Secure and Fault-Tolerant Voting in Distributed Systems[*]

Ben Hardekopf, Kevin Kwiat, Shambhu Upadhyaya[†]
Air Force Research Laboratory
AFRL/IFGA
525 Brooks Rd.
Rome, NY 13441-4505
{hardekob, kwiatk}@rl.af.mil; shambhu@cse.buffalo.edu

Abstract – Concerns about both security and fault-tolerance have had an important impact on the design and use of distributed information systems in the past. As such systems become more prevalent, as well as more pervasive, these concerns will become even more immediately relevant. From national defense, to commercial interests, to private citizens, distributed systems are making an ever greater impact on our lives.

We will focus here on integrating security and fault-tolerance into one, general-purpose protocol for secure distributed voting. Distributed voting is a well-known fault-tolerance technique [4]. For the most part, however, security had not been a concern in systems that used voting. More recently, several protocols have been proposed to shore up this lack. These protocols, however, have limitations which make them particularly unsuitable for many aerospace applications, because those applications require very flexible voting schemes (e.g., voting among real-world sensor data).

We present a new, more general voting protocol that reduces the vulnerability of the voting process to both attacks and faults. The algorithm is contrasted with the traditional 2-phase commit protocols typically used in distributed voting and with other proposed secure voting schemes. Our algorithm is applicable to exact and inexact voting in networks where atomic broadcast and predetermined message delays are present, such as local area networks. For wide area networks without these properties, we describe yet another approach that satisfies our goals of obtaining security and fault tolerance for a broad range of aerospace information systems.

* 0-7803-6599-2/01/$10.00 ©2001 IEEE

† Author affiliated with SUNY Buffalo; work performed while under the Air Force Research Laboratory/Information Directorate's 2000 Summer Faculty Research Program.

TABLE OF CONTENTS

1. INTRODUCTION
2. PAST WORK
3. THE ALGORITHM
4. DISCUSSION
5. WIDE-AREA NETWORKS
6. CONCLUSION

1. INTRODUCTION

As evidenced by recent news stories, the aerospace realm is not immune from the concerns of security as well as fault-tolerance. Although a recent hacking incident at NASA was deemed not to have endangered a shuttle mission ([1]), at the very least, it calls attention to the potential threats posed to the aerospace community. Networking on-board systems with those that are ground-based raises the concern of opening avenues for highly detrimental attacks. To be truly comprehensive, dependable aerospace information systems must tolerate faults that manifest themselves as a result of random phenomena or deliberate interference.

Historically, aerospace missions have been among the first to use fault tolerance. Early visionaries [2, 3] of the Apollo Program foresaw using redundancy to combat failure: by having a second spacecraft accompany a crew to the moon's surface, their return would be ensured should their primary landing vehicle be damaged. Redundancy at this level of granularity never came about because durability improvements in the eventual design of the Lunar Module reduced the risk to the crew of having only one of them on the moon.

Leaping to the present we see that special design techniques have been required for the computers used in aerospace missions. The Self-Testing And Repairing (STAR) Computer (1971), the Fault-Tolerant Multiprocessor (1975), the Fault-Tolerant Spaceborne Computer (FTSC) (1976), and the Multi-Microprocessor Flight Control System (1981) are practical computer systems that perform critical mission functions and have been specifically designed to ensure mission success [4]. Redundancy of selected computer components within these designs plays an important role in reducing the risk associated with relying upon any single component to operate flawlessly. In one of the most highly visible applications of fault-tolerant computing, the Space Shuttle makes use of redundancy at the level of general-purpose computers to ensure that flight-critical operations such as ascent, reentry, and landing are performed in spite of the failure of any one computer.

Distributed computer systems, as an automatic consequence of their architecture, can be configured for concurrent operation in addition to offering resilience against hardware failure [6]. This attractive dual-property was also observed by the designers of the Space Shuttle computer in their vision of on-board systems for advanced Space Shuttles and Space Stations [5].

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

The focus of this paper is on distributed voting, a well-known fault-tolerance technique in which multiple voters independently compute their results and vote to determine a majority; the majority result is then *committed* (sent to the user). Much work has been done in this area, as described in the next section on past work. Our contribution is a radical approach, in which the normal order of events is reversed – any arbitrary voter can initiate a committal, but the result is buffered long enough for the other voters to check the result and vote on whether they should recommit a new result if the first was incorrect. This process is repeated until a majority of voters agree, and the final result is sent to the user. The security implications of this simple change are profound. The next section reviews the basic concepts of distributed voting, and presents several algorithms that have been proposed in the past. Then our new proposal, the timed-buffer distributed voting algorithm (TB-DVA) is described and analyzed in sections three and four. Section five presents some preliminary work on another algorithm which operates under looser restrictions than the TB-DVA, and hence can be even more generally applied.

2. Past Work

Replication and majority voting are the conventional methods for achieving fault tolerance in distributed systems. *Decentralized voting*, in which the replicated voters independently determine the majority rather than relying on a central server to tally the results, has become the strategy of choice, and has had a number of incarnations [7, 8, 9]. Most of these systems have used the *2-phase commit protocol* in order to implement the voting scheme. In this protocol, the replicated voters first exchange their votes and independently determine the majority result. Once a final result has been calculated, one of the voters is arbitrarily chosen to commit that result (i.e. to pass the result on to the user). This method is widely advocated in designing fault-tolerant open distributed systems [10]. The problem with this type of protocol lies in the committal phase. If the voter chosen to commit the result fails right before or during the committal, the user will receive a bad result. The probability of this happening is slight, and usually falls within acceptable risk parameters. However, if security as well as fault-tolerance is to be taken into account, then the problem is greatly exacerbated. If a hostile attacker has taken control of the committing voter, then the attacker can control what results the user sees, regardless of the other voters' results.

There have been several protocols proposed that attempt to overcome this problem. For example, the algorithm presented in [11] works as follows (in a very simplified presentation):

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

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

Another protocol that attempts to alleviate this problem is described in [12]. It makes use of a *(k,n)-threshold signature* *scheme*. Informally, this describes a scheme wherein a public key is generated, along with n shares of the corresponding private key, each of which can be used to produce a partial result on a signed message m. Any k of these partial results can then be used to reconstruct the whole of m. In this particular protocol, n is the number of voters, and k is set as one more than the number of tolerated faults. Each voter signs its result with its particular share of the private key and broadcasts it to the other voters. The voter then sorts through the broadcast messages for k partial results which agree with its own result and can be combined into the whole message m, where m would be the signed final result. The voter then sends m to the client, which accepts the first such valid m sent. Again, this protocol is not subject to the error inherent in the 2-phase commit protocol, and it is also not computationally expensive for the client. However, it achieves this by shifting the computational burden to the voters. As a result, this system also does not scale very well.

Other protocols have also been proposed, each with its own advantages and disadvantages [13, 14, 15, 16]. However, all of the above schemes for securing the distributed voting process make the common assumption which underlies the idea of state-machine replication – two different voters, starting in the same state and following the same instructions, will inevitably arrive at the same result. While there are many cases when this assumption holds, there are also times when it does not. This is true in the case of so-called *inexact voting* [17].

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

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

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

3. The Algorithm

Assumptions

The proposed algorithm has two sets of participants. One is the set of voters, which can be arbitrarily large but must have at least three elements. These voters are completely independent; the only exchange of information that takes place between them is communicating the voters' individual results. The other set contains the user and an interface module. The interface module buffers the user from the voters (see Figure 1). The interface module consists, in its abstract form, of a simple memory buffer and timer. A task is sent from the user, through the interface module, to the voters. At the termination of the algorithm, the interface module passes the final result back to the user.

The environment for the algorithm is a network with an atomic broadcast capability and bounded message delay (e.g., a local area network). It is assumed that a fair-use policy is enforced, so that no host can indefinitely appropriate the broadcast medium [18]. It is also assumed that no voter will commit an answer until all voters are ready – this can be easily enforced by setting an application dependent threshold beyond which all functional voters should have their results ready; any commits attempted before this threshold is reached are considered automatically invalid. Each voter can commit only once – this is enforced at the interface module, which ignores commits from a voter which has previously committed. The most important assumption made is that a majority of the participating voters are fault-free and follow the protocol faithfully. No assumptions are made about the remaining voters – they can refuse to participate, send arbitrary messages, commit incorrect results, etc.; they are not bound in any way.

Description

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

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

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

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

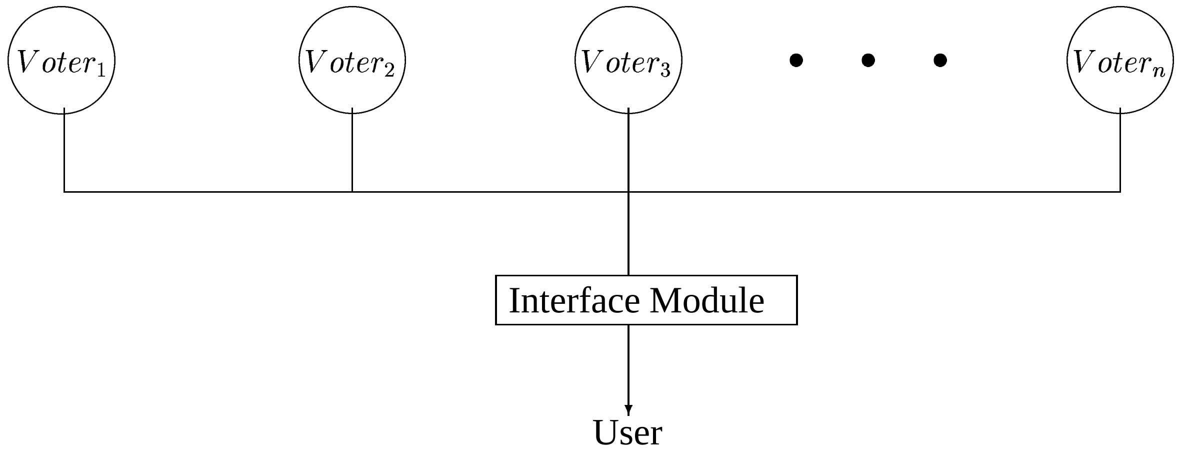


Figure 1: System Architecture

4. Once all voters have had a chance to compare their votes with the committed value (this interval would be determined by a timer), the voter analyzes all the dissenting votes to determine if a majority dissenting vote exists.
5. If no majority exists, then the voter does nothing.
6. If a new majority exists (or if another, perhaps faulty, voter commits a new result), then the voter returns to step 1.

The interface module will follow these steps:

1. Once a commit is received, the result is stored in the buffer and the timer is started. The timer is set to allow time for all the voters to check the committed value and dissent if necessary.
2. If a new commit is received before the timer runs out, the new result is written over the old one in the buffer, and the timer is restarted.
3. If no commit occurs before the timer runs out, then the interface module sends the result in its buffer to the user, and the algorithm is terminated.

4. Discussion

Authentication

For the correct execution of the voting algorithm it is necessary that the commits sent to the timed-buffer module from the various voters be authenticated. Any known sophisticated authentication techniques can be used to enforce secure communication, but it should be done without increasing the complexity of the buffer module. For illustrative purposes we describe a simple authentication technique that doesn't employ standard cryptographic methods such as public key encryption [19]. The technique described here is called SKEY authentication [19], which is simple to implement but is capable of strong authentication with minimal communication between the voters and the interface module. This approach allows the implementation of our secure and fault tolerant voting on the existing platforms without any modifications to the underlying protocols.

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

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

The Interface Module

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

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

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

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

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

Correctness

The algorithm described here has been formally specified in Lamport's Temporal Logic of Action [20], and verified to be correct. The proof of this result, however, is beyond the scope of this paper.

Performance

Besides the security and fault-tolerance attributes of the algorithm, another important characteristic is its performance. A detailed analysis was performed in [21], and showed that this algorithm had definite performance advantages. To summarize the conclusions of that paper, it was determined that this algorithm had an average $O(1)$ performance in relation to the number of voters used – i.e., the algorithm scales extremely well to systems with large numbers of voters. This result is especially important given that the security and fault-tolerance (as opposed to performance) of a system using this algorithm rises linearly with the number of voters in the system.

Intrusion Tolerance

Another benefit of this algorithm that we have yet to fully explore is its applicability to the problem of intrusion tolerance. Any voter that commits an incorrect value can be partitioned from the network and flagged for review by a higher authority (either automated or human) as a possible security breach. Assuming that all voters are denied access to covert channels (a strong assumption), we can also have each voter monitor all other voters, and in a similar manner flag any voter that is releasing confidential information. We will be further exploring these possibilities and what benefit they can bring to the security of the system as a whole.

5. Wide-Area Networks

This section discusses some preliminary work on secure inexact voting in a wide-area network, where the assumptions of atomic broadcast and bounded message delay are not practical. We take advantage of Lamport's results described in [22], where he concludes that Byzantine fault-tolerance can be much simplified through the use of digital signatures. Again, the unique aspect of this algorithm, just as for the previous, is the way it juxtaposes the requirements for security, fault-tolerance, and performance in inexact voting.

Motivation

Distributing data and computation over a wide area network is becoming a standard practice. Critical databases have already been replicated and dispersed to various geographical sites to increase their longevity [23, 24]. Redundant computations are also distributed in order to combat localized network failures and attacks, increasing both security and fault-tolerance. As a consequence, redundant computations on replicated data at remote locations must somehow coordinate their results in order to present a majority result to the user. One example of this requirement is gathering data from distributed sensors with overlapping areas of coverage. Determining a majority result from these sensors produces the lowest probability of error for the widest range of observation probabilities [25]. Data need not be identical. It may even be made different deliberately: data diversity [26] is a software fault tolerance strategy where a related set of points in a program's data space are obtained, executed upon using the same software, and then a decision algorithm (i.e., voter) determines the resulting output.

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

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

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

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

Decentralized voting also allows the necessary computation for determining the majority to be distributed and calculated in parallel among the voters. Insufficient computing capacity of the coordinator can restrict the usefulness of centralized voting. Research has been done in the area of software agents that perform centralized voting [27], but no consideration has been given to agents that may not be able to compute the majority, but only apply it. Such "bounded rational agents" have limited decision capabilities due to restrictions placed upon them regarding the computational resources they can consume [28]. This may be a problem when the task of comparing two votes involves complex calculations, such as when the votes may be somewhat different, yet still be in agreement. Determining the majority of correct-yet-different results calls for "inexact voting" that, being potentially far more complex than a mere bit-wise comparison of results [29], can readily exceed an agent's limited decision making power. Requiring the coordinator to correctly decide among results that can differ but still be correct is understandable when one considers, e.g., the tolerances of sensor readings. Being unable to compute a majority result, an agent that obtains the result from elsewhere could nonetheless use it to, for example, manipulate an actuator through a microcontroller.

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

Issues concerning network congestion, the inability to designate alternate trustworthy coordinators, link failures, the potentially complex computation for determining a majority, and security all motivate the use of distributed voting in a wide area network.

Assumptions

While the underlying wide-area network itself may be unreliable, we assume that this algorithm operates on top of a reliable transport protocol, guaranteeing eventual delivery of messages (although the messages are not necessarily delivered in the order they were sent). On top of this layer is another layer which guarantees eventual delivery of *valid* messages - messages which have been digitally signed and correctly verified as described in the next paragraph. Messages which cannot be verified are discarded. We assume the presence of a public-key infrastructure [19], in which each voter has a private key and each voter knows (or can securely obtain) the public key of every other voter. Each voter knows *a priori* who the other voters are. We further assume that a majority of the voters are fault-free and will correctly follow the protocol (i.e., they are *trustworthy*). As before, no assumptions are made about the remaining voters. There is no interface module in this system – just the voters and the client. The ultimate goal of the algorithm is to have each trustworthy voter agree with every other trustworthy voter on one final result, and to have proof that its result is that which was agreed on.

Two different functions are employed in the algorithm: *one-way hashes* and *digital signatures*. A one-way hash is a function that maps an argument to a unique fixed-width value in such a way that it is impossible to recover the original argument from that value. A digital signature can be accomplished in several ways; one mechanism is encrypting a message (or the hash of a message) with a private key. The signature can be verified by decrypting the signature with the corresponding public key. This provides a secure method of authentication. All signatures include a timestamp to guard against replay attacks.

Description

Each (correct) voter will follow the steps below:

1. Compute a result.
2. Compute the hash of the result and save that value.
3. Sign the result and send it to all the other voters.
4. For all the signed results received from the other voters:
   (a) Make sure that this result isn't a repeat (i.e., there is only one result per voter).
   (b) Verify the signature to make sure it is a valid result.
   (c) If the result agrees with this voter's result (using inexact comparison if necessary), then hash the other voter's result, sign the hash, and send it back to the other voter (this signed hash is called an *endorsement*).
5. For all endorsements received from the other voters:
   (a) Make sure the endorsement isn't a repeat (i.e., only one endorsement per voter).
   (b) Verify the signature and compare the hash value to the value saved in step 2 in order to make sure it is a valid endorsement.
6. Once a majority of endorsements has been received, the algorithm is terminated.

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

Discussion

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

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

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

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

Table 1: Comparison Chart for Centralized and Distributed Voting.

times. This would necessitate that the vote from each voter be sent to a majority of the other voters, greatly expanding the number of messages necessary. A side-benefit of using the hash is that the intermediary nodes cannot determine the value of the various votes they are relaying, since there is no way to de-hash a one-way hash (hence the name). Of course, an untrustworthy voter can relay its own result to anyone it wishes, so this does not provide absolute confidentiality.

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

Performance

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

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

6. Conclusion

Faults present risk to the success of an aerospace mission, so they will continually be a concern of the fault tolerance community. We have taken up the issue of security in conjunction with fault tolerance. This motivated us to devise new approaches to distributed voting. Within a LAN (and some cases a WAN) we replaced the ubiquitous 2-phase commit protocol with one that is light-weight and improves both performance and security without losing *any* of the traditional fault coverage. Accompanying this algorithm is one that we proposed for resolving correct-but-possibly-not-identical votes within a WAN. Both of these algorithms are used to uniquely enhance the integrity of distributed information systems – protecting them from faults and hostile attacks. Table 1 contrasts the advantages and disadvantages of centralized and distributed voting. The contribution of the techniques described in this paper is to remove some of the disadvantages of distributed voting evident in the third quadrant. Applying these algorithms to those distributed systems used for aerospace missions can significantly contribute to the likelihood that the mission will succeed.

References

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

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

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

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

[5] Spector, A., and Gifford, D., "Case Study: The Space Shuttle Primary Computer System," *Communications of the ACM*, Vol. 27, No. 9, September 1984.

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

[7]Harper, R. E., Lala, J. H., and Deyst, J. J., "Fault Tolerant Parallel Processor Architecture Overview," *Proceedings of*

the 18th Fault-Tolerant Computing Symposium, June, 1988, pp. 252-257.

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

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

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

[11] Castro, M., Liskov, B. “Practical Byzantine Fault Tolerance,” it Proceedings of the Third Symposium on Operating System Design and Implementation, Feb 1999.

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

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

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

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

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

[17] Johnson, Barry W., *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley, 1989.

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

[19] Schneier, Bruce *Applied Cryptography, Second Edition*, John Wiley & Sons, 1996.

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

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

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

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

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

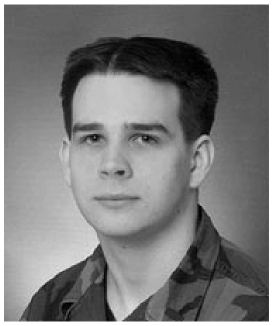
[25] Varshney, P. K., *Distributed Detection and Data Fusion*, Springer, 1997.

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

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

[28] Hendler, J., “Unmasking Intelligent Agents,” *IEEE Intelligent Systems*, IEEE Computer Society Press, March/April 1999.

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



Ben Hardekopf is a lieutenant in the United States Air Force, stationed at the Air Force Research Laboratory. He received the BSE in Electrical Engineering from Duke University and is currently working towards the MS in Computer Science from the State University of New York at Utica/Rome.

Dr. Kevin A. Kwiat has been with the U.S. Air Force Research Laboratory for over 17 years. He is an adjunct professor of Computer Science at the State University of New York at Utica/Rome, and an adjunct professor of Mathematics at Utica College of Syracuse University. He received the BS in Computer Science, the BA in Mathematics, the MS in Computer Engineering, and the Ph.D. in Computer Engineering, all from Syracuse University. He holds 1 patent.

Shambhu Upadhyaya received his Ph.D. in Electrical and Computer Engineering from the University of Newcastle, Australia in 1987. He is currently an Associate Professor of Computer Science and Engineering at the State University of New York at Buffalo. His research interests are fault-tolerant computing, distributed systems, and security.