Specification and Verification of a Secure Distributed Voting Protocol

Ben Hardekopf, Kevin Kwiat, Shambhu Upadhyaya*
Air Force Research Laboratory
AFRL/IFGA
525 Brooks Rd.
Rome, NY 13441-4505
phone: (315)330-2838
{hardekob, kwiatk}@rl.af.mil; shambhu@cse.buffalo.edu

Keywords: Security, Fault-Tolerance, Distributed Voting, Formal Methods, TLA

Abstract

The Timed-Buffer Distributed Voting Algorithm (TB-DVA), a secure distributed voting protocol, is introduced and described. A formal specification of the algorithm is developed using Lamport's specification language TLA+. Then strategies for proving the correctness of the specification using Lamport's Temporal Logic of Actions (TLA) are discussed.

INTRODUCTION

The Timed-Buffer Distributed Voting Algorithm was developed to fulfill a need for inexact voting [1] in a potentially hostile distributed environment, while maintaining the goals of fault-tolerance, security, and performance. A description of the algorithm, along with a detailed performance analysis and comparison to other distributed voting protocols, has been previously published in [2, 3, 4]. In this paper, we present an a formal specification of the algorithm and prove the correctness and completeness aspects of the protocol.

Lamport's Temporal Logic of Actions (TLA) [5] and TLA+ [6] are the ideal tools for this purpose. TLA (as well as TLA+, which is built on top of TLA) is an easy to use language designed specifically to reason about concurrent and distributed systems. Using these tools and the specification, the proofs become fairly simple and direct.

The next three sections discuss the TB-DVA algorithm and its assumptions and properties. Next, both TLA and TLA+ are briefly summarized. The TB-DVA specification is then presented and explained. Last, the strategies for proving both partial correctness and termination of the algorithm are presented.

ASSUMPTIONS

The TB-DVA algorithm has two sets of participants. One is the set of voters, which can be arbitrarily large but must have at least three elements. These voters are completely independent; the only exchange of information that takes place between them is the communication of the voters' individual results. The other set contains the user and an interface module. The interface module buffers the user from the voters (see Figure 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 (as described in [7]). It is also assumed that no voter will commit an answer until all voters are ready – this can be easily enforced by setting an application dependent threshold beyond which all functional voters should have their results ready; any commits attempted before this threshold is reached are considered automatically invalid. Each voter can commit only once – this is enforced at the interface module, which ignores commits from a voter which has previously committed. The most important assumption made is that a majority of the participating voters are fault-free and follow the protocol faithfully (these are called *trustworthy* voters). Increasing the effort required for an attacker to breach security can enforce this assumption. To successfully

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

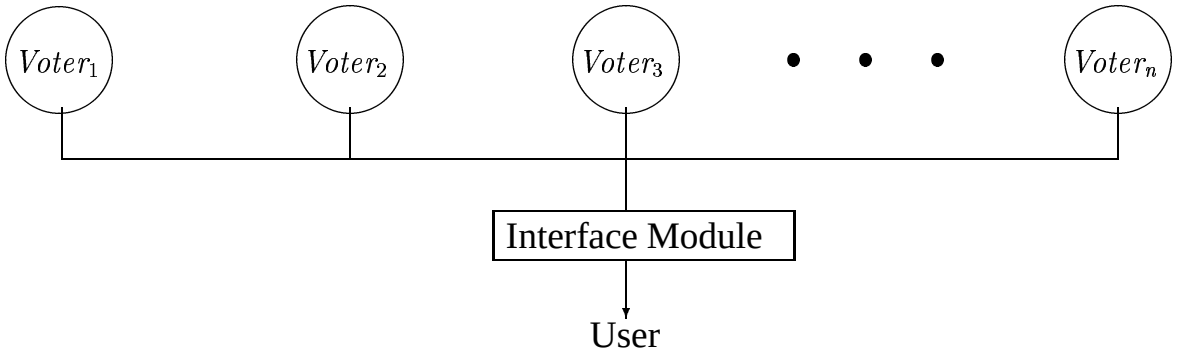


Figure 1: System Architecture

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 [8]. No assumptions are made about the remaining voters (the *untrustworthy* voters) – they can refuse to participate, send arbitrary messages, commit incorrect results, etc.; they are not bound in any way.

DESCRIPTION

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

1. If no other voter has committed an answer to the interface module yet, the voter does so with its own vote; it then skips the remaining steps.
2. In the case that another voter has committed, the voter compares the committed value from the other voter with its own vote.
3. If the results agree, the voter does nothing; otherwise it broadcasts its dissenting vote to all the other voters.
4. Once all voters have had a chance to compare their votes with the committed value (this interval would be determined by a timer), the voter analyzes all the dissenting votes to determine a dissenting vote exists which is in the majority.
5. If no majority exists, then the voter does nothing.
6. If a new majority exists (or if another, perhaps untrustworthy, voter commits a new result), then the voter returns to step 1.

The interface module will follow these steps:

1. Once a commit is received, the result is stored in the buffer and the timer is started. The timer is set to allow time for all the voters to check the committed value, dissent, and recommit if necessary.
2. If a new commit is received before the timer runs out, the new result is written over the old result 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.

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 [9, 10, 11]. Heavy reliance has been placed on the 2-phase commit protocol [12], 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 [13]. 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. It does all this in a high-performance manner, as discussed in [3]. To summarize the results presented in that paper, the average asymptotic performance of TB-DVA is $O(1)$ with respect to the number of voters employed, while the security and fault-tolerance of the system rise linearly with the number of voters.

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 [14]), 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.

CORRECTNESS

In order to prove the correctness of this algorithm, we turn to Lamport's Temporal Logic of Action (TLA) [5]. This temporal logic is ideally suited to proving properties of distributed or concurrent processes.

The first step is to utilize TLA+ (a specification language based on TLA; [6]) to write a precise, formal specification for the algorithm which faithfully followed the steps outlined above. This specification can then be used to prove the partial correctness of TB-DVA (i.e., **if** the algorithm terminates, **then** it produces the correct result). Secondly, the specification can be used to prove termination (i.e., the algorithm always terminates). From these two properties, we can conclude that the algorithm is correct (it terminates, and when it terminates it produces the correct result).

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

Temporal Logic of Actions

TLA

TLA is a temporal logic specifically designed to reason about concurrent and distributed algorithms. In this section we will discuss the subset of TLA relevant to the specification and proofs below. More complete information can be found in [5, 6, 15].

All TLA formulas can be expressed using the familiar operators of predicate logic (e.g., $\wedge$, $\vee$, $\neg$), in addition to a few new operators such as $'$ (prime) and $\Box$ (read as *always*), which will be discussed below. TLA is a *state-based* logic – we use it to describe states and state-transitions, where a state is simply a mapping from the set of variable names to the collection of possible values. Every state is universal, in the sense that it assigns some value to every possible variable name (although usually, we are only interested in a small subset of the variables and ignore the rest). TLA is typeless, meaning that any variable can assume any value. If type-invariance is desired, that property must be enforced by the formulas used (i.e., the

formulas must make certain not to assign more than one type of value to any one variable).

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

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

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

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

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

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

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

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

TLA+

TLA+ is a language built atop the foundation of TLA that enables the precise definition of algorithmic specifications. In this section we will introduce some of the TLA+ operators that are used in the specification of the TB-DVA algorithm. More about TLA+ and the TLA+ operators can be found in [6].

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

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

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

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

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

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

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

DOMAIN DOMAIN f is the domain of function f.

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

$$\begin{aligned} &\wedge A \\ &\wedge \vee B \\ &\quad \vee C \end{aligned}$$

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

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

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

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

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

The Specification

Explanation of the Specification

This section contains the TLA+ specification for the TB-DVA algorithm. More information about TLA+ can be found in [6]. Here we provide a brief explanation of this particular specification. The actual specification is given in Figure 2.

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

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

The variables are: $buffer$, $user$, cv, $votes$, $rcvd\_commit$, $dissented$, $analyzed$, rdy_commit, four distinct T variables, and now. $buffer$ represents the buffer in the interface module, which records each committal. $user$ represents the end user that receives the final result. cv is a tuple that records the voters that have already committed. $votes$ is a two-dimensional array, each row and column of which corresponds to a voter – $votes[i][j]$ represents the vote which voter i received from voter j. $rcvd\_commit$, $dissented$, $analyzed$, and rdy_commit are all boolean arrays, each element of which corresponds to a voter. For $rcvd\_commit$, an element is TRUE if that voter has received a commit from another voter, FALSE if it has not; for $dissented$, an element is TRUE if that voter has had a chance to dissent to a committal, FALSE otherwise; for $analyzed$, an element is TRUE if the corresponding voter has analyzed the results of the various dissents from all the voters, FALSE otherwise; and finally for rdy_commit, an element is TRUE if the corresponding voter is ready to commit a value, otherwise it is FALSE. To be clear, these arrays are not necessarily part of the protocol itself, but simply keep state information about the system. For instance, the specification has all the voters, including the untrustworthy ones, record in the $dissent$ array whether they have had a chance to dissent. In a real system, this would be obvious to all the voters simply by recording whether any particular voter broadcasted a dissent over the network. The variables here are abstractions, and do not correspond directly to a real implementation. The T and now variables have to do with the real-time aspect of the algorithm, and will be explained later.

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

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

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

The initial conditions are an important part of the algorithm – if these conditions are not met, the behavior of the system would be unpredictable, and likely incorrect. $Init$ defines these conditions for TB-DVA: $buffer$ and $user$ are equal to '?' (i.e., they are not recognized as possible answers; see assumptions above); cv is empty (no voters have committed yet); for all trustworthy voters i, $votes[i][i]$ equals $RightAnswer$ and all other elements of $votes[i]$ equal '?' (all trustworthy voters have computed the correct answer and have not received any votes from other voters yet); all elements of $dissented$ and $analyzed$ equal FALSE (no voter

has either dissented to a committal or analyzed any votes); and for all trustworthy voters i, $rdy\_commit[i]$ equals TRUE and $rcvd\_commit[i]$ equals FALSE (all trustworthy voters are ready to commit an answer, and have not yet received a committal from another voter). Note that the initial conditions make no statements about what the untrustworthy voters have recorded in $votes$, rdy_commit, or $rcvd\_commit$. Being untrustworthy, we can't say anything about their state. $dissented$ and $analyzed$ are special cases – these are global variables, not local to each voter. Also note that the initial conditions require all trustworthy voters to have already calculated a result before the algorithm begins, as described in the assumptions in section 3.1.

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

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

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

Finally, $Terminate$ is the last action that can be taken – it is the action that terminates the algorithm. It makes the same local definition of CV that was made in $Commit(v)$. The enabling conditions are: $user$ is not an element of $Answers$ (the algorithm has not terminated yet); and for all the voters i which have not committed yet, $analyzed[i]$ is TRUE and $rdy\_commit[i]$ is FALSE (the voters have analyzed the votes, and each has decided not to commit another value). When this condition is met, $user$ is set to the value of $buffer$ (which contains the value which was last committed). This terminates the algorithm, since all actions require that $user$ is not an element of $Answers$, and therefore no more actions can be taken.

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

The final part of the specification defines a real-time version of the specification. The initial specification, Φ, sets a safety condition on the algorithm – it will only take an allowable step. However, that statement is satisfied by a series of steps which never change the values of the variables (i.e., time stands still). In order to force the behavior of the specification to take some action, we have to enforce some timing constraints. This is what the Δ constants and the T and now variables are for. The Φ^t specification definition basically states that each action ($Commit(v)$, $Dissent(v)$, $Analyze(v)$, and $Terminate$), must take place within Δ time units of when they are first enabled. In other words, once it is possible to take a particular action, that action must be taken within some set time limit. For more information about real-time TLA, consult [15].

module *TB-DVA*

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

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

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

$Card(S) \triangleq$ **let** $c[R \in \text{SUBSET } S] \triangleq$ **if** $R = \{\}$ **then** 0
else $1 + c[R \setminus \{\text{CHOOSE } r \in R\}]$
in $c[S]$

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

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

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

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

module *TB-DVA (cont)*

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

$$
\begin{array}{l}
Next \triangleq \lor \exists\, v \in Voters : \lor\ Commit(v) \\
\qquad\qquad\qquad\qquad\qquad\ \lor\ Dissent(v) \\
\qquad\qquad\qquad\qquad\qquad\ \lor\ Analyze(v) \\
\qquad\quad \lor\ Terminate \\
\\
\Phi \triangleq Init \land \Box[Next]_{var}
\end{array}
$$

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

Figure 2: TLA+ Specification of TB-DVA

Proof of Partial Correctness

The proof of partial-correctness is long (approximately 13 pages), but conceptually it is not very difficult. An algorithm is partially-correct if the following statement is true: **if** the algorithm terminates, **then** it produces the correct result. In terms of the specification, we can describe the same statement as:

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

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

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

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

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

$$Init \Rightarrow Inv \quad (1)$$

$$Inv \wedge [Next]_{var} \Rightarrow Inv' \quad (2)$$

$$Inv \Rightarrow (user \in Answers \Rightarrow user = RightAnswer) \quad (3)$$

The actual invariant used in this proof consists of eleven conjuncts, chosen specifically in order to prove the statements above. From these statements, we can then conclude:

$\Phi \Rightarrow$ {By definition of Φ}
$Init \wedge \Box[Next]_{var}$
$\Rightarrow$ {By (1) above}
$Inv \wedge \Box[Next]_{var}$
$\Rightarrow$ {By (2) above and rule INV1}
$\Box Inv$
$\Rightarrow$ {by (3) above}
$\Box(user \in Answers \Rightarrow user = RightAnswer)$

Proof of Termination

The proof of termination is much shorter and simpler than the proof of partial-correctness. It is proved using a simple loop termination argument. To show this, Figure 3 provides a graphical view of the flow of control of the TB-DVA algorithm. It can be clearly seen that essentially the algorithm loops through the $Commit$, $Dissent$, and $Analyze$ functions until the criteria are met for falling through to $Terminate$. The basic argument is that each time the $Commit$ action is taken, the number of voters left that are able to commit is decremented by one (since each voter can commit at most once, as enforced by the interface module). Eventually, there will be no voters left that are both able and willing to commit. However, that circumstance satisfies the exit criteria for the loop, and the algorithm terminates. The actual proof is consistent with scenarios in which all the voters attempt to commit (hence exhausting the set of available voters), as well as those in which the remaining voters decide not to commit, thereby leaving some voters able to commit, but not willing to do so. The algorithm terminates in either case.

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

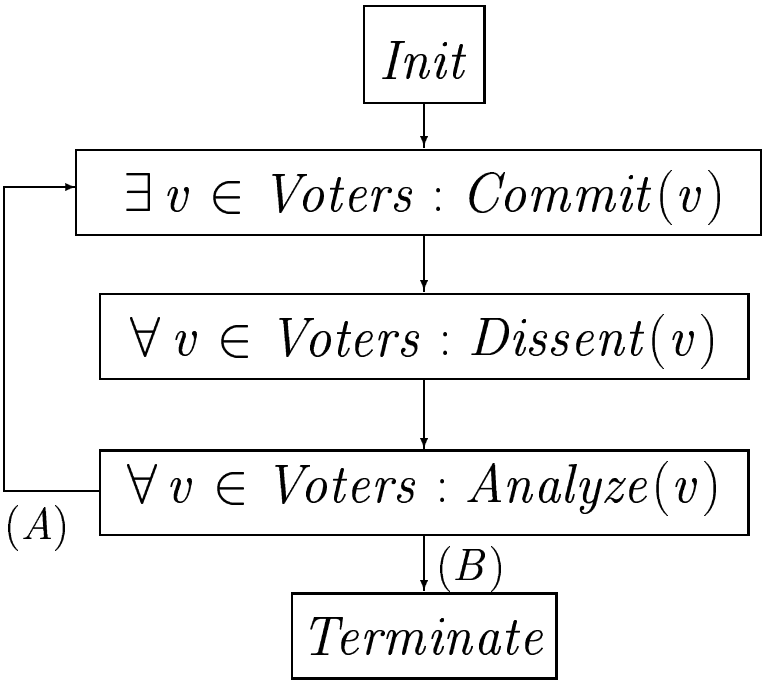
1. Initially, $t > 0$.
2. $t = 0 \Rightarrow$ the algorithm terminates (i.e., formula B in Figure 3 is satisfied).
3. t is decremented each time through the loop.

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

CONCLUSION

The main purpose of this paper has been to offer a rigorous proof for the TB-DVA voting protocol, which incorporates security as well as fault-tolerance. The voting protocol has applications in critical systems that require assurance against both accidental and intentional faults.

Using TLA and TLA+, we have formally specified and verified the TB-DVA secure distributed voting algorithm. As expected, the process of formalizing the algorithm led to the discovery of 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



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

Figure 3: TB-DVA Flowchart

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 circumstance. A majority of 'not ready' votes would then force the timer in the interface module to stop, giving the voters time to finish their computations. Upon closer examination, however, this proved to be an ineffective 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 removing 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.

TB-DVA reverses the 2-phase commit protocol by initiating a commit to a timed buffer and then allowing for a period of dissenstion in a voting phase. Although conceptually simple, this change forces an attacker to overcome a majority of the voters in order to compromise the system. This has important security implications because it greatly increases the cost of a successful attack. To ensure that the TB-DVA protocol does not suffer from any flaws which would negate this benefit, we subjected it to the rigors of formal methods using the Temporal Logic of Actions. The verification of the protocol, in addition to the realistic assumptions upon which it is based, give convincing evidence that it can be effectively employed.

References

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

[2] B. Hardekopf, K. Kwiat "Exploiting the Overlap of Security and Fault-Tolerance," *Proc. of the Acedemia/Industry Working Conference On Research Challenges (AIWORC)*, Apr 2000.

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

[4] B. Hardekopf, K. Kwiat, S. Upadhyaya "Secure and Fault-Tolerant Voting in Distributed Systems," *accepted for publication by the IEEE Aerospace Conference*, March 2001.

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

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

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

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

[9] 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.

[10] 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.

[11] 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.

[12] Gray, J., and Reuter, A., *Transaction Processing: Concepts and Technologies*, Morgan Kauffmann Publishers, 1993.

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

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

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

Biographies

Ben Hardekopf is a lieutenant in the United States Air Force, stationed at the Air Force Research Laboratory. He received a BSE in Electrical Engineering from Duke University and is an MS in Computer Science from the SUNY Institute of Technology 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.