A Decentralized Voting Algorithm for Increasing Dependability 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

Replication and majority voting are well-known and widely-used methods for achieving fault-tolerance in distributed systems. An open area of investigation is coordinating the voting in a secure manner, so as to withstand malicious attacks. The Timed-Buffer Distributed Voting Algorithm (TB-DVA), a secure distributed voting protocol, is introduced for this purpose. It is contrasted with several other distributed voting schemes in order to show its unique contribution for both fault-tolerance and security, which are essential ingredients for system dependability.

1 Introduction

Dependability can be defined as the trustworthiness of a system, so that a high level of confidence can be placed in the service it performs. Dependability encompasses (but is not limited to) the concepts of *availability* (the service is ready whenever it is needed), *continuity* (the service is never prematurely terminated), and *security* (unauthorized access and tampering of the system is not permitted) [1].

One aspect of dependability is fault-tolerance, which is mainly concerned with the first two concepts described above – availability and continuity. Distributed voting is a common method for achieving fault-tolerance, consisting of a set of distributed processors all working on the same task, then voting on the independent results to pick one as the correct answer. This technique has been in use in the Space Shuttle's Primary Computer System since the 1970s [2]. Obviously, all the voters involved must have a similar amount of complexity in order to work on a given task. It is a truism that the more complex the system, the more likely it is to fail – in other words, the more complex the voter, the more unreliable it is. If one voter is chosen to coordinate the voting and resolve the votes into a final answer, the dependability of the system as a whole revolves around the dependability of that one voter. Current distributed voting schemes assume that there is a sufficiently low probability of failure during that last stage that this doesn't become a problem.

This voting methodology has had several embodiments ([3, 4, 5, 6]) in the development of fault-tolerant computing. More recently, distributed voting has been used for fault diagnosis in linear processor arrays ([7]) where, in the absence of a centralized voter, the array elements share error flags stemming from output comparisons performed between connected elements. Nevertheless, in fault tolerance terms, current distributed voting schemes continue to subscribe to the idea called the *two-phase commit protocol* [8]: they undertake a voting phase, in which all participants prepare to commit, followed by a second phase, in which they execute the commit. A distinguished process coordinates the commit. We discovered a way to improve upon distributed voting by taking the somewhat non-intuitive approach of actually reversing the two-phase commit protocol: *we initiate the commit first, then we vote.* This simple idea turns out to have profound implications for security.

Security is another important aspect of dependability. Even if in normal operation a system is perfectly reliable, the introduction of malicious logic can have a wide range of undesired effects, from denying availability of the system (i.e., the denial of service threat) to causing the system to give users incorrect results (i.e., the integrity threat). Current voting schemes for tolerating faults (e.g. the two-phase commit protocol) do not involve the question of security, assuming that it can be grafted onto the system at some other level. For example, fault tolerance is designed into the security kernel of operating systems so that unavoidable faults do not result in security policy compromise [9]. It then becomes conceivable to build trusted services upon this underlying layer that will remain secure even in the presence of faults. It should be noted that evidence ([10]) strongly

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

suggests the need for security measures at the lower levels of software abstraction to form the foundation for secure computing at the higher levels. This motivates us to consider distributed voting at the processor level without assuming an over-arching computing environment to provide security. Rather than treating fault-tolerance and security as two entirely separate issues, our new algorithm provides both of them at the same time, meaning additional security over and above what is provided elsewhere in the system.

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

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.

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 [11]. Most of these systems have used the *2-phase commit protocol* [8] 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 [15]. 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 [16] 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 [17]. 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 [18, 19, 20, 21]. 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* [11].

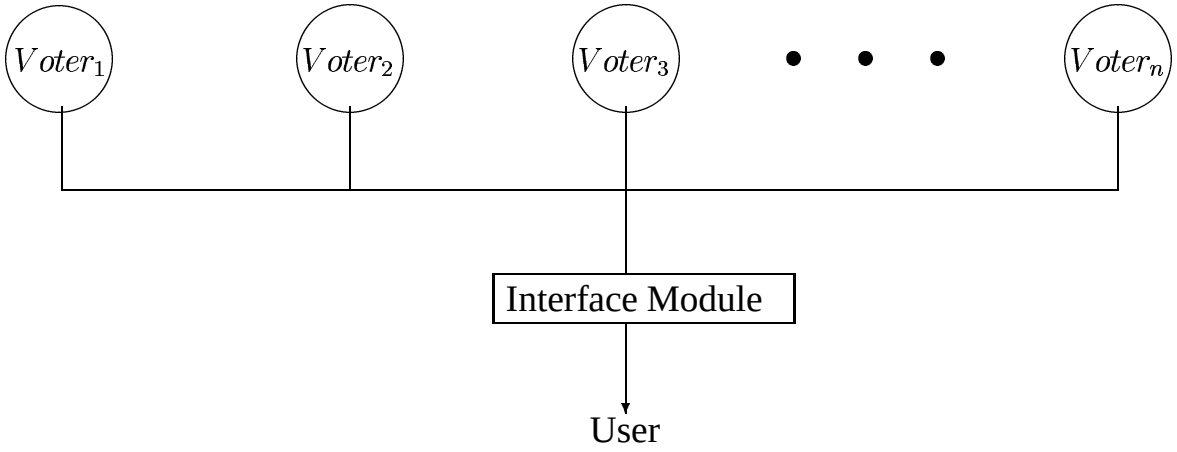

Figure 1: System Architecture

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

3.1 Assumptions

The proposed algorithm has two sets of participants. One is the set of voters, which can be arbitrarily large but must have at least three elements. These voters are completely independent; the only exchange of information that takes place between them is 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 [22]. 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. Increasing the effort to breach security can enforce this. To successfully overtake a majority of voters, each having diverse intrusion detection packages and user interfaces, requires attackers to possess greater experience and ability, and costs them more in terms of both financial cost and elapsed time [23]. 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.

3.2 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.
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

Replication and majority voting are the conventional methods for achieving fault tolerance in distributed systems. Distributed voting has become the strategy of choice, and has had a number of incarnations [12, 13, 14]. Heavy reliance has been placed on the 2-phase commit protocol [8], in which the voters first exchange votes and independently determine the majority result, and then one arbitrary voter within the majority commits this value to the user. This method is widely advocated in designing fault-tolerant open distributed systems [15]. 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.

The purpose of the TB-DVA algorithm is to provide the user with a correct answer despite a known number of the participating voters being faulty, or even actively hostile. It also allows for inexact voting (voting in which two votes may be effectively equal, even if not necessarily bit-wise identical). This is particularly important for applications which interface with the real world, such as distributed sensor arrays. TB-DVA is a radical approach to distributed voting because it reverses the 2-phase commit protocol: a committal phase (to a timed buffer) is followed by a voting phase. This conceptually simple change greatly enhances security by forcing an attacker to compromise a majority of the voters in order to corrupt the system; whereas in the 2-phase commit protocol only one voter must be compromised to achieve the same end.

For the correct execution of the voting algorithm it is necessary that the commits sent to the interface module from the various voters be authenticated, as well as the messages between the voters themselves. Any known sophisticated authentication techniques can be used to enforce secure communication (such as those in [24]), but it should be done without increasing the complexity of the interface module. For authentication between the voters, the particular method used isn't as important since there is plenty of computational power available. The interface module, however, should be kept as simple as possible.

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

Depending upon the level of voting, the design of the interface module may vary. Voting may proceed at either hardware or software levels. It essentially depends on the volume of data, complexity of computation, and the approximation and context

dependency of the voting algorithms. If low-level, high-frequency voting is to be done, a hardware implementation might be preferred; if high-level voting with low-frequency is desired, a software implementation of the interface module may be suitable. This is because the voting is generally much more complex at higher levels of abstraction. A software implementation is fairly simple, although considering the real-time nature of the timer, a real-time operating system would provide better performance than a non-real-time operating system; the non-real-time operating system would require multiple context switches in order to process each event, providing a much courser time granularity than a real-time operating system.

4.1 Correctness

The algorithm described here has been formally specified in Lamport's Temporal Logic of Action [25], and verified to be correct. The proof of this result, however, is beyond the scope of this paper. As expected, the process of formalizing the algorithm led to the discovery of several ambiguities or outright errors in the initial formulation of the algorithm, which were subsequently corrected. As an example, initially the algorithm provided for the case where a voter would commit a value to the user before the other voters had a chance to calculate a result – if the other voters didn't have a result, then they didn't know whether they should dissent or not. We overcame this difficulty by providing a special 'not ready' vote, which a voter would use for exactly this cicumstance. A majority of 'not ready' votes would then force the timer in the interface module to stop, giving the voters time to finish their computations. However, upon closer examination this wasn't a very effective method. It is possible for each voter to finish its calculations before any of the remaining voters do (e.g., the first voter finishes before the others and commits; the other voters vote 'not ready' and stop the timer; the second voter finishes before the rest and commits; the other voters vote 'not ready' and stop the timer; *et cetera*). In such a situation, a faulty voter could commit an incorrect result, and there may not be a majority of trustworthy voters left that have not committed. We solved this difficulty by getting rid of the 'not ready' votes and establishing a global time threshold, beyond which all functional voters should have calculated a result. Any voter which tries to commit before this time is ignored; therefore all trustworthy voters are guaranteed to have a result ready when a value is committed.

The specification also forced us to detail the exact assumptions we were working under, as specified in the Asusmptions section. We found TLA and TLA+ to be easy to work with, and they gave us excellent results. The experience gained through this process has served to convince us even more of the importance of formal methods in algorithm design and analysis.

4.2 Performance

Besides the security and fault-tolerance attributes of the algorithm, another important characteristic is its performance. A detailed analysis was performed in [26], 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.

5 Conclusion

Society's growing dependence upon critical computations together with the increase in network hostilities motivated us to further investigate the issue of security in conjunction with fault-tolerance. This prompted us to devise new approaches to distributed voting. 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. This algorithm uniquely enhances the dependability of distributed information systems – protecting them from faults and hostile attacks. Applying this algorithm to distributed systems contributes significantly to their dependability.

References

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

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

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

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

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

[6] 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.

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

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

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

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

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

[12] 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.

[13] 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.

[14] 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.

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

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

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

[18] 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.

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

[20] 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.

[21] 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.

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

[23] Brocklehust, S., *et al.*, "On Measurement of Operational Security", *Proceedings of COMPASS '94*, June 1994.

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

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

[26] 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.