theta_{m-1}] + (1 - E[\theta_{m-1}]) \times \{S + (1 - S) \times \sum_{i=1}^{I} q_i \sum_{n=\lceil N/2 \rceil}^{N} \binom{N}{n} (p_{i,m})^{n} \left(1 - p_{i,m}\right)^{N-n} \}$$

System Probabilities.

E[Δ]: Mean value of the PFD of a generic SAVE System

$E[\Delta] = E[\theta_M]$

4.3 Application of the Model to the SAVE System

The parameter values are unknown and are difficult to measurable in practice. However estimates of these parameters are still practically useful for making loose predictions

about the reliability of the system and special cases can still provide valuable insight. Let us consider some simple parameter values so that we can use the model to develop intuition about the system, even though we do not yet have actual data.

1. Consider $p_{i,m}$ the same for all i,m. Let this value be p.
2. Consider q_i the same for all i. Let this value be q.
3. From 1 and 2, we have the assumption that "θ_m | all previous stages correct" is the same for all m.

We assume that failure regions of different faults do not overlap, therefore each demand hits at most one fault. Let us assume, for simplicity that each fault is equally likely to be "hit" – that q = h/I, where h is some probability that a demand hits some fault. Let us assume that h=.8.

Let us also assume, for simplicity, that P(R=r | stage m-1 correct) is uniformly distributed, with probability $1/(\lfloor N/2 \rfloor + 1)$ for $\lceil N/2 \rceil \leq r \leq N$. We assume a basic distribution so that we can use the model to develop intuition about the system, even without actual values. As work on SAVE progresses, it is expected that we will get increasingly better estimates for these parameters and distributions.

Even with these simplifications, an N-version system is expected to perform better than its corresponding single version system. For example, for N=10, and c=.01 and p=.2, our model indicates that the 10-version system is expected to perform 5 times better than its single-version counterpart (N=1). A 0.2 probability that a fault is included in a version is conservative; according to [66] results from N-version experiments over the years have shown that the chance of a completely fault-free version can range from 60% to 90%. Additionally, they expect that good quality software development – in the traditional sense – can deliver software with fault densities between 1 and 0.1 faults/KLOC [66]. We believe that for our system an N of 10 is also reasonable. Since our modules are relatively simple, professional programmers should be able to produce 10 diverse versions of each module quickly. Further, from a performance perspective, our testing of multiple instances of our

versions running together, lead us to believe that our voting system could still allow a voter to cast a ballot in less than 2 minutes. We intend to keep working on efficiency in our communication. As expected, 1-version system reliability is reduced more by unreliable channels than the N-version system reliability. For example, for N=10, and c=.1 and p=.2, our model indicates that the 10-version system is expected to perform 7 times better than its corresponding single-version.

These estimates are conservative. For example, if our distribution of p_i was more realistic, the likelihood of having high-quality modules would be apparent, as would the skew of the module quality distribution towards the higher-quality end. Having multiple versions, each with different sets of faults, is less risky than having a single-version. If the version in the single-version system happened to be one of low or moderate quality, it is expected that the N-version system would show even greater gains in reliability than our model estimates with our simplifying assumptions.

We should also realize that the N-version ability to withstand channel failure directly results in a more secure system than its corresponding single version system. If the output channel of a single-version system is corrupted or disconnected, the entire system fails. However an N-version system will be able to continue, even with some channel failures. The extent of the N-version system's ability to resist channel failure, at any given module input point, depends on the number of correct modules at the previous stage. A similar argument establishes that an N-version system is able to withstand denial-of-service attacks directed at a minority of modules, whereas its corresponding single-version system is not. A voting system that can help defend against denial-of-service would be very useful as this attack is one of the most feared for distributed systems communicating over a network. Because of these security features, our system can more reliably be used with components at separate computers, communicating over the internet.

Chapter 5

Implementation

For the current prototype, we had 4 students, Shawn Sullivan, Arturo Hinojosa, David Chau and Kevin Emery, working as independent developers. These students created software modules for the SAVE system. Shawn worked on a User interface Module and on ballot design, whereas the others worked on internal modules. I performed the duties of the specification team, coordinating team and independent testing team. I created the specifications for the modules and took steps to maximize and maintain software diversity among the modules and software quality.

We did not use a formal specification language, because we wanted to have a workable prototype in a short time – we wanted to begin work immediately and felt that the unfamiliarity with a formal specification language would cost more time and effort to understand than a plain English specification. The specification included some communication details such as xml dtd sections, providing the formats of various messages to and from the modules. However, module functionality was generally separated from the authenticated SSL communication protocol implementation. The functional specifications are included as Appendix A.

Development began as early as possible, but was limited by the relatively late date that I joined this project and by the limited number of hours that the students had to contribute to this project. However, several modules were completed. I added modules to the system as they were completed and tested them both individually and back-to-back. Testing focused on correctness and performance testing. Tiger-team testing and rigorous security testing is left as future work since we believe that such testing would be more useful after an optimal number of versions – in terms of system performance and reliability – are created. However, we established some confidence in the security of our prototype by testing module and communication protocol correctness. For example, we tested to verify that authentication is implemented correctly in each module, as authentication is crucial to security. Though we had fewer versions than we plan to have

eventually, we were able to do some performance testing for larger values of N using copies of the versions we have. The official prototype, and correctness testing on this prototype, would not contain copies of versions, as this would obviously limit diversity.

5.1 User Interface Implementation

SAVE is committed to being accessible and intuitive, and also to preventing avoidable voter errors. The implemented SAVE UI allows a voter to correct mistakes by unselecting candidates and selecting new ones, as many times as desired before the vote is cast. The SAVE UI also makes it impossible to make an obscure indication, between two candidates, for example, as might occur with paper ballots or misaligned punch cards. Many ballots are traditionally discarded and not counted because of such stray or obscure marks [3]. We list here some of the major features of our current SAVE UI implementation.

The implemented SAVE UI also intentionally makes it easy for a voter to navigate the ballot, by using color and texture to serve as cues to events that should be noted. Though the color is not reproduced here, the cues are still distinctive in greyscale, and by the textural changes. Figure 5.1 shows a ballot, displayed as expressed in the current SAVE UI display specification. The voter is at the Supreme Court Justice race, as indicated by the lighter colored tab. Further, this race allows a maximum of 4 candidates to be selected, as indicated by the 4 sections on that tab.

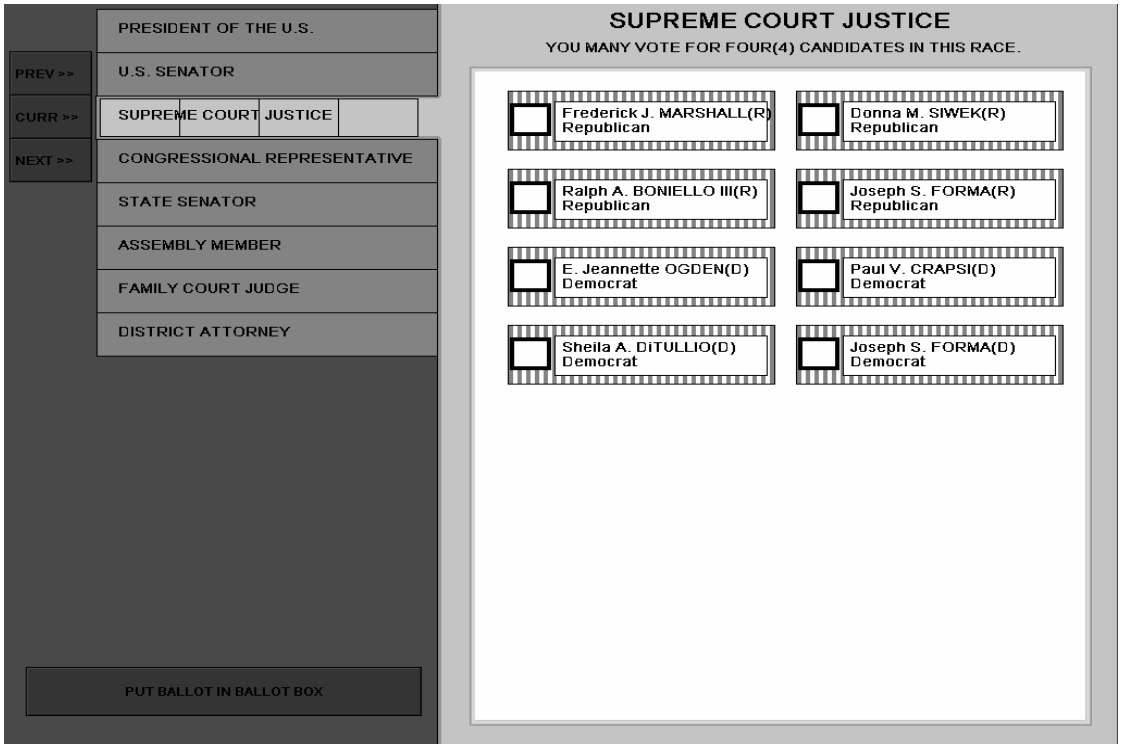

Figure 5.1: SAVE ballot

If a candidate is selected, the box containing that candidate's name changes color and texture to indicate the selection. The texture change is depicted in Figure 5.2; the lines at the candidate's button changes from vertical to horizontal, and additionally a check mark is placed near the selected candidate's name. Also, as shown in Figure 5.2, one of the sections on the Supreme Court Justice race's tab changes color to indicate that 1 candidate has been selected, and up to 3 more may be selected.

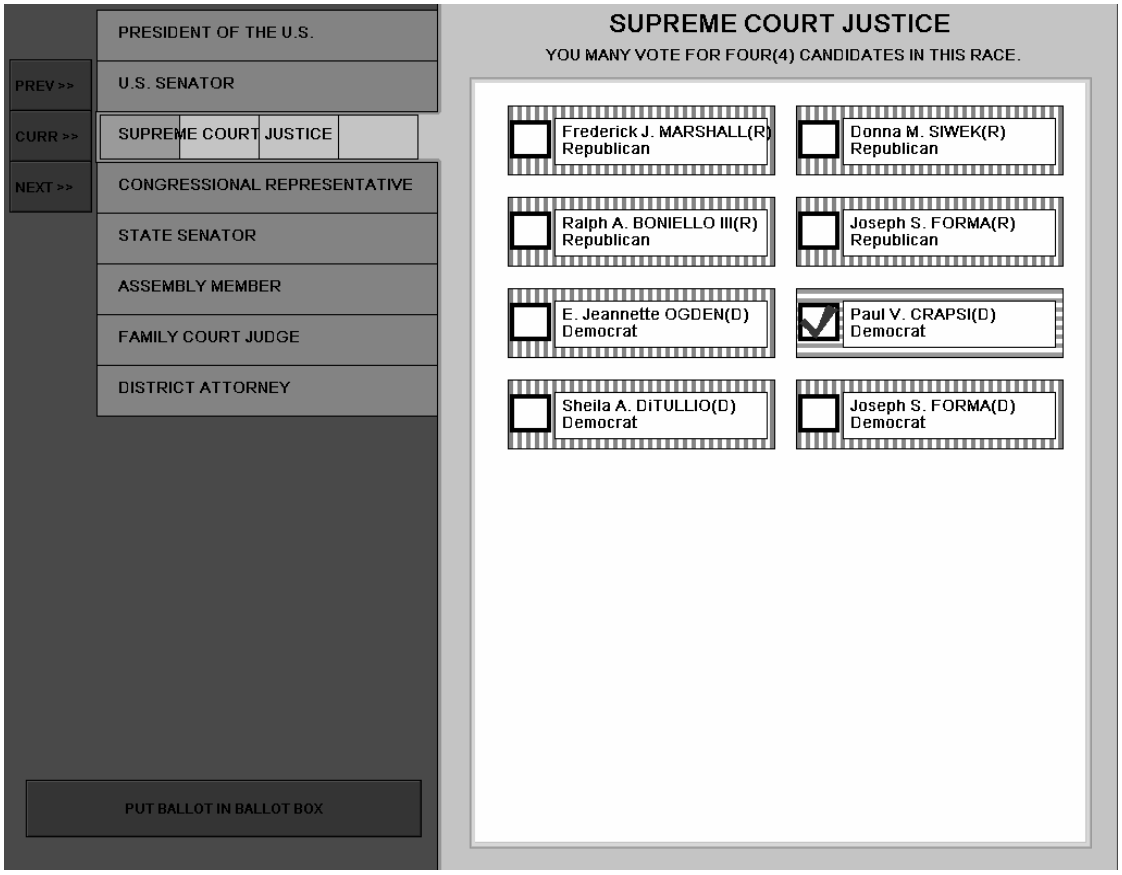

Figure 5.2: Candidate selection

If the maximum allowed number of candidates for a race has been selected, the surrounding canvas similarly changes color and texture to mark the completion. The completed race's tab also changes color and texture, so that it clearly stands out among the other, uncompleted, races on the ballot. Additionally, as depicted in Figure 5.3, the SAVE UI prevents a voter from overvoting. If the voter has already selected the maximum amount of candidates allowed for a race, and tries to select an additional candidate, the UI prevents this and informs the voter that he must unselect one of his chosen candidates before he can select the new candidate.

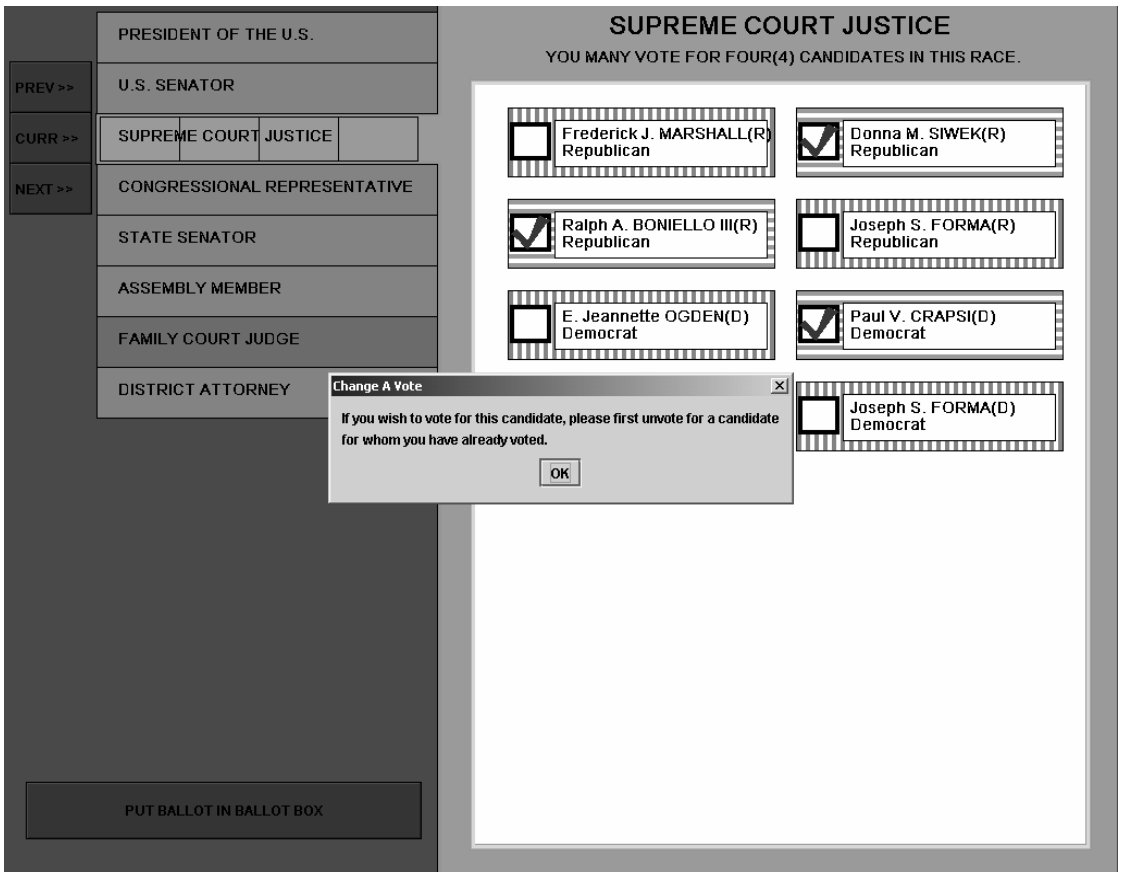

Figure 5.3: Overvoting prevented

The title of the race is clearly indicated on the race's tab, and it is easy to skip to any race on the ballot by clicking the tab; this makes it easy for the voter to find, and go to, the races he is most interested in, in the order that he would like to. The tab color and texture changes when races have been completed are very useful cues, which help the voter keep track of his progress. For example, in Figure 5.4, the color changes are represented as 3 shades of grey at the tabs. The darkest shade, as shown on the Supreme Court Justice tab and the Family Court Judge tab, represents races where selections have already been made. The middle shade, as shown on the District Attorney tab, represents races where no selections have yet been made. The lightest shade, shown on the State Senator tab, represents the race that the voter is currently viewing.

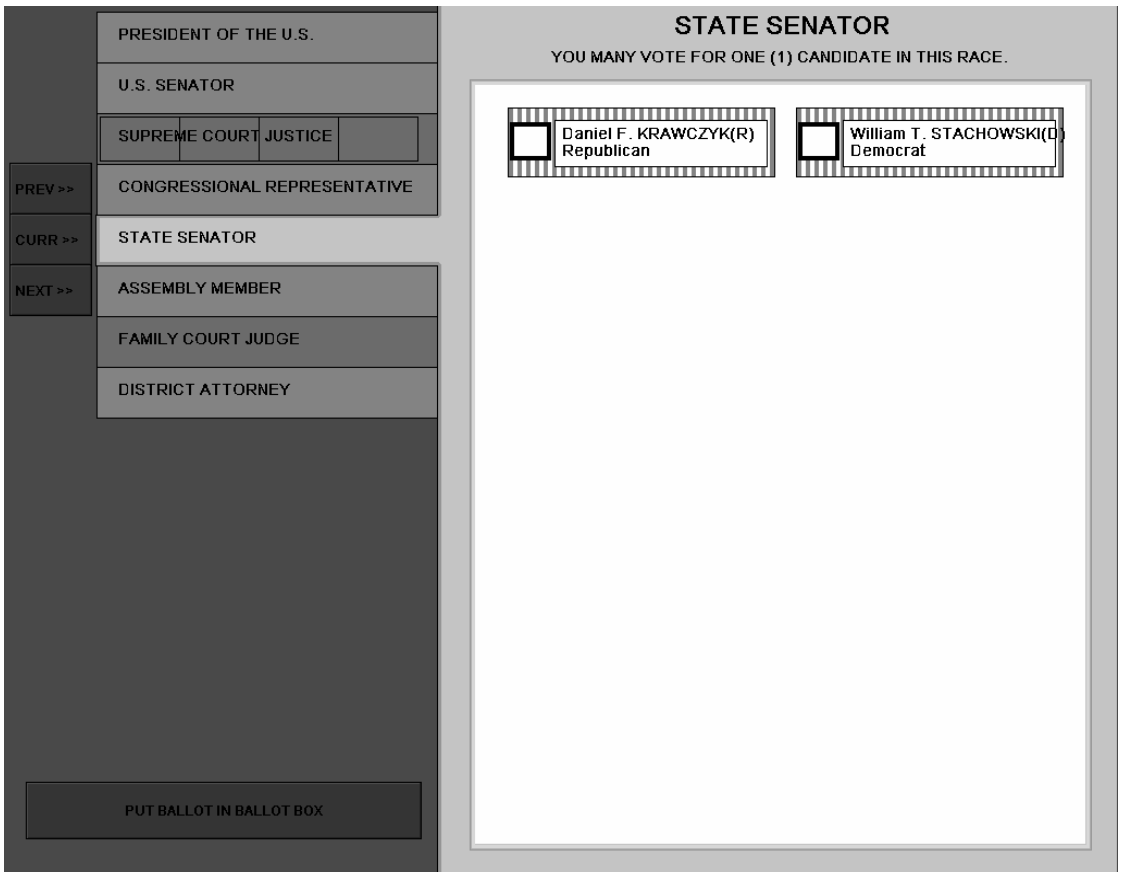

Figure 5.4: Tab color cues

Significantly, our UI provides an alert for verification before the ballot is submitted, containing the races that the voter has voted in and clearly indicating races where the voter undervoted. If, upon receiving this feedback, the voter decides he wants to make changes, the user is allowed to return to the ballot and edit it. The voter does not have to completely start over, with a fresh ballot, as would generally be the case with paper; all of the voter's previous selections are still present on the ballot, and may be edited individually. Figure 5.5 shows such a summary of the voter's selections.

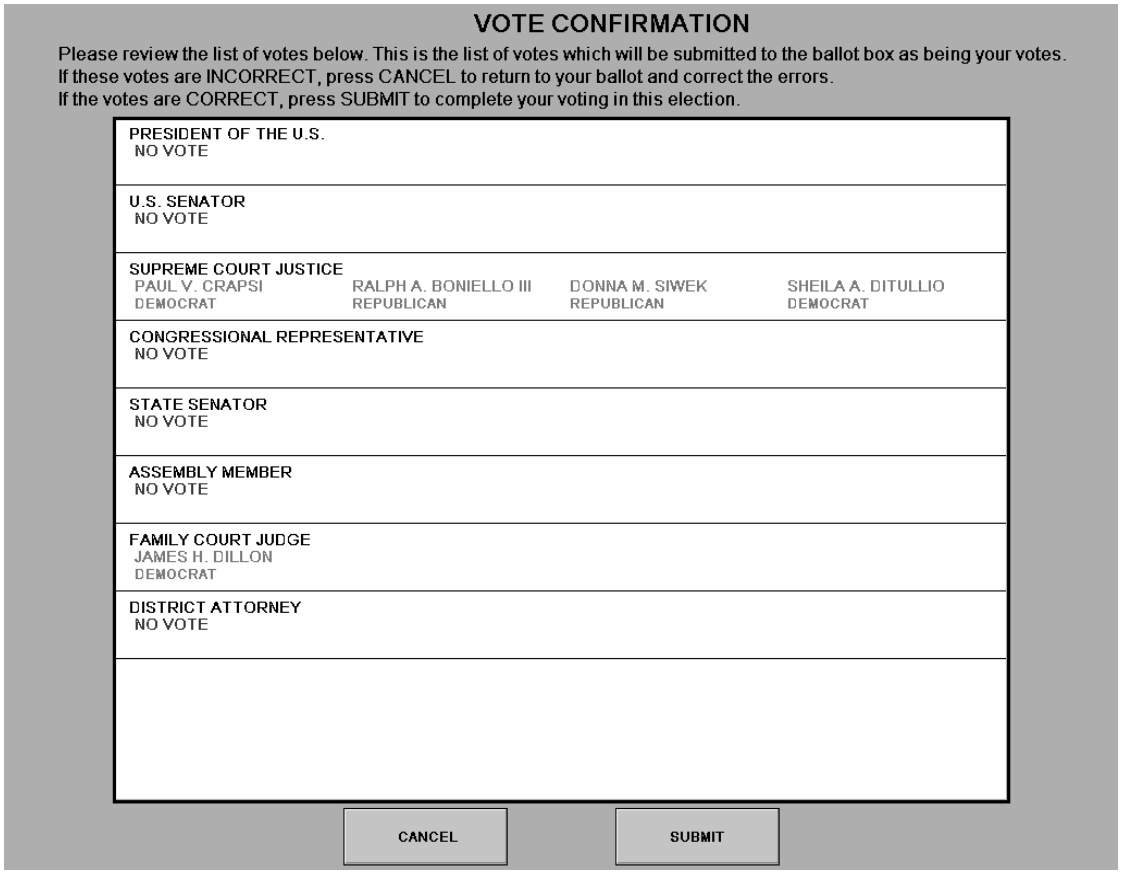

Figure 5.5: Summary of voter's selections – displayed before vote is cast

In addition to color and texture changes, we have added other sensory cues. Audio is available, and headphones can be provided to the voter so that this capability can be used in private. Audio is used at all the places where alerts or color/texture changes where mentioned above, to provide cues to voters with vision impairments, or other disabilities. When the voter selects a candidate for example, the UI reports that "Candidate __ has been selected for race __." Additionally, the ballot presented for verification is read off by the UI so that the voter can determine whether or not the ballot is satisfactory, even if he is unable to read the record. These are first steps towards making the SAVE system accessible to all voters. We intend to continue work in this area.

Chapter 6

Issues, Limitations, Considerations, Lessons and Future Work

The SAVE system uses N-version programming to provide a voting system with increased reliability and security. This thesis describes the first formal end-to-end design of the SAVE system. Though a rough proof of concept has been developed, it is our aim to develop an improved system, which takes advantage of the lessons we have learnt in the course of this current work. We discuss here, some of the major issues, considerations and limitations concerning the current SAVE system, as well as our plans for future work.

6.1 Issues and Limitations

With any system, the designer will encounter several tradeoffs, and possible advantages must be weighed against associated limitations to create the best possible system overall. We have achieved a robust, reliable and trustworthy system. However, there are several limitations to the current SAVE system. We expect that many of these limitations will be removed or alleviated in the future, as progress continues to be made towards an optimal n-version voting system.

Security

There is currently a lot of redundancy and repeated patterns in the SAVE system messages due to the XML based message format. We need to investigate how vulnerable this makes the system to ciphertext-only attacks, known-plaintext attacks, and etcetera. Then we can decide, for example, whether the benefits of XML are worth this risk and whether one-time pads should be used to mitigate the risk.

Duration of secrets

The Listener module has access to the voter's completed clear-text ballot. Thus it is important that the Listener not also have access to the voter's identifying information so that malicious Listener modules are unable to create a receipt that may be stored and later used as proof of vote. The voter's identifying information must be available to

Registration modules without being available to Listener modules, or any other modules. To achieve this, we encrypt copies of the voter's identifying information with the encryption public keys of the Registration modules. However, this means that the Registration encryption key pair must be generated sufficiently in advance for each voter to be able to present his encrypted information to the voting machine. It is highly unlikely that the Registration modules' private keys can be extracted from their trusted computing bases or determined from their public keys or signatures, in the weeks or months that it takes to distribute identification tokens. However, the longer a key-pair is in existence, the more likely it becomes, that the secret can be discovered, by some insidious insider for example.

Vertical Collusion

The n-version system is designed to resist collusion among modules in the same stage; a majority of modules must collude to cause incorrect data to be determined correct. However the n-version system is still susceptible to collusion from modules in neighboring stages. For example, a Listener and a Registration module can collude to produce a receipt that is correct with some probability. Encrypting the voter information with the Registration modules' public keys is sufficient only when the Registration modules are prevented from revealing the secret voter id information to the Listener modules and the Listener modules are likewise prevented from revealing the plaintext ballot to the Registration modules. A Registration Module and Aggregator Module could similarly collude. Note that since existing data errors are likely to be detected and corrected at every stage in the SAVE system, any receipt that is generated is not guaranteed to contain the vote that the voter eventually casts. However such a receipt would be correct with some positive probability and that probability might be enough to make it economically viable to some.

One of our design assumptions is that our testing is sufficient to detect whether a module can reveal such secrets to another module with XML messages. The content that would need to be sent is clear in both cases, and therefore such code should be easy to trace. Therefore, we believe that this is a reasonable assumption. Despite this design

assumption, we currently have policies in place to reduce the likelihood of such collusion. Each module is checked for evidence of such collusion, in addition to any other maliciousness or errors, before the module can be certified. Also, developers of modules that may particularly compromise the system by collusion are strictly isolated and particularly monitored. Further, each message sent in the system is logged, along with both its sender and its recipient, so that any Listener module that sends a message exposing secret information will be detected and investigated. Note that the messages stored in the logs are encrypted with the public keys of the modules intended to finally read them and strict separation of knowledge is enforced in the monitoring of these logs. Each employee is allowed to monitor at most a minority of modules' logs from a single stage, and this happens only after the election has ended and the keys are discarded. Therefore each employee is restricted to as much knowledge as the stage being monitored, and that is designed to be minimal. No employee is able to generate a receipt from the logs without several other employees being aware of the problem because collusion of modules from different stages is required for receipt generation. Because of the logs, the threat of discovery is significant and is a substantial deterrent, but we intend to further reduce the effect of possible module collusion in the future.

Historically, we have designed the SAVE system so that voter check-in and casting occurred together, at the Registration Module [45]. This was done to facilitate absentee ballots. However, as indicated above, this presents difficulties given that our current trust model allows for malicious modules. We can instead implement another version of the SAVE system which separate the two. Such an alternative has already been designed. Though it would involve some significant restructuring from our original design [45], we can extend the alternative design to make sure that neighboring modules do not together possess illicit information.

Session Establishment

A critical part of system initialization is establishing, authenticating and securing the directory containing the addresses and functions of each module. We do not trust just any trusted computing base's claim that it contains an official voting module as doing so

would risk attesting fraudulent modules. Instead, we physically authenticate each official voting module's function and address and compile the result into a directory. We do this because software cannot, by itself, verify the claims of remote software, without some pre-established knowledge about that remote software. This has long been the case, and generally trusted humans and companies such as Verisign are relied on to certify the connection between a physically authenticated identity and a public key, and pass on knowledge of that connection to other parties wishing to communicate with the identity. To avoid placing our trust in few external roots of trust which we do not know very well, we rely on a team of carefully selected humans to validate and certify each {module, public-key$_m$} pairing and each {DirG, public-key$_{DirG}$} pairing.

The collection of individuals on the team are chosen so as to minimize the chance of collusion and all the individuals we place any trust in are known and can be held accountable. However this process is somewhat inconvenient. Also, while it might be possible to predict the trustworthiness of persistent software, it is very much more difficult to predict the trustworthiness of human beings, whose predilections might change at any time.

Treatment of Faulty modules

The SAVE system would be more efficient if modules could ignore faulty modules in the previous stage. However, the current SAVE remote communication protocol, geared around the exchange of non-executable messages over unreliable channels, makes it is difficult to detect faulty modules with certainty; a module could be faulty, or some of the channels from that module could be suffering from transient or permanent failures. This is an application of the Byzantine problem, and as such, requests for retransmits would still not offer certainty, and would reduce the efficiency of the system – both in terms of messages and time.

The faulty behaviors of modules are currently noted and the outputs of modules that repeatedly act faulty are given less consideration at comparison points. However, we will continue to investigate improvements over the current system.

Limits to Redundancy

The SAVE system presents only a single display to the voter. This is done to make the system simple to interact with and easy to understand. We believe that it is unreasonable to expect a human voter to be able or willing to make decisions based on multiple screens or screen segments. We are able to detect errors with the single I/O devices and can potentially replace them, but this is inconvenient and harms voter trust in the system. Further, the voting machine's operational software is not redundant and so many software errors here would not be detectable under the current conditions. Future work will involve creating a redundant trusted computing base that could eliminate some of these points of failure. We believe that our system is still overall an improvement over systems with no error masking at all.

Diversity

The SAVE system is highly dependent on module diversity for error detection and error masking. Our work thus far has indicated that it is likely that we can achieve such diversity. However, because of the large size of our possible input space, we cannot be absolutely certain that we have achieved, or can achieve, the diversity necessary to distinguish and correct all possible errors. In short, we have reduced, but not removed, the effects of the theoretically established fallibility of complex software [37]. Our certainty of the diversity we have achieved is also limited by the fact that little progress has been achieved by the research community in measuring software diversity or modeling the effects of individual diversity introducing factors on software failure [38, 40-41, 51, 62].

Complete Elections Protocol

The voting machine and its operations form only part of the entire election process. We assume, for our purposes, that the rest of the election protocol is correct so that we can focus on our contribution. However we are well aware of problems with voter registration and authentication and poll worker adherence to policies [3]. These problems are currently beyond our scope, but others in our research team are addressing these

issues. Additionally, we intend to make contributions to improving registration in the future.

Number of messages

An n-version system generally generates more messages than its single version counterpart. The SAVE system generates $(M-1)*N^2$ messages, where M is the number of stages and N is the number of messages per stage, because the election of inputs for each stage is itself distributed. This distribution is very important for reducing single points-of-failure and the number of comparison points – one for each stage - was chosen to provide suitable error masking. The number of messages appear somewhat large, but M is typically ≤ 6 and N is typically 5, so that $(M-1)*N^2$ is itself manageable, particularly where the modules are distributed across several processors. Each module need only process $\leq N$ messages for each task, and these messages are rather small documents of XML instructions and data. Our prototype manages this number of messages well and we believe that a SAVE system will be able to facilitate voting in reasonable time.

6.2 Considerations and Lessons

It is important that the human component of a secure software system is not ignored during the system's design. We note several particularly helpful and significant managerial measures that can improve the quality and trustworthiness of the software produced.

Relationship Protocols.

Our n-version programming protocol requires humans to implement strong, secure and distrustful protocols that literally constrain their relationship with their coworkers. We have to carefully consider how to make sure that all workers are motivated to do this and trained on how to detect possible actions and resist possible lurings from corrupt coworkers. Humans naturally tend to relax their guard over time and this must also be guarded against; thus the coordination team must continuously monitor the implementation of relationship protocols amongst workers. These issues may arise even within the same development team and should not be neglected or ignored, even if they

are not expected to occur. We found that periodical requests for status reports in this area would probably help keep individuals alert.

Identify Experts and Expert Tasks

As noted by Knight, Leveson and others, some aspects of each function may be harder than others [53]. Where possible designers should attempt to extract and separate the harder aspects or the specialized aspects of the systems functionality so that they can be implemented by expert teams. This reduces the cost of software development because developers are assigned the tasks they are most likely to succeed at. Experts are not wasted on mundane tasks and, since developers need not be skilled at all the technologies and solutions required by the system as a whole, developers will be easier to find at more reasonable costs. This separation also reduces the correlation of software failures with task difficulty [53] since the implementers of each task are those that find that task minimally difficult.

Developer Screening and Training

It may also be possible to "create" experts. Application of forced diversity, for example, is particularly suitable to forms of developer training that do not overly limit diversity. In particular, because sufficiently diverse methods have already been prescribed under the forced diversity paradigm, it becomes possible to train developers in their separate methodologies without affecting the diversity of the system as a whole. The possibility of training is significant as it can improve the quality of the software and the speed with which the software can be developed.

6.3 Future Work

We have several plans for the evaluation, expansion and improvement of the current SAVE system. The work on SAVE is expected to continue in the foreseeable future and will focus initially on validating and verifying the current prototype and design; extending the prototype to fully implement the current design with more versions of each module; developing and using schemes for measuring software diversity; implementing or reviewing the source code of several trusted computing base systems so that we can

certify the reliability and trustworthiness of the trusted computing base that we use; improving the current registration system; and improving election protocols surrounding voter use of the voting machines.

Completion, Testing and Expansion

We plan to implement our own trusted computing base or review the source code of externally implemented systems so that we can certify our expectations of correctness and trustworthiness of this system. The trusted computing base theoretically ensures that the system's I/O is also correct and trustworthy. However, we plan to complete our display monitoring modules so that we can consistently use our system premise of n-version programming and apply the added reliability and other benefits of n-version programming to detecting errors in the system's output. For this reason as well, we also plan to investigate the possibility of a redundant trusted computing base and the practicality of including it in our system.

Already, the ballot display style that we have designed is much easier to read and navigate, gives much clearer feedback about the voter input it receives, and prevents much more errors, including over-voting, than current paper ballot systems [3]. However, we plan to continue our ballot display design work so that we can optimally convey the contents of a ballot to any franchised person, including those persons with special needs [5].

Further, we intend to extend the number of versions of each module to at least 5 so that we can more rigorously test the performance and reliability of the SAVE system. Further testing is necessary and a formal *test coordination team* must be established to examine the reliability and diversity of our system and the nature of correlation of the failures discovered. The *test coordination team* will also be responsible for editing and enforcing the test specifications, especially for back-to-back testing where there is a threat to diversity. We intend to vary the size of N and test recursively until we achieve optimal performance and reliability for the system. Also, the SAVE protocols are subject to

change if modifications to them show better testing results. In fact, it is expected that the protocols will be improved on as a more advanced and sophisticated prototype evolves.

We also intend to investigate the many avenues for forced diversity in the SAVE system and incorporate the most beneficial of these. We expect that forced diversity will improve the reliability and diversity of our system, in accordance with n-version research predictions [38, 58].

Measuring and Modeling Software Diversity and its Effects

Because of the importance of diversity in an n-version system, it is crucial that we find ways of measuring diversity and also ways of measuring and modeling the effect of various types of diversity on system failures. We have already begun plans for statically and dynamically measuring software diversity from the source code and its executables as well as from the execution of the software. We hope to use resulting diversity rankings of sets of modules to help elect the subset of modules that would provide the best combination of diversity and individual correctness. This extra knowledge will improve the overall SAVE system created. Additionally, we realize that the effects of the diversity present will produce a much better prediction of the quality of the generated SAVE system than the magnitude of the diversity. We thus expect to also begin work in the future on modeling and measuring the effects of diversity on the system.

As an added pursuit, we hope to determine whether software modules can be made sufficiently diverse that very different techniques would be needed to hack them. If the module vulnerabilities can be made diverse enough, we postulate that it would be difficult for an adversary to gain control of a majority of the modules at any stage. This has very obvious implications for the contribution of diversity to security.

Voter Registration and Election Protocols

Voter registration problems may have caused up to 3 million votes to be lost in the 2000 Presidential election [3]. It is clear therefore that problems with voter registration technologies and protocols need to be addressed. We intend to work on the voter

registration database management systems. Our work will include designing protocols to make it easier for voters to enter and update their registration information and for polling stations to be aware of these changes in a timely fashion; designing input schemas to facilitate the presence of correct and timely voter information in the registration databases; and designing and implementing constraint-checkers to ensure that the voter information in the registration databases are as consistent and correct as possible.

Additionally, we intend to work on improving the protocols surrounding the actual use of the voting machines, so that we can improve the reliability of voting overall. The relationship between the poll workers and the voting machines needs to be as simple and efficient as possible and needs to be clearly established and expressed. Further, poll worker training and adherence to protocol is important and thus we need to find ways of achieving improvement in these areas.

Chapter 7

Conclusion

Many persons are unwilling or unable to vote because current systems are not sufficiently accessible or secure [5, 84]. Others who make the attempt often find that they are unable to vote, or their votes are not counted, due to flaws in the voting system [3]. SAVE is a viable approach to improved accessibility, security and correctness for voting and, as such, SAVE can address these problems.

SAVE principally reduces single points of failure and trust. This is a significant improvement over single-version voting systems, where a few corrupt programmers or faulty sections of software could corrupt the entire system and spoil cast votes. The distributed and redundant nature of the SAVE architecture allows SAVE to detect and correct most non-pervasive errors, including channel failures, and recover from attacks that may cause such errors. SAVE can therefore function well, even over unreliable networks or in the face of certain attacks, thus offering much needed reliability and robustness to the field of voting systems. Further, SAVE is modular, and each of these modules is relatively simple. This makes a SAVE system relatively easier to implement and certify than many current voting systems. The SAVE modularity also makes it easy to remove faulty modules if any are discovered, or incorporate new improved modules or new user interface display styles for the benefit of the public; there need be no legacy code.

The SAVE system is practical and can be used in many voting environments. Though we are not yet ready for internet voting, it has already been indicated that SAVE, as an N-version system, performs better over unreliable networks than corresponding single version systems. SAVE would thus be more suitable for internet voting than single-version systems when improved authentication, etcetera makes internet voting feasible. The SAVE theory promotes as much diversity as possible, and so a SAVE system would be most secure implemented across diverse operating systems, hardware, etcetera, in a closed network. However SAVE can be implemented on a single computer. Further,

basic operating systems can be used; there is no need to use large, monolithic operating systems. As such, SAVE can conceivably be implemented on a simple game machine. In general, SAVE can be implemented in almost any computing environment.

As indicated in this thesis, N-version programming increases the reliability and security of systems – most of the benefits of SAVE were achieved by applying the concepts of N-version programming and modularity. The awareness that SAVE brings, about the suitability of N-version programming for voting, is a major contribution to the field of voting. We believe that SAVE is sufficient to demonstrate the feasibility and advantage of N-version voting systems over traditional single-version systems. Further, the benefits of SAVE, and its applicability to many voting environments, assure us that SAVE is a useful voting system in practice. SAVE is poised to meet the urgent need of the voting public for a voting system that is accessible and that counts their votes reliably, correctly and securely.

References

[1] U.S. Department of State International Information Programs Publication. "Rights of the People: Individual Freedom and the Bill of Rights," [online document], (October 2003) [cited January 18 2004], Available at: http://usinfo.state.gov/products/pubs/rightsof/

[2] M. Foreman, D. Reid, P. Reilly and A. Wiggins "Voting Irregularities in Florida During the 2000 Presidential Election" [online article], (June 2001), [cited January 18 2004], Available at: http://www.usccr.gov/pubs/vote2000/report/main.htm

[3] California Institute of Technology and Massachusetts Institute of Technology. "Voting, What Is, What Could Be." July 2001.

[4] R. Smith, "Electronic Voting: Benefits and Risks," Trends & Issues in Crime and Criminal Justice, Australian Institute of Criminology, No. 224, May 2002.

[5] J. Dickson, "Voter Verified Paper Ballot: De facto Discrimination Against Americans with Disabilities," in First Symposium on Building Trust and Confidence in Voting Systems, December, 2003.

[6] M. Bellis, "The History of Voting Machines," [online article], (November 1998) [cited January 29 2004], Available at: http://inventors.about.com/library/weekly/aa111300b.htm

[7] S. Knack and M. Kropf. "Invalidated Ballots in the 1996 Presidential Election: A County- Level Analysis." in Journal of Politics, 65:881-897, 2003.

[8] L. Landes "Voting machine fiasco: SAIC, VoteHere and Diebold," [online article], (August 2003), [cited January 18 2004], Available at: http://www.onlinejournal.com/Special_Reports/082003Landes/082003landes.html

[9] T. Kohno, A. Stubblefield and A. D. Rubin, "Analysis of an Electronic Voting System," [online article], (February 2003), [cited January 18 2004], Available at: http://avirubin.com/vote.pdf

[10] Associated Press. "Site of electronic voting firm hacked," [online article], (December 2003), [cited January 18 2004], Available at: http://www.cnn.com/2003/TECH/biztech/12/29/voting.hack.ap/index.html

[11] L. Landes, "Elections In America - Assume Crooks Are In Control" [online article], (September 2002), [cited January 18 2004], Available at: http://www.ecotalk.org/AmericanElections.htm

[12] “VoteHere Reviewers Archives,” [online article], (January 2004), [cited January 18 2004], Available at: http://www.cs.virginia.edu/pipermail/votehere-reviewers/2004-January/thread.html

[13] J. Schwartz, “Security Poor in Electronic Voting Machines, NEW Study Warns,” New York Times, January 29th 2004.

[14] P. Troy, “The Georgia Debacle - in plain english," [online article], (November 1998) [cited January 29 2004], Available at: http://workersrighttovote.org/summary.htm

[15] D. Perata and R. Johnson, “Don't take chances on electronic voting machines,” [online article], (April 2004), [cited May 11 2004], Available at: http://www.signonsandiego.com/uniontrib/20040423/news_lz1e23peralta.html

[16] M. Hardy, “California nixes e-voting,” [online article], (May 2004), [cited May 11 2004], Available at: http://www.fcw.com/fcw/articles/2004/0503/web-evote-05-03-04.asp

[17] A. Viglucci, “1,700 Dade Voters Mispunched Chads,” [online article], (January 6, 2001), [cited May 11 2004], Available at: http://www.commondreams.org/headlines01/0106-01.htm

[18] P. G. Neumann, “Security Criteria for Electronic Voting,” in 16th National Computer Security Conference, Baltimore, Maryland, September 20-23, 1993.

[19] B. Williams, “Description of a Voting System,” for I.E.E.E. Voting Systems Standards Committee, November, 2001.

[20] HELP AMERICA VOTE ACT OF 2002, [online], (April 2003), [cited January 23 2004], Available at: http://fecweb1.fec.gov/hava/hava.htm

[21] M. Andino, Help America Vote Act Of 2002 - South Carolina State Plan, September, 2003.

[22] D. Jefferson, “Requirements for Electronic and Internet Voting Systems in Public Elections, Compaq Systems Research Center,” [online document], (August 2001), [cited January 18 2004], Available at: www.vote.caltech.edu/wote01/pdfs/djefferson.pdf

[23] “Election Systems and Software,” [online], (January 2004), [cited January 18 2004], Available at: http://www.essvote.com

[24] “Diebold Election Systems,” [online], (January 2004), [cited January 18 2004], Available at: http://www.diebold.com/dieboldes

[25] “Vote Here Inc,” [online], (December 2003), [cited January 18 2004], Available at: http://www.votehere.com

[26] “Diversified Dynamics, Inc,” [online], (January 2004), [cited January 18 2004], Available at: http://www.divdyn.com

[27] D. Malkhi, O. Margo and E. Pavlov, “E-Voting Without `Cryptography,” in Financial Cryptography ‘02, February 2002.

[28] R. Cramer, R. Gennaro and B. Schoenmakers, “A Secure and Optimally Efficient Multi-Authority Election Scheme,” in Advances in Cryptology — EUROCRYPT ’97, in volume 1233 of Lecture Notes in Computer Science, pages 103–118, Konstanz, Germany, 11–15 May 1997. Springer-Verlag.

[29] A. Fujioka, T. Okamoto, and K. Ohta, "*A Practical Secret Voting Scheme for Large Scale Elections*", in Advances in Cryptology -- Auscrypt'92, in LNCS Vol.718, pages 244--260, Springer-Verlag, 1992

[30] M. J. Radwin, “An untraceable, universally verifiable voting scheme,” in Seminar in Cryptology, December 1995.

[31] L. F. Cranor, R. K. Cytron, “Sensus: A Security-Conscious Electronic Polling System for the Internet,” in Proceedings of the Hawaii International Conference on System Sciences ‘97, January 1997.

[32] M. A. Herschberg, “Secure Electronic Voting Over the World Wide Web,” Masters of Engineering Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1997.

[33] D. Clausen, D. Puryear and A. Rodriguez, “Secure Voting Using Disconnected, Distributed Polling Devices,” [online document], (June 2000) [cited January 19 2004], Available at: http://dave.47jane.com/voting/cs444n_voting_report.pdf

[34] M. Burmester and E. Magkos, “Towards Secure And Practical E-Elections In The New Era,” Advances In Information Security - Secure Electronic Voting, Kluwer Academic Publishers Pp. 63-76, 2003.

[35] A. Baraani, J. Pieprzyk and R. Safavi, “A Review Study on Electronic Election,” Department of Computer Science, The University of Wollongong, December 1994.

[36] R. Mecuri, “Florida 2002: Sluggish Systems, Vanishing Votes,” Vol. 45, No. 11 Communications of the ACM, November 2002.

[37] F. P. Brooks Jr, The Mythical Man-Month: Essays on Software Engineering, Addison Wesley Longman, Inc, May 2001.

[38] A.A. Avizienis, “The Methodology of N-version Programming”, Software Fault Tolerance, edited by M. Lyu, John Wiley & Sons, 1995.

[39] M. R. Lyu and A. Avizienis, "Assuring design diversity in N-version software: a design paradigm for N-version programming," in Proc. DCCA '91, pp. 197-218, 1991.

[40] A. Avizienis and L. Chen, "On the implementation of N-version programming for software fault-tolerance during program execution," in Proc. Intl. Computer software and Appl. Conf. '97, pp. 145-155, 1977.

[41] P. Popov, L. Strigini and A. Romanovsky, "Choosing effective methods for design diversity – how to progress from intuition to science," in Proc. SAFECOMP '99, 18th International Conference on Computer 65, Reliability and Security: 272-285, Toulouse, France 1999.

[42] I.V. Kovalev, K.-E. Grosspietsch, "An Approach for the Reliability Optimization of N-version Software under Resource and Cost/Timing Constraints," Institut für Autonome intelligente Systeme, 1999 Publication.

[43] S. Mitra, N. Saxena and E. J. McCluskey, "A design diversity metric and reliability analysis for redundant systems," in Intl. Test Conference '99, pp. XX 1999.

[44] A. Romanovsky, "An exception handling framework for N-version programming in object-oriented systems," in ISORC '00, Object-Oriented Real-Time Distributed Computing, 2000 Proceedings. Third IEEE International Symposium, March 2000 Pages:226 – 233

[45] T. Selker and J. Goler, "SAVE: A Secure Architecture for Voting Electronically," [online document], (November 2003), [cited January 18 2004], Available at: http://www.vote.caltech.edu/Reports/vtp_WP7.pdf

[46] T. Yamamoto, M. Matsushita, T. Kamiya and K. Inoue, "Measuring Similarity of Large Software Systems Based on Source Code Correspondence," in IEEE Transactions On Software Engineering, Vol. XX, No. Y, Month 200X

[47] E. Herbert, "Ballot Design," Advanced Undergraduate Project, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 2002.

[48] Vote America, 48 Electronic Voting, [online article], (January 2004) [cited January 22 2004], Available at: http://www.voteamericavote.com/electronicvoting.html

[49] Votewatch: Your Eye On Elections, [online article], (January 2004) [cited January 22 2004], Available at: http://www.votewatch.us/

[50] Electronic Frontier Foundation, [online article], (January 2004) [cited January 22 2004], Available at: http://www.eff.org/Activism/E-voting/

[51] P. Popov and L. Strigini, “The Reliability of Diverse Systems: a Contribution using Modelling of the Fault Creation Process,” in DSN'01, The International Conference on Dependable Systems and Networks, July 01 - 04, 2001, Goteborg, Sweden, p. 0005

[52] L. Hatton, “N-Version Design Versus One Good Version,” in IEEE Software, Volume 14, Issue 6, November 1997, Pages: 71 – 76.

[53] J. Knight and N. Leveson, “An experimental evaluation of the assumption of independence in multi-version programming,” in IEEE Transactions on Software Engineering, 1986. 12(1): p. 96-109.

[54] S. Brilliant, J. Knight and N. Leveson, “Analysis of faults in an N-version software experiment,” in IEEE Transactions on Software Engineering, 1990. 16(2): p. 238-247.

[55] B. Littlewood and D. Miller “Conceptual Modeling of Coincident Failures in Multiversion Software,” in IEEE Transactions on Software Engineering, Volume 15 , Issue 12, p. 1596-1614, December 1989.

[56] L. Hatton, “Are N average software versions better than 1 good version?” in IEEE Software, (14), p. 71-76, Nov-Dec 1997.

[57] D. Partridge, N. Griffith, D. Tallis and P. Jones, An Experimental Evaluation of Methodological Diversity in Multi-version Software Reliability. Technical Report 358, University of Exeter, 1996.

[58] D. Partridge and W. Krzanowski, *Distinct failure diversity in multiversion software*. Technical report, 348. Department of Computer Science, Exeter University, 1997.

[59] B. Littlewood, P. Popov and L. Strigini, "Modelling software design diversity - a review", in ACM Computing Surveys, Vol. 33, No. 2, June 2001, pp. 177-208.

[60] B. Littlewood, “The Use of Proof in Diversity Arguments,” in IEEE Transactions On Software Engineering, Vol. 26, No. 10, October 2000.

[61] B. Littlewood, P. Popov and L. Strigini, "Design Diversity: an Update from Research on Reliability Modelling", in Proc. 65-Critical Systems Symposium 2001, Bristol, UK, Springer-Verlag.

[62] B. Littlewood and L. Strigini, “A discussion of practices for enhancing diversity in software designs,” DISPO Project Technical Report LS_DI_TR_04, 2000.

[63] P. Popov and L. Strigini, “Estimating Bounds on the Reliability of Diverse Systems,” in IEEE Transactions on Software Engineering, Volume: 29, NO: 4, April 2003, pp.345-359.

[64] P. T. Popov and L. Strigini, “Conceptual Models for the Reliability of Diverse Systems - New Results,” in FTCS ’98, pp. 80-89.

[65] B. Littlewood, P. Popov and L. Strigini, "Assessing the Reliability of Diverse Fault-Tolerant Software-based Systems," in 65 Science, vol.40, 2002, pp.781-796.

[66] P.G. Bishop, Adelard, “Review of Software Design Diversity,” Software Fault Tolerance, edited by M. Lyu, John Wiley & Sons, 1995.

[67] A. Avizienis, M.R. Lyu, and W. Schutz, “In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software,” in Proc. IEEE CS 18th Int’l Symp. Fault-Tolerant Computing, IEEE CS Press, June 1988, pp. 15–22.

[68] U. Voges (Ed.), Software diversity in computerized control systems, Wien, Springer Verlag, 1988.

[69] D. Briere and P. Traverse, "Airbus A320/A330/A340 Electrical Flight Controls - A Family Of Fault-Tolerant Systems", in FTCS-23, Proc. 23rd International Symposium on Fault-Tolerant Computing, Toulouse, France, 22 - 24, 1993, pp. 616-623.

[70] H. Kantz and C. Koza, "The ELEKTRA Railway Signalling-System: Field Experience with an Actively Replicated System with Diversity", in FTCS-25, Proc. 25th IEEE Annual International Symposium on Fault -Tolerant Computing, Pasadena, California, 1995, pp. 453-458.

[71] D. Eckhardt, A. Caglayan, J. Knight, L. Lee, D. McAllister, M. Vouk and J. Kelly, “An Experimental Evaluation of Software Redundancy as a Strategy For Improving Reliability,” in IEEE Transactions On Software Engineering, Vol 17, NO. 7, July 1991.

[72] M. R. Lyu, “Software Reliability Measurements in N-Version Software Execution Environment,” in ISSRE'92, Proceedings 3rd International Symposium on Software Reliability Engineering, Research Triangle Park, North Carolina, October 8-10 1992, pp. 254-263.

[73] Trusted Computing Group. [online], (February 2004) [cited February 23 2004], Available at: https://www.trustedcomputinggroup.org/home

[74] HELP AMERICA VOTE ACT OF 2002, [online], (April 2003), [cited January 23 2004], Available at: http://fecweb1.fec.gov/hava/hava.htm

[75] N. Lynch, Distributed Algorithms, Morgan Kaufman Publishers, 1996.

[76] B. Littlewood and L. Strigini, “Redundancy and diversity in security,” [online document], (February 2003) [cited February 26 2004], Available at: http://www.csr.city.ac.uk/people/lorenzo.strigini/ls.papers/_03_FTsecurity/sec_FT_v19.pdf

[77] S. Asiri, "Open Source," in ACM SIGCAS, Computers and Society, Volume 32, Issue 5, March 2003.

[78] R. Anderson, "Security in Open versus Closed Systems - The Dance of Boltzmann, Coase and Moore," in Open Source Software: Economics, Law and Policy Conference '02, June 2002.

[79] M. Jakobsson, A. Juels and R. Rivest, "Making mix nets robust for electronic voting by randomized partial checking," in USENIX '02, p. 339-353, February 2002.

[80] M. K. Joseph. Architectural Issues in Fault-Tolerant, Secure Computing Systems. PhD thesis, Computer Science Department, University of California, Los Angeles, June 1988.

[81] D. Chaum, "Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms," in Communications of the ACM, 24(2):84-88, February 1981.

[82] R. Butler and G. Finelli, "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software," in IEEE Transactions on Software Engineering, vol. 19, no. 1, Jan. 1993, pp 3-12.

[83] D. Eckhardt and L. Lee, "A theoretical basis for the analysis of multiversion software subject to coincident errors," in IEEE Transactions on Software Engineering, SE-11, p. 1511-1517, 1985.

[84] U.S. Census Bureau, Table 12. Reasons for Not Voting, by Sex, Age, Race and Hispanic Origin, and Education: November 1998, [online article], (July 2000) [cited January 29 2004], Available at: http://www.census.gov/population/socdemo/voting/cps1998/tab12.txt

[85] Americans With Disabilities Act Of 1990, [online], (January 1990) [cited February 17 2004], Available at: http://www.usdoj.gov/crt/ada/pubs/ada.txt

[86] A. Avizienis, P. Gunningberg, J. P. J. Kelly, L. Strigini, P. J. Traverse, K. S. Tso and U. 68, "The UCLA Dedix System: A Distributed Testbed for Multiple-Version Software", 15th Int. Symposium on Fault-Tolerant Computing, Michigan, June 1985, pp. 126-134.

Appendix A

Module Specifications

Ballot Request SubModule specification
(C) 2004 Soyini Liburd / MIT Media Laboratory

Purpose:

The Ballot Request SubModule presents, to the Ballot Server Modules, the information required to determine the correct ballot and interface definition for that election and precinct. (The Ballot Server Modules then send said ballot and interface definition to the User Interface Module.)
The Ballot Request Modules run locally on the Voting Machine - the same machine that contains the User Interface Module that will eventually display the ballot.
Note, the Ballot Request is called a SubModule (dependent on the User Interface Module) because it runs locally, on the same machine as the User Interface Module,
and the ballot it requests is eventually displayed on its Master User Interface Module.

Initialization:

The Ballot Request SubModule has access to the location of a XML config file containing the ids, locations (hosts & ports) and public keys of its (master) User Interface Module and the Ballot Server Modules.

The Ballot Request SubModule also has access to its own private key which is stored locally, in secure memory managed by a trusted computing base and accessible only to that Ballot Request Module.

The Ballot Request SubModule is initialized with the election id and precinct id relevant to the Voting Machine on which the Ballot Request SubModule resides.

Operation:

The Ballot Request SubModule compiles the election id and precinct id, along with the local (master) User Interface's id into an XML document. The Ballot Request Module
sends this XML document to each Ballot Server Module listed in the config file.

Expected Communications (XML):

Output Message:

Ballot Request SubModule to Ballot Server Module:

```
<!ELEMENT command (execute)>
<!ELEMENT execute (get_ballots)>
<!ELEMENT get_ballots (election_id, precinct_id, ui_id)>
<!ELEMENT election_id (#PCDATA)>
<!ELEMENT precinct_id (#PCDATA)>
<!ELEMENT ui_id (#PCDATA)>
```

Exception Handling: Terminate any open connections then cease operation.

Timing Constraints: BallotRequest SubModule should output within __ seconds after it successfully reads the cd contents.

Diversity Requirements: Independent implementation by programmer.

Ballot Server Module specification

Purpose:

The Ballot Server Module services requests for blank ballots and interface definitions.

Initialization:

The Ballot Server Module has access to the location of a XML config file containing the ids, locations (hosts & ports) and public keys of the Ballot Request SubModules and User Interface Modules.

The Ballot Server Module also has access to its own private key which is stored locally, in secure memory managed by a trusted computing base and accessible only to that Ballot Server Module.

The Ballot Server Module is initialized with the absolute start and end (date &) time of the election in milliseconds.

Operation:

The Ballot Server Module is responsible for delivering a blank ballot and interface definition to the UI for display.

The Ballot Server receieves ballot requests from Ballot Request SubModules. The ballot request is in the form of an XML document containing the election id and precinct id, as well as the id of the User Interface module to which the ballot and interface should be sent for display. The Ballot Server only services requests received during the voting period (start to close of the election), requests at any other time are ignored. The Ballot Server uses some global, trusted time source such as NIST internet time servers.

The Ballot Server waits for additional input (Ballot Request messages) for at most some Tmax seconds after receiving the first minF related messages. minF is some fraction of the maximum expected number of messages n, where n refers to the number of Ballot Request SubModules listed for the specified User Interface Module in the config file. minF has an absolute value greater than 1 to prevent a malicious module from sending a message extremely early and wrongfully suspending the Ballot Server Module's operation.
The Ballot Server Module must receive at least ceil(n/2) related messages by Tmax seconds, otherwise the Server performs no operation on the input (because a majority is impossible with fewer messages, so there isn't enough confidence in the correctness of the input).
Note that messages that are incorrectly formatted or unauthenticated, or repeat (related) messages from the same client, are immediately discarded.

After the receiving period, the module compares the related input messages it has received and chooses the message that occurs some majority (> n/2) of times as its official input message. The Ballot Server uses the chosen (election id, precinct id) message to retrieve the right blank ballot and interface definition for that precinct and election from its database. The database specification and implementation are left to the developer (the only requirement is that the correct ballot and interface definition be associated with each (precinct, election) pair. The Ballot Server sends the retrieved blank ballot and interface definition to the indicated User Interface for display.

The Ballot Server Module keeps a record of User Interface Modules it has sent ballots to.

Expected Communications (XML):

Input Messages:

BallotRequest Module to Ballot Server Module:

```
<!ELEMENT command (execute)>
<!ELEMENT execute (get_ballot)>
<!ELEMENT get_ballot (election_id, precinct_id, ui_id)>
<!ELEMENT election_id (#PCDATA)>
<!ELEMENT precinct_id (#PCDATA)>
<!ELEMENT ui_id (#PCDATA)>
```

Output Messages:

Ballot Server Module to User Interface Module:

```
<!ELEMENT command (execute)>
<!ELEMENT execute (display_ballots)>
<!ELEMENT display_ballot (interface, ballots)>
<!ELEMENT interface (#PCDATA)>
<!ELEMENT ballots (one_ballot)+>
        <!ELEMENT one_ballot(ballot_id, creation_date, l_modification_date, author_name,
                language, state, county, precinct, election_id, interpretor_id, ballot_items) >
        <!ELEMENT ballot_id (#PCDATA)>
        <!ELEMENT creation_date (#PCDATA)>
        <!ELEMENT l_modification_date (#PCDATA)>
        <!ELEMENT author_name (#PCDATA)>
        <!ELEMENT language (#PCDATA)>
        <!ELEMENT state (#PCDATA)>
        <!ELEMENT county (#PCDATA)>
        <!ELEMENT precinct (#PCDATA)>
        <!ELEMENT election_id (#PCDATA)>
        <!ELEMENT interpretor_id (#PCDATA)>
        <!ELEMENT ballot_items (ballot_item)+>
        <!ELEMENT ballot_item (ballot_item_id, is_issue_p, is_preferential_p, issue_name,
                        ballot_item_option+) >
        <!ELEMENT ballot_item_id (#PCDATA)>
        <!ELEMENT is_issue_p (#PCDATA)>
        <!ELEMENT is_preferential_p (#PCDATA)>
        <!ELEMENT issue_name (#PCDATA)>
        <!ELEMENT ballot_item_option (option_id, option_name,is_option_selected_p)>
        <!ELEMENT option_id (#PCDATA)>
        <!ELEMENT option_name (#PCDATA)>
        <!ELEMENT is_option_selected_p (#PCDATA)>
```

Exception Handling: Terminate any open connections then cease operation.

Timing Constraints: Ballot Server Module should output within __ seconds.

Diversity Requirements: Independent implementation by programmer.

User Interface Module specification

Purpose:

The User Interface Module displays a ballot for the user. It serves as the means for voters to make their selections. Additionally, it receives a copy of the ballot actually stored for that vote and displays this copy to the user, allowing the user to verify that his/her vote was actually cast as intended.

Operation:

Initialization:

The User Interface Module has access to the location of a XML config file containing the ids, public keys and locations (hosts & ports) of the Aggregator Modules, Listener Modules and Ballot Server Modules.

The User Interface Module also has access to its own private key which is stored locally, in secure memory managed by a trusted computing base and accessible only to that User Interface Module.

The User Interface module receives from each Ballot Server Module, a blank ballot for that precinct and election, as well as an interface definition consistent with that precinct and election. The interface definition describes the way in which the UI is to render the ballot.

The User Interface module waits for additional input (ballot+interface) messages for at most some Tmax1 seconds after receiving the first ceil(n/2) related messages, where n refers to the number of Ballot Server modules listed in the config file. All related messages will have the same interface definition and the same ballot (except for possibly the ballot id).
Note that messages that are incorrectly formatted or unauthenticated, or repeat (related) messages from the same client, are immediately discarded.
After the receiving period, the module compares the related input messages it has received and chooses the message that occurs some majority (> n/2) of times as its official input message.
Note that this particular ballot and interface definition is used by the UI throughout the election. A fresh instance of this ballot is displayed for each new voter, and by request from the current voter.

Function:

The User Interface module generates a unique ballot_id and displays the blank ballot to the voter as specified by the elected interface definition. The voter then fills in the ballot, verifies its contents, and submits it.
The Listener Sub Modules are responsible for capturing the completed ballot submitted by the user, rendering it into XML, encrypting it with the aggregator public keys and passing the encrypted ballots to the Registration Server Modules.

The User Interface Module is also responsible for receiving copies of the ballots, tentatively stored for that voter, from the Aggregator modules. These ballots are encrypted with UI Module's public key.
Again, the UI waits some maximum period Tmax2 seconds after receiving the first ceil(n/2) related messages, where n refers to the number of Aggreagator modules listed in the config file. Here, related messages should be identical encrypted ballots. Again messages that are incorrectly formatted or unauthenticated, or repeat (related) messages from the same client, are immediately discarded.

After the waiting period, the UI Module displays to the voter, the completed ballot that was sent in a majority of the input messages. At this point, the voter can indicate whether the ballot displayed for verification is the ballot that he/she intended to cast. The ability to re-vote (the part of the user interface that facilitates requesting a new ballot) should only be available to the voter for a limited amount of time. This restriction limits the possibility of subsequent exploitation if a voter leaves before making the confirmation. The Listeners are again responsible for communicating the voter's response to the rest of the system. If the voter verifies that the ballot displayed is his/her intent, the voter's interaction with the system ends here. The Listeners would then report the voter's approval
to the Aggregator Modules and Registration Server Modules.

If however, the ballot is not what the voter intended to cast, the voter indicates whether there was a human error (on the part of the voter) or a system error, and can, if desired, request a new ballot. Once the voter indicates an error, the Listener Modules will relay this to the Aggregator Modules so that they can invalidate the faulty ballot (so that it can be removed from official counts, etc).

The Listeners would relay any request for a new ballot to the User Interface Module.
After receiving such "fresh ballot" requests from Listener Modules and electing an official input from these, the User Interface Module determines whether the "fresh ballot" request is valid (whether a majority of the Listeners claim that a new ballot was requested). If a "fresh ballot" was indeed requested, the User Interface Module generates a new ballot_id and displays a fresh instance of the ballot.

Expected Communications (XML):

Input Messages:

From Ballot Server Module to User Interface Module:
New Ballot displayed to someone on their first vote attempt:

```
<!ELEMENT command (execute)>
<!ELEMENT execute (display_ballots)>
<!ELEMENT display_ballot (interface, ballots)>
<!ELEMENT interface (#PCDATA)>
<!ELEMENT ballots (one_ballot)+>
        <!ELEMENT one_ballot(ballot_id, creation_date, l_modification_date, author_name,
                language, state, county, precinct, election_id, interpretor_id, ballot_items) >
        <!ELEMENT ballot_id (#PCDATA)>
        <!ELEMENT creation_date (#PCDATA)>
        <!ELEMENT l_modification_date (#PCDATA)>
        <!ELEMENT author_name (#PCDATA)>
        <!ELEMENT language (#PCDATA)>
        <!ELEMENT state (#PCDATA)>
        <!ELEMENT county (#PCDATA)>
        <!ELEMENT precinct (#PCDATA)>
        <!ELEMENT election_id (#PCDATA)>
        <!ELEMENT interpretor_id (#PCDATA)>
        <!ELEMENT ballot_items (ballot_item)+>
        <!ELEMENT ballot_item (ballot_item_id, is_issue_p, is_preferential_p, issue_name,
ballot_item_option+) >
        <!ELEMENT ballot_item_id (#PCDATA)>
        <!ELEMENT is_issue_p (#PCDATA)>
        <!ELEMENT is_preferential_p (#PCDATA)>
        <!ELEMENT issue_name (#PCDATA)>
```

<!ELEMENT ballot_item_option (option_id, option_name,is_option_selected_p)>
<!ELEMENT option_id (#PCDATA)>
<!ELEMENT option_name (#PCDATA)>
<!ELEMENT is_option_selected_p (#PCDATA)>

From Aggregator Module to User Interface Module:

<!ELEMENT command (execute)>
<!ELEMENT execute (verify)>
<!ELEMENT verify (encrypted_ballot)>
<!ELEMENT encrypted_ballot (#PCDATA)>

Listener SubModule to User Interface Module (to obtain a fresh ballot for second chance voting):

<!ELEMENT command (execute)>
<!ELEMENT execute (fresh_ballot)>
<!ELEMENT fresh_ballot (#PCDATA)>

Exception Handling: terminate any open connections then cease operation

Listener SubModule specification

Note:

The Listener is called a SubModule (dependent on the User Interface Module) because it runs locally, on the same machine as the User Interface Module, and is dependent on the User Interface Module for initiation.

Purpose:

The Listener SubModule listens for activity at the UI, records the voter's ballot selections and the ballot id and renders this information in XML. This completed XML ballot is encrypted using the public keys of the Aggregator Modules and sent to the Registration Server Modules.

Initialization:

The Listener SubModule has access to the location of a XML config file containing the ids, locations (hosts & ports) and public keys of the Registration Server Modules and Aggregator Modules. The Listener SubModule also has access to its own private key which is stored locally, in secure memory managed by a trusted computing base and accessible only to that Listener SubModule.

Operation:

The User Interface module starts up each Listener SubModule at which point the Listener module is actively listening for user input at the UI.

When the user presses "Enter Vote" at the UI, the Listener collects the voter's selections and the ballot id from the User Interface. For each Aggregator Module listed in the config file, the Listener SubModule encrypts a copy of the voter's completed ballot using that Aggregator Module's public key.
The Listener Module will also read the user's encrypted voter information from a read-only compact disc (associated with only one voter and mailed to the voter). This voter information is encrypted with the Registration Modules public keys.
The Listener SubModule then sends the encrypted ballots and voter information to the Registration Server Modules.

The ballot that has been stored by the Aggregator modules are again presented to the voter so that the voter can indicate whether or not the ballot stored is the ballot he/she intended to cast.
The Listener SubModule is responsible for communicating the voter's response to the rest of the system.

If the voter verifies that the ballot displayed is his/her intent, the Listener SubModule communicates this approval to the Aggregator Modules. The Listener then informs the Registration Server Modules that the voter has submitted and finalized a ballot, by resending the voter's encrypted voter information along with a note that the associated voter has approved a ballot.

If however, the ballot is not what the voter intended to cast, the voter indicates whether there was a human error (on the part of the voter) or a system error, and can, if desired, request a new ballot. The error report (human or system error indication) is sent to servers that store such information for quality control purposes. The Listener SubModules report that the ballot was faulty to the Aggregator Modules so that they can invalidate the faulty ballot (so that it can be removed from official counts, etc).
If the voter requests a new ballot, the Listener SubModule would relay this request to the User Interface Module.

Expected Communications (XML):

Output Message:

Listener SubModule to Registration Server Module: Voter Check In
<!ELEMENT command (execute)>
<!ELEMENT execute (approve_ballot)>
<!ELEMENT approve_ballot (ui_id, voter_info, encrypted_ballot_packages)>
<!ELEMENT ui_id (#PCDATA)>
<!ELEMENT voter_info (#PCDATA)>
<!ELEMENT encrypted_ballot_packages (encrypted_ballot_package)+>
<!ELEMENT encrypted_ballot_package (agg_id, encrypted_ballot)>
<!ELEMENT agg_id (#PCDATA)>
<!ELEMENT encrypted_ballot (#PCDATA)>

Listener SubModule to Aggregator Module:
<!ELEMENT command (execute)>
<!ELEMENT execute (finalize_ballot)>
<!ELEMENT finalize_ballot (encrypted_ballot)>
<!ATTLIST finalize_ballot status (VALID|INVALID) #REQUIRED>
<!ELEMENT encrypted_ballot (#PCDATA)>

Listener SubModule to Registration Server Module: Finalize Voter
<!ELEMENT command (execute)>
<!ELEMENT execute (finalize_voter)>
<!ELEMENT finalize_voter (voter_info)>
<!ATTLIST finalize_voter status (DONE) #REQUIRED>
<!ELEMENT voter_info (#PCDATA)>

Listener SubModule to Error Servers:
<!ELEMENT command (execute)>
<!ELEMENT execute (report_error)>
<!ELEMENT report_error (error_type)>
<!ELEMENT error_type (#PCDATA)>

Listener SubModule to User Interface Module:
<!ELEMENT command (execute)>
<!ELEMENT execute (fresh_ballot)>
<!ELEMENT fresh_ballot (#PCDATA)>

Exception Handling: Terminate any open connections.

Timing Constraints: The Listener Module should output within __ seconds from the time the user hits enter vote.

Diversity Requirements: Independent implementation by programmer.

Registration Server Module specification

Purpose:

The Registration Server Module manages the registration data and check-in procedures for the election. The Registration Server Module holds a private key with which it signs ballots. In addition to the Registration Server signing key, third parties(the political parties, SIG's, etc) may have the option to include a witness module so that they can sign ballots as well. The server returns a ballot package devoid of voter name or other identifying information, and includes the signatures produced. The Registration Server Module will be set up on three ports, one for servicing Ballot Request Modules, another for servicing Listener Modules and a third for servicing Witness Modules.

Initialization:

The Registration Server has access to the database of registration data, defined by the Registration Data Model[voter-registration.sql].

The Registration Server Module also has access to the location of a XML config file containing the ids, locations (hosts & ports) and public keys of the User Interface Modules, the Listener Modules, the Witness Modules and the Aggregator Modules.

The Registration Server Module also has access to its own private key which is stored locally, in secure memory managed by a trusted computing base and accessible only to that Registration Server Module.

Operation:

The functional requirements for initialization and maintenance of the registration database are not enumerated specifically because they do not interact with the rest of the system. For instance, the database does not specify the internal representation of data, although, for any practical purpose a reliable RDBMS such as Oracle, Informix, DB2, etc is expecteded.

The Registration Server receives encrypted ballot packages from Listener Modules. An encrypted ballot package contains an Aggregator Module ID as well as an encrypted ballot, encrypted with that Aggregator Module's public key. The Registration Server waits for additional input messages for at most some Tmax2 seconds after receiving the first ceil(n/2) related messages, where n here refers to the number of Listener modules listed for the specified User Interface Module in the config file.
Here related messages have the same associated User Interface and ballot id (the unique ballot id distinguishes the encrypted ballot). Again messages that are incorrectly formatted or unauthenticated, or repeat (related) messages from the same client, are immediately discarded.

The Registration Server chooses the ballot_packages message it has received a majority of times (among the related ballot_packages) as the official input ballot_packages message. The Registration Server Module then extracts and decrypts the voter information from that message. The voter information extracted includes the election_id of the election that the voter is certified to vote in, the precinct_id that the voter is certified to vote at, and the voter's voter_registration_id which is a unique identifier that associates that voter with a record in the registration database if in fact the voter should be allowed to vote.

The Registration Server uses the voter_registration_id and the precinct_id to check the registration database to see if the voter is in fact registered to vote. If the voter is registered to vote and has not yet submitted a valid verified ballot, the Registration Server

signs the encrypted ballot in each encrypted_ballot_package in the elected input message. If the voter is registered to vote and has not yet been checked-in, the Registration Server checks in the voter using the voter_registration_id, election_id and precinct_id.

At this point, the Registration Server forwards each encrypted ballot, along with the Registration Server's signature of that encrypted ballot, to the Witness Modules. The Witness Modules verify the Registration Server's signature and then respond with their own signature of the encrypted ballot (if the Registration Server's signature is valid). These Witness signatures are appended to the encrypted ballot and the Registration Server's own signature in a signature package that consists of an Aggregator id, a ballot encrypted with that particular Aggregator Module's public key, and all the signatures related to that encrypted ballot.

After the Registration Server Module has finished communicating with the Witnesses, it sends each signature package to the Vote Aggregator Module with the relevant public key (the public key that was used to encrypt the ballot in that package). The id of the User Interface module that displayed that particular ballot is also included in the message to the Vote Aggregator Modules.

Format of (encrypted) voter_info:

Note that the voter_info contains minimally the voter_registration_id, but may include any of the other fields as well.

```
<!ELEMENT voter_info (voter_registration_id, first_names, last_name, address, address_2, city,
                      state, zip)>
<!ELEMENT voter_registration_id (#PCDATA)>
<!ELEMENT first_names (#PCDATA)>
<!ELEMENT last_name (#PCDATA)>
<!ELEMENT address   (#PCDATA)>
<!ELEMENT address_2 (#PCDATA)>
<!ELEMENT city      (#PCDATA)>
<!ELEMENT state     (#PCDATA)>
<!ELEMENT zip       (#PCDATA)>
```

Expected Communications (XML):

Input Messages:

Listener SubModule to Registration Server Module: Voter Check In

```
<!ELEMENT command (execute)>
<!ELEMENT execute (approve_ballot)>
<!ELEMENT approve_ballot (ui_id, voter_info, encrypted_ballot_packages)>
<!ELEMENT ui_id (#PCDATA)>
<!ELEMENT voter_info (#PCDATA)>
<!ELEMENT encrypted_ballot_packages (encrypted_ballot_package)+>
<!ELEMENT encrypted_ballot_package (agg_id, encrypted_ballot)>
<!ELEMENT agg_id (#PCDATA)>
<!ELEMENT encrypted_ballot (#PCDATA)>
```

Listener SubModule to Registration Server Module: Finalize Voter

```
<!ELEMENT command (execute)>
<!ELEMENT execute (finalize_voter)>
<!ELEMENT finalize_voter (voter_info)>
<!ATTLIST finalize_voter status (DONE) #REQUIRED>
<!ELEMENT voter_info (#PCDATA)>
```

Witness Module to Registration Server Module:

```
<!ELEMENT results (witness_result)>
<!ELEMENT witness_result (signed_ballot_package)>
<!ATTLIST witness_result status (FAILURE|SUCCESS) #REQUIRED>
<!ELEMENT signed_ballot_package (agg_id, encrypted_ballot, encrypted_signature_packages)>
<!ELEMENT agg_id (#PCDATA)>
<!ELEMENT encrypted_ballot (#PCDATA)>
<!ELEMENT encrypted_signature_packages (encrypted_signature_package)+>
<!ELEMENT encrypted_signature_package (signer_id, encrypted_signature)>
<!ATTLIST encrypted_signature_package signer_type (regserver|witness) #REQUIRED>
<!ELEMENT signer_id (#PCDATA)>
<!ELEMENT encrypted_signature (#PCDATA)>
```

Output Messages:

Registration Server Module to Witness Module:

```
<!ELEMENT command (execute)>
<!ELEMENT execute (witness)>
<!ELEMENT witness (signed_ballot_package)>
<!ELEMENT signed_ballot_package (agg_id, encrypted_ballot, encrypted_signature_packages)>
<!ELEMENT agg_id (#PCDATA)>
<!ELEMENT encrypted_ballot (#PCDATA)>
<!ELEMENT encrypted_signature_packages (encrypted_signature_package)+>
<!ELEMENT encrypted_signature_package (signer_id, encrypted_signature)>
<!ATTLIST encrypted_signature_package signer_type (regserver|witness) #REQUIRED>
<!ELEMENT signer_id (#PCDATA)>
<!ELEMENT encrypted_signature (#PCDATA)>
```

Registration Server Module to Vote Aggregator Module:

```
<!ELEMENT command (execute)>
<!ELEMENT execute (aggregate)>
<!ELEMENT aggregate (ui_id, signed_ballot_package)>
<!ELEMENT ui_id (#PCDATA)>
<!ELEMENT signed_ballot_package (agg_id, encrypted_ballot, encrypted_signature_packages)>
<!ELEMENT agg_id (#PCDATA)>
<!ELEMENT encrypted_ballot (#PCDATA)>
<!ELEMENT encrypted_signature_packages (encrypted_signature_package)+>
<!ELEMENT encrypted_signature_package (signer_id, encrypted_signature)>
<!ATTLIST encrypted_signature_package signer_type (regserver|witness) #REQUIRED>
<!ELEMENT signer_id (#PCDATA)>
<!ELEMENT encrypted_signature (#PCDATA)>
```

Witness Module Specification

Purpose:

To permit an independent agency to examine the hash of an encrypted ballot.
And produce a digital signature to attach to the ballot.

Initialization:

The Witness Module has access to the location of a XML config file containing the public keys and locations (hosts & ports) of the Registration Server Modules.

The Witness Module also has access to its own private key which is stored locally, in secure memory managed by a trusted computing base and accessible only to that Witness Module.

Operation:

Whenever a new encrypted ballot is received, once it is validated by the registration system, each witness in the registration server's witness list is contacted with a hash of the ballot. The Witness module verifies that the Registration Server's signature of the encrypted ballot is valid. If the Registration Server's signature is valid, the Witness creates a timestamp(to millisecond) and digitally signs the combination of the timestamp and the hashed ballot with the Witness's private key. If the Registration Server's signature is invalid, or there is some other problem, the Witness module responds with witness_result status="FAILURE"

Expected Communications (XML):

Input Message:

Registration Server Module to Witness Module:

```
<!ELEMENT command (execute)>
<!ELEMENT execute (witness)>
<!ELEMENT witness (signed_ballot_package)>
<!ELEMENT signed_ballot_package (agg_id, encrypted_ballot, encrypted_signature_packages)>
<!ELEMENT agg_id (#PCDATA)>
<!ELEMENT encrypted_ballot (#PCDATA)>
<!ELEMENT encrypted_signature_packages (encrypted_signature_package)+>
<!ELEMENT encrypted_signature_package (signer_id, encrypted_signature)>
<!ATTLIST encrypted_signature_package signer_type (regserver|witness) #REQUIRED>
<!ELEMENT signer_id (#PCDATA)>
<!ELEMENT encrypted_signature (#PCDATA)>
```

Output Message:

Witness Module to Registration Server Module:

```
<!ELEMENT results (witness_result)>
<!ELEMENT witness_result (signed_ballot_package)>
<!ATTLIST witness_result status (FAILURE|SUCCESS) #REQUIRED>
<!ELEMENT witness_signed_ballot_package (agg_id, encrypted_ballot, digest_timestamp,
                                         encrypted_signature_package)>
<!ELEMENT agg_id (#PCDATA)>
<!ELEMENT encrypted_ballot (#PCDATA)>
<!ELEMENT digest_timestamp (#PCDATA)>
```

```
<!ELEMENT encrypted_signature_package (signer_id, encrypted_signature)>
<!ATTLIST encrypted_signature_package signer_type (witness) #REQUIRED>
<!ELEMENT signer_id (#PCDATA)>
<!ELEMENT encrypted_signature (#PCDATA)>
```

Exception Handling: responds with witness_result attribute status="FAILURE"

Timing Constraints: The Witness Module should return within __ seconds (otherwise it's return value is ignored).

Diversity Requirements: Independent implementation by programmer.

Aggregator Module specification

Purpose:

The Aggregator Module is responsible for storing and verifying all of the data related to votes cast. The Aggregator will be set up on two ports, one for receiving ballots and the other for verifying ballots.

Initialization:

The Vote Aggregator Module has access to the location of a XML config file containing the ids, public keys and locations (hosts & ports) of the Registration Server, Listener and User Interface Modules.

The Aggregator Module also has access to its own private key which is stored locally, in secure memory managed by a trusted computing base and accessible only to that Aggregator Module.

Operation:

The Aggregator will receive encrypted signed ballot packages from the Registration Server Modules.
In addition to the ballot itself, which is encrypted using this server's public key, the ballot package contains digital signatures of the Registration Server, and those of the signing Witnesses.
The Aggregator waits for additional input (Register Server Module messages) for at most some Tmax1 seconds after receiving the first ceil(n/2) related messages, where n refers to the number of Register Server modules listed in the config file. All related messages will have the same encrypted ballot package. Note that messages that are incorrectly formatted or unauthenticated, or repeat (related) messages from the same client, are immediately discarded.
After the receiving period, the Aggregator module compares the related input messages it has received and chooses the message that occurs some majority (> n/2) of times as its official input message. The Aggregator then unpacks the chosen ballot package: it verifies all of the included digital signatures, decrypts the ballot with its private key, and stores the ballot's data in its database.

To verify that the vote stored is the vote cast, the Aggregator retrieves the ballot from its database, encrypts the ballot with public key of the User Interface module indicated in the message from the Registration Server Module, and sends the encrypted ballot to that User Interface Module.
The User Interface module then displays the ballot and the voter indicates whether or not this is the vote he / she intended to cast. The Listener Modules are responsible for communicating the voter's intention back to the Aggregator Module. The Listener Module does this by sending the Aggregator Module a message containing the completed ballot (encrypted) that the UI displayed for verification by the user, as well as a status attribute indicating whether that ballot was deemed VALID or INVALID.
The Aggregator waits for additional input (Listener Module messages) for at most some Tmax2 seconds after receiving the first ceil(n/2) related messages, where n refers to the number of Listener modules listed in the config file. All related messages will have the same ballot and status attribute value. Note that messages that are incorrectly formatted or unauthenticated, or repeat (related) messages from the same client, are immediately discarded.
After the receiving period, the Aggregator module compares the related input messages it has received and chooses the message that occurs some majority (> n/2) of times as its official input message.

If the finalize_ballot status of the elected input message is VALID, the aggregator checks to see whether the elected ballot is identical to the one it has stored in its database. If the elected ballot is the same as the aggregator's stored ballot, the aggregator knows that the vote has been successfully cast.
The Aggregator Module would then encrypt the ballot using the public keys of appropriate, authenticated storage servers (precinct, county, etc) and sends the ballot out to these servers.

If the finalize_ballot status of the elected input message is INVALID, the Aggregator Module is responsible for ensuring that its copy of that ballot (the ballot it has stored with that ballot's ballot id) is invalidated. That is, the Aggregator Module must ensure that its stored ballot is not included in any official count or declaration of votes.

Expected Communications (XML):

Input Messages:

Registration Server Module to Aggregator Module:

```
<!ELEMENT command (execute)>
<!ELEMENT execute (aggregate)>
<!ELEMENT aggregate (ui_id, signed_ballot_package)>
<!ELEMENT ui_id (#PCDATA)>
<!ELEMENT signed_ballot_package (agg_id, encrypted_ballot, encrypted_signature_packages)>
<!ELEMENT agg_id (#PCDATA)>
<!ELEMENT encrypted_ballot (#PCDATA)>
<!ELEMENT encrypted_signature_packages (encrypted_signature_package)+>
<!ELEMENT encrypted_signature_package (signer_id, encrypted_signature)>
<!ATTLIST encrypted_signature_package signer_type (regserver|witness) #REQUIRED>
<!ELEMENT signer_id (#PCDATA)>
<!ELEMENT encrypted_signature (#PCDATA)>
```

Listener SubModule to Aggregator Module:

```
<!ELEMENT command (execute)>
<!ELEMENT execute (finalize_ballot)>
<!ELEMENT finalize_ballot (encrypted_ballot)>
<!ATTLIST finalize_ballot status (VALID|INVALID) #REQUIRED>
<!ELEMENT encrypted_ballot (#PCDATA)>
```

Output Messages:

From Aggregator Module to User Interface Module:

```
<!ELEMENT command (execute)>
<!ELEMENT execute (verify)>
<!ELEMENT verify (encrypted_ballot)>
<!ELEMENT encrypted_ballot (#PCDATA)>
```

Exception Handling: terminate any open connections then cease operation

Timing Constraints: The Aggregator Module should return within __ seconds (otherwise it's return value is ignored).

Diversity Requirements: Independent implementation by programmer.

DTD Files

ballot.dtd

```
<!ELEMENT ballots (one_ballot)+>
        <!ELEMENT one_ballot(ballot_id, creation_date, l_modification_date, author_name,
                                language, state, county, precinct, election_id, interpretor_id,
                                ballot_items) >
                <!ELEMENT ballot_id (#PCDATA)>
                <!ELEMENT creation_date (#PCDATA)>
                <!ELEMENT l_modification_date (#PCDATA)>
                <!ELEMENT author_name (#PCDATA)>
                <!ELEMENT language (#PCDATA)>
                <!ELEMENT state (#PCDATA)>
                <!ELEMENT county (#PCDATA)>
                <!ELEMENT precinct (#PCDATA)>
                <!ELEMENT election_id (#PCDATA)>
                <!ELEMENT interpretor_id (#PCDATA)>

                <!ELEMENT ballot_items (ballot_item)+>
                <!ELEMENT ballot_item (ballot_item_id, is_issue_p, is_preferential_p, issue_name,
                                        ballot_item_option+) >
                                <!ELEMENT ballot_item_id (#PCDATA)>
                                <!ELEMENT is_issue_p (#PCDATA)>
                                <!ELEMENT is_preferential_p (#PCDATA)>
                                <!ELEMENT issue_name (#PCDATA)>
                                <!ELEMENT ballot_item_option (option_id,
                                                option_name,is_option_selected_p)>
                                        <!ELEMENT option_id (#PCDATA)>
                                        <!ELEMENT option_name (#PCDATA)>
                                        <!ELEMENT is_option_selected_p (#PCDATA)>
```

cipher-ballot.dtd

```
<!ELEMENT cipher-ballot(ciphertext, signature)>
<!ELEMENT ciphertext (#PCDATA)>
<!ELEMENT signature (#PCDATA)>
```

voter-registration.dtd

```
<!ELEMENT voter_reg_dataset (voter_registration_record)+)>

        <!ELEMENT voter_registration_record (record_id, first_names,
        last_name, address, address_2, city, state, zip, precinct_id,
        license_no, ssn)>


        <!ELEMENT record_id (#PCDATA)>
        <!ELEMENT first_names (#PCDATA)>
        <!ELEMENT last_name (#PCDATA)>
        <!ELEMENT address   (#PCDATA)>
        <!ELEMENT address_2 (#PCDATA)>
        <!ELEMENT city      (#PCDATA)>
        <!ELEMENT state     (#PCDATA)>
        <!ELEMENT zip       (#PCDATA)>
        <!ELEMENT precinct_id (#PCDATA)>
        <!ELEMENT license_no (#PCDATA)>
        <!ELEMENT record_id (#PCDATA)>
```

directory.dtd

```
<!ELEMENT dir (module)+>
<!ELEMENT module (id, public_keys, certificates, host, ports, services, submodules)>
<!ATTLIST module module_type CDATA #REQUIRED>
<!ELEMENT id (#PCDATA)>
<!ELEMENT public_keys (public_key)*>
<!ELEMENT public_key (#PCDATA)>
<!ATTLIST public_key owner_id CDATA #REQUIRED>
<!ATTLIST public_key function CDATA #REQUIRED>
<!ELEMENT certificates (certificate)*>
<!ELEMENT certificate (#PCDATA)>
<!ATTLIST certificate owner_id CDATA #REQUIRED>
<!ATTLIST certificate signer CDATA #REQUIRED>
<!ELEMENT host (#PCDATA)>
<!ELEMENT ports (port)*>
<!ELEMENT port (#PCDATA)>
<!ATTLIST port function CDATA #REQUIRED>
<!ELEMENT services (service)+>
<!ELEMENT service (#PCDATA)>
<!ATTLIST service service_name CDATA #REQUIRED>
<!ELEMENT submodules (submodule)*>
<!ELEMENT submodule (id, public_key, certificates, host, ports, services)>
<!ATTLIST submodule submodule_type CDATA #REQUIRED>
```

authenticationToken.dtd

```
<!ELEMENT authenticate_token (election_id, precinct_id, voter_information_packages,
election_signature)>
<!ELEMENT election_id (#PCDATA)>
<!ELEMENT precinct_id (#PCDATA)>
<!ELEMENT voter_information_packages (voter_information_package)+>
<!ELEMENT voter_information_package (reg_id, encrypted_voter_information)>
<!ELEMENT reg_id (#PCDATA)>
<!ELEMENT encrypted_voter_information (#PCDATA)>
<!ELEMENT election_signature (#PCDATA)>
```

Appendix B

Script to calculate PFD estimates according to a simplification of the Model in Chapter 4.

```
/** ReliabilityEst.java
        Author: Soyini Liburd.

        Simple, informal class based on a simplified version of
        the SAVE reliabilty model.  Created to allow quick calculation
        of an estimate of the Probability of Failure of a
        multi-stage, N-version system, constrained as described
        in Chapter 4 of the associated thesis.
        */

public class ReliabilityEst{

        //N number of modules per stage
        private int N=10;
        private double Nd = 10.0;
        //M number of stages
        private int M=6;
        //I number of fault types
        private int I=20;
        //c pr channel failure
        private double c=.05;
        //p pr that a module contains a given fault
        private double p = .4;
        //q pr that a given fault is "hit
        private double q = .0;
        //h pr that a demand hits a fault
        private double h = .8;

        public static void main(String[] args){
                ReliabilityEst est = new ReliabilityEst();

                if(args == null){
                        est.test();
                        return;
                }

                if(args.length ==1){
                        est.p=(new Double(args[0])).doubleValue();
                }
                else if(args.length ==2){
                        est.setcpVals((new Double(args[0])).doubleValue(),
                                                        (new Double(args[1])).doubleValue());
                }
                else if(args.length ==3){
                        est.setNcpVals((new Integer(args[0])).intValue(),
                                                        (new Double(args[1])).doubleValue(),
                                                         (new Double(args[2])).doubleValue());
                }
                else if (args.length ==4){
                        est.setNcphVals((new Integer(args[0])).intValue(),
```

```
                                                        (new Double(args[1])).doubleValue(),
                                                        (new Double(args[2])).doubleValue(),
                                                         (new Double(args[3])).doubleValue());
            }
            else if (args.length ==5){
                        est.setNIcphVals((new Integer(args[0])).intValue(),
                                                        (new Integer(args[1])).intValue(),
                                                         (new Double(args[2])).doubleValue(),
                                                          (new Double(args[3])).doubleValue(),
                                                           (new Double(args[4])).doubleValue());
            }
            else if (args.length ==6){
                        est.setAllVals((new Integer(args[0])).intValue(),
                                                       (new Integer(args[1])).intValue(),
                                                        (new Integer(args[2])).intValue(),
                                                         (new Double(args[3])).doubleValue(),
                                                          (new Double(args[4])).doubleValue(),
                                                           (new Double(args[5])).doubleValue());
            }
            est.test();
}

public void test(){
            System.out.println("N-version Pr Failure: "+RelEst(N));
            System.out.println("1-version Pr Failure: "+RelEst(1));
}

public ReliabilityEst(){}

public double stuffCalc(int n){
            int ceiln_2 = (new Double(Math.ceil((double)n/2))).intValue();
            int floorn_2p1 = (new Double(Math.floor((double)n/2))).intValue() + 1;
            int rmin = (new Double(Math.ceil(floorn_2p1))).intValue();

            double V = Vcalc(n, rmin, n, c);
            double S = sum_binomial(n, V, n, ceiln_2);

            return S + ((1 - S) * O1calc(n));

}

public double O1calc(int n){
            double o1 = 0.0;
            int ceiln_2 = (new Double(Math.ceil((double)n/2))).intValue();
            q=h/((double)I);

            for(int i=0; i<I; i++){
                        o1 = o1 + (q*sum_binomial(n, p, n, ceiln_2));
            }

            return o1;
}

public double Vcalc(int Rmax, int Rmin, int n, double cv){
            double dN_2 = (double)n/2.0;
            int ceilN_2 = (new Double(Math.ceil(dN_2))).intValue();
```

```
		int floorN_2 = (new Double(Math.floor(dN_2))).intValue();

		double inv_ceilN_2 = 1.0/(double)ceilN_2;
		double v = 0.0;
		double cr_n = cv*(((double)Rmin)/((double)n));

		int min = 0;

		for(int R=Rmin; R<=Rmax; R++){
			min = (new Double(Math.ceil((double)R - floorN_2))).intValue();
			v = v + (sum_binomial(n, cr_n, R, min)*inv_ceilN_2);
		}
		return v;
	}

	public double RelEst(int n){
		int ceiln_2 = (new Double(Math.ceil((double)n/2))).intValue();
		double Om = O1calc(n);

		double stuff = stuffCalc(n);
		for(int i=2;i<M;i++){
			Om = Om + (1.0 - Om)*stuff;
		}
		return Om;
	}

	public double sum_binomial(int n, double p, int max, int min){
		double sum = 0.0;

		for(int i=min; i<=max; i++)
			sum = sum + binomial(n, p, i);
		return sum;
	}

	public double weight_sum_binomial(double w, int n, double p, int max, int min){
		double sum = 0.0;

		for(int i=min; i<=max; i++)
			sum = sum + binomial(n, p, i)/w;
		return sum;
	}

	public double binomial(int n, double p, int k){
		return bincoeff(n, k) * Math.pow(p, k) * Math.pow((1-p), (n-k));
	}

	public int bincoeff(int n, int k) {
	 if (k == 0) return 1;
	 if (n==k) return 1;
	 return bincoeff(n-1, k-1) + bincoeff(n-1,k);
}

	public void setAllVals(int Nv, int Mv, int Iv, double cv, double pv, double hv){
		N=Nv;
		Nd= (double)N;
		M=Mv;
```

```
                I=Iv;
                c=cv;
                p=pv;
                h=hv;
        }

        public void setNIcphVals(int Nv, int Iv, double cv, double pv, double hv){
                N=Nv;
                Nd= (double)N;
                I=Iv;
                c=cv;
                p=pv;
                h=hv;
        }

        public void setNcphVals(int Nv, double cv, double pv, double hv){
                N=Nv;
                Nd= (double)N;
                c=cv;
                p=pv;
                h=hv;
        }


        public void setNcpVals(int nv, double cv, double pv){
                N=nv;
                c=cv;
                p=pv;
        }

        public void setcpVals(double cv, double pv){
                c=cv;
                p=pv;
        }

        public void setpVal(double pv){
                p=pv;
        }

}
```