Middleware Support for Voting and Data Fusion

David E. Bakken, Zhiyuan Zhan
School of Electrical Engineering
and Computer Science
Washington State University
Pullman, Washington USA
{bakken,zzhan}@eecs.wsu.edu

Christopher C. Jones, David A. Karr
Distributed Systems and Logistics Dept.
BBN Technologies
Cambridge, Massachusetts USA
{ccjones,dkarr}@bbn.com

Abstract

Middleware is a class of software systems above the operating system which is becoming widely used for programming distributed systems. Voting is a fundamental operation when distributed systems involve replicated components. However, support for voting in middleware is very limited. This paper describes issues involved with supporting voting, and more general data fusion, in middleware. We describe the Voting Virtual Machine (VVM) architecture, which can be embedded in different middleware substrates such as .NET and CORBA. We also describe its companion Voting Definition Language (VDL), which allows for portable description of voting algorithms. The VVM and VDL, together with the external VVM voting manager, provide for voting transparency and adaptive voting. Finally, we describe how the simple “byte-by-byte” value comparison schemes used in other voting middleware as well as in Byzantine fault tolerant multicast systems do not work in the face of the heterogeneity inherent in distributed systems.

1 Introduction

Replication is a technique commonly used to increase the availability of computer-based services. Active replication is a common form of replication [11, 38]. In this scheme, the service request from a client is multicast to all replicas in a consistent manner (atomically and totally ordered). Each replica then processes the request and sends back a reply. One of the replicas’ replies must be chosen for the client. This process of choosing one reply from many is called voting. The reply chosen by the voting process is then used by the client, e.g., it may contain data the client has requested from the service.

A simple example of this is shown in Figure 1. Each reply arrives at the voter, which must choose one and send it on to the client. In addition to this simple scenario, clients themselves may be replicated, as with a nested invocation chain, in which case the requests sent out by the client replicas must be collated into a single request to be sent to each of the server replicas.

Data Fusion [41] is similar to voting, in that it involves producing one output datum from many related input data. It is different from voting in a number of key ways. For example, there are not well-defined replicas to expect inputs from, so naming and the number of inputs to expect are issues. Another difference is that the values are not necessarily expected to be the same or even close, as with actively replicated servers. Data fusion is a key technique used with distributed sensor networks [18], which are becoming increasingly prevalent, and intrusion detection systems [5], which are becoming increasingly necessary.

Middleware is a layer of software above the operating system that offers a common programming abstraction across a distributed system and helps mask the different kinds of heterogeneity inherent in distributed systems [3]. Middleware systems such as CORBA [31] have become very popular in recent years largely due to these benefits. However, current middleware support for voting is extremely limited in both research projects and in comme
cial products and standards, and is virtually nonexistent for the more general techniques of data fusion. The voting supported in middleware generally allows at best the specification of “majority” or “all,” based only on the return value of a method. These schemes implicitly assume that all methods have a return value and that no values from returned parameters (e.g., CORBA `out` or `inout` parameters) will be used. Further, with a single recent exception [30], they are undefined where the clients are replicated, and thus, multiple requests must be collated into one.

Current implementations of voting in middleware are also very inflexible in dealing with floating point values in general as well as inexact voting in particular. Every middleware system of which we are aware performs voting in a “byte-by-byte” fashion on a marshalled network message rather than voting on unmarshalled application-level data. Moreover, little progress has been made toward the ability to change voting algorithms at runtime in response to changing conditions. This would be useful, for example, to provide operating points with different tradeoffs with respect to performance, fault tolerance, precision, and correctness, and switch between them when appropriate. More details on these limitations can be found in [19, 42].

The contributions of this paper are as follows:

- A description of the Voting Virtual Machine (VVM), a middleware component which can be embedded in different middleware substrates and which votes on application-level data.

- A presentation of the Voting Definition Language (VDL), which is used with the VVM and which allows portable and reusable coding of voting algorithms.

- A detailed description of why current byte-by-byte implementations of voting in middleware do not work in the face of the very kinds of heterogeneity middleware is designed to mask. Additionally, we describe why value comparisons in current byzantine-fault-tolerant multicast substrates do not work for the same reasons, and discuss the architectural implications of this.

Together, the VVM and VDL help push the complexity of adding fault tolerance away from the application programmer and towards the infrastructure.

The remainder of this paper is as follows. Section 2 presents the VVM and Section 3 describes VDL. Section 4 provides VDL examples for clock synchronization and a supermajority voting algorithm. Section 5 discusses implementation and other pragmatic issues regarding the VVM. Section 6 discusses related work, and Section 7 concludes this paper and discusses future work.

2 VVM Architecture

2.1 Overview

We have designed and developed a prototype version of the Voting Virtual Machine (VVM), and its companion Voting Definition Language (VDL) policy language [4]. The VVM provides a powerful, well-specified, general, portable, flexible, and adaptable mechanism for voting. It can be used stand-alone to develop and test portable voting algorithms. The VVM allows voting policies to be changed at runtime, and can be integrated with presentation layers such as those for distributed objects (CORBA, SOAP, .NET, DCOM), message-oriented middleware (IBM’s MQSeries or Microsoft’s Message Queue ( MSMQ)), as well as with other kinds of middleware. Indeed, to the best of our knowledge, the VVM is the first system for voting or collation which is embeddable into different middleware substrates, and VDL is the first independent and portable voting language that has been implemented.

A simplified picture of the VVM architecture is shown in Figure 2, as applied to CORBA (though it could equally well apply to any other middleware system). The dashed box represents the module labeled “Voter” in Figure 1. The VVM operates as follows. Network messages (marshalled for transmission, e.g., into the CDR format used in CORBA IIOP or with Sun’s XDR) are received independently from each replica into the `unmarshal` module which converts the linearized messages into an array of parameters. The parameter arrays are sent to the `voter core` module, which selects one set based on the current voting policy, as described in the following section. Finally, the `marshal` module “flattens” the voted parameters into one message, which is sent to the server or client. The voter as shown here can be configured in a gateway [36]; alternatively, it can be embedded in other middleware, e.g., in a CORBA ORB, in which case its `unmarshal` and `marshal` steps will be the ORB’s ones.

The voter core module accepts inputs which inform it of the current voting policy (VDL code fragment) as well as notifying it when a failure occurs. It outputs status information on the voting, which can be used by external managers to help detect performance problems, potential intrusions, or other anomalies. This status information can be passed to a Voting Status Service (not described in this paper) to support features such as intrusion detection.

2.2 Basic Voter Functionality

We now describe the basic voting functionality provided by VVM, after first providing some definitions. A ballot is a request or reply message sent by a single replica. A vote is the process of choosing (or constructing) one message from among all the ballots.
conditions & QOS requirements

voting policy params; weights

Voting
Manager

Voting Status
Service (VSS)

Current conditions & QOS requirements

Alerts to subscribers

Figure 2. VVM Architecture

Figure 3. States in the Voter Core

The voter core shown in Figure 2 processes a single vote through the three states, as depicted in Figure 2.2. At each stage, the VDL code for the current voting policy dictates the action of the voter core at each stage.

The voter begins each vote in the quorum state, where it waits for enough ballots to arrive to begin the actual voting process. After sufficient ballots have arrived, the exclusion state is entered. Here a number of ballots can be excluded from further consideration to enable the application to tolerate value failures, e.g., those caused by a faulty (and undetected) floating point processor or a breach in security. The collation state follows exclusion. Here, one value is chosen from those not excluded.

After the collation state is finished, the voter may also return a confidence value, indicating the level of confidence in the vote. These are discussed further in Section 3.2.

In each of the three states, exceptions may be thrown as directed by the VDL; in which case, an exception is returned to the client that would normally receive the vote. For example, in the quorum state it may be desirable to throw an exception if the quorum is not met after a certain timeout. An exception could be thrown in the exclusion state if too many members are excluded, or in the collation state if the number of ballots which included the voted-upon value was too small.

In the last two states the voter may branch back to a previous state, based on the same status conditions on which an exception may be thrown. For example, in the exclusion state, if too many ballots are excluded, the voter can branch back to the quorum state to wait for more.

2.3 Advanced Voter Functionality

The voter can be specified to perform random operations at any or all of the three states to help thwart an adversary intent on corrupting a vote; this is, of course, specified in VDL. For example, in the quorum state, the voter can be instructed to wait for a random percentage of the maximum ballots (size of the group sending the ballots) to arrive. Further, this number could change with each vote or could be the same for all votes the voter managed. In the exclusion state the voter can exclude a random number or percentage of the ballots, or all but a random number of the ballots; again, this percentage could potentially be reset at different time intervals. Finally, the collation state can randomly choose a value. This is reasonable in many applications where the particularly bad data have already been discarded in the exclusion state.

Another way in which the VVM can resist corruption is by weighted voting, or giving different replicas varying amounts of trust in different states of voting. For example, in the quorum state, instead of waiting for a given number
of ballots, it can wait for some number of points, with each replica assigned a different number of points. When a ballot arrives, that ballot’s arrival counts towards the number of points dictated by its quorum weighting. In the collection state, operations can be performed after the remaining ballots are expanded based on a weighting. For example, suppose there are ballots with values \{2,3,4\} from replicas 1–3, respectively, and the weighting is \{1,2,2\}. After the expansion, the ballots will be \{2,3,3,4,4\}, and this is what a collation operation such as median or random will utilize.

The weights used can be computed based on many factors, including intrusion detection feeds, randomizations, and other factors. They are set by an external manager, not specified in the VDL.

### 2.4 Failure Model

The VVM assumes a crash failure model for the hosts it runs on. This is a reasonable assumption in many situations, because, in most configurations, the VVM will be running in the same machine or even process as the client. Thus, for all practical purposes, the client and its voter will fail together or not at all; and the client will not have to deal with the possibility of a failed voter.

The VVM also assumes a Byzantine failure model for applications running on servers, but only a crash failure model for the middleware and operating system code running on them. This model can be useful for applications that use data fusion or N-version programming, where the data to be collated are generated in multiple different ways and some may be inaccurate or erroneous. If this were not realistic for a given system, then a stronger failure model (weaker assumptions) could be provided with techniques such as the network attachment controller (NAC) in the Delta-4 system [11] or with more expensive protocols similar to those in [35] or [9]. This is being pursued by Trusted Information Systems, as discussed in Section 7.1.

### 3 Voting Definition Language

In this section we describe VDL. We first define its syntax, then describe the primitive operations supported in the language. We then describe confidence values. We note that this research is not a theoretical investigation of how to represent all voting algorithms. Rather, it is a pragmatic investigation into how to support voting in middleware, though supporting as wide as set of algorithms as possible is obviously useful.

#### 3.1 VDL Syntax

Figure 4 gives an overview of the VDL syntax. Symbols for parentheses and curly braces (e.g., "[]") are part of VDL’s syntax. Square brackets ("["]"), asterisks, and pluses are used to denote portions of syntax that may occur zero, one, or more times, in the traditional use of these symbols.

A VDL policy begins with the policy’s name and parameters, which can include long, double, the built-in enumerated types for operations in each state.

Following is the declaration of the quorum operation and its parameters. It can optionally be followed by a statement indicating that a named exception should be thrown to the client under certain conditions. These conditions are expressions built with the policy’s parameters and also condition variables that the voter core provides regarding the status of a vote. These are listed in Table 1. For example, elapsed_time denotes the milliseconds since the voter was initialized for a given vote, for example at the time a client’s request was sent out. It is useful, for example, to set a timeout in the quorum state, or for confidence values. Note that in the descriptions of the confidence variables all percentages are based on the total number of ballots.

The statement specifying the exclusion operation and its parameters follows the quorum statement. It can optionally be followed by a clause which indicates that, instead of being discarded, all values excluded by this operation are to be replaced by either the median value, the mean value, or a given value. The exclusion statement may contain multiple goto and exception statements in any order. A goto statement sends the vote back to the quorum state under certain conditions, which are specified with expressions involving the exact same set of parameters and variables as the throw statement for exceptions. This goto statement may also specify, with the using clause, a new operation and new parameters to use in the quorum state. This clause also specifies whether this quorum applies to all ballots received (qualifier total) or only those received after the ex-
cution of the goto (qualifier more).

Note that multiple exclusion statements are allowed in a policy. If there are more than one, any exclusion that can be branched to by a goto statement must specify a label to enable branching without ambiguity.

The VDL policy finally specifies the operation to be used for the collation state (there are presently no parameters to these — they just choose one), as well as the optional exceptions and branching to quorum exactly as exclusion allowed. After collation, there may be a branch to exclusion. Finally, a confidence expression may optionally be returned to indicate how much trust should be put in the vote.

If an exception is thrown by the VVM, the exception type is one defined by the middleware system (middleware typically requires clients to catch exceptions, e.g., SYSTEMEXCEPTION for CORBA) so as not to introduce new exceptions into the interface.

Table 2 shows the primitives for each state. Primitives in the exclusion state are applied after the values have been sorted.

3.2 Confidence Values

All other voters are aware of allow only binary output behavior: either the voter outputs a voted value, or either fails or refuses (and perhaps throws an exception). However, the choices of throwing an exception and returning a value represent two extremes. VDL allows the expression of confidence values, with multiple uses which we are exploring, in order to provide some middle ground between these extremes.

One use is to allow the client to decide how to use a reply based on how good it is perceived. The confidence value is presently specified in VDL, as described in Section 3.1. Alternately, we are investigating allowing its specification via a separate confidence definition language. This would allow pieces of code (confidence policies) to be written by a different person from the VDL programmer, and they also could potentially be reused with different VDL policies. No matter how its specified, the confidence value will allow the client to adapt with a much better granularity than the current boolean output behavior.

We are also investigating utilizing the confidence value for application-level adaptation while still providing voting and replication transparency. This is enabled by using delegates, which are proxies interposed between the application client and the middleware proxy (stub), that provide the same API as that proxy (the API of which the client has programmed to). In particular, we are investigating the use of delegates from the Quality Objects (QuO) framework [43]. QuO delegates are generated based on QuO’s Structure Description Language (SDL) [24] plus the middleware’s interface definition language (IDL) and the QuO’s contract description language (CDL). The SDL allows specification of adaptation strategies which are above the middleware layer yet below (and generally transparent to) the client.

Note that the voter cannot return a confidence in the method’s return value, because the method’s definition has fixed its signature. Alternate means of returning the value are being developed, analogous to a Unix system call “returning” an error report “on the side” via the errno variable. In CORBA, for example, one possibility for passing the confidence value involves context variables.

<table>
<thead>
<tr>
<th>CONDITION VARIABLES</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>elapsed_time</td>
<td>the time since vote initialized (msec)</td>
</tr>
<tr>
<td>pct_same</td>
<td>the percentage of same ballots</td>
</tr>
<tr>
<td>num_same</td>
<td>the number of same ballots</td>
</tr>
<tr>
<td>pct_remaining_total</td>
<td>the percentage of remaining ballots</td>
</tr>
<tr>
<td>num_remaining_total</td>
<td>the total number of remaining ballots</td>
</tr>
<tr>
<td>pct_remaining_now</td>
<td>the percentage of remaining ballots (after this exclusion)</td>
</tr>
<tr>
<td>num_remaining_now</td>
<td>the number of remaining ballots (after this exclusion)</td>
</tr>
<tr>
<td>pct_excluded_total</td>
<td>the percentage of excluded ballots</td>
</tr>
<tr>
<td>num_excluded_total</td>
<td>the total number of excluded ballots</td>
</tr>
<tr>
<td>pct_excluded_now</td>
<td>the percentage of excluded ballots (after this exclusion)</td>
</tr>
<tr>
<td>num_excluded_now</td>
<td>the number of excluded ballots (after this exclusion)</td>
</tr>
<tr>
<td>ballots_total</td>
<td>total # of ballots possible, e.g. number of replicas; if &gt; ballots_total, quorum does not wait for all ballots to arrive.</td>
</tr>
<tr>
<td>ballots_max</td>
<td>total # of ballots possible, e.g. number of replicas; if &gt; ballots_total, quorum does not wait for all ballots to arrive.</td>
</tr>
</tbody>
</table>

Table 1. VDL Condition Variables

<table>
<thead>
<tr>
<th>STATE</th>
<th>PRIMITIVES</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>quorum</td>
<td>until k[%]</td>
<td>wait until k [%] ballots arrive</td>
</tr>
<tr>
<td>all but k[%]</td>
<td>wait until all but k [%] ballots arrive</td>
<td></td>
</tr>
<tr>
<td>random n[%] m[%]</td>
<td>wait until a random number (range from n[%] to m [%]) of ballots arrive</td>
<td></td>
</tr>
<tr>
<td>exclusion</td>
<td>lowest n[%]</td>
<td>exclude the lowest n [%] values</td>
</tr>
<tr>
<td>highest n[%]</td>
<td>exclude the highest n [%] values</td>
<td></td>
</tr>
<tr>
<td>furthest n[%]</td>
<td>exclude the furthest n [%] values from the median</td>
<td></td>
</tr>
<tr>
<td>distance e from [mean</td>
<td>median</td>
<td>value]</td>
</tr>
<tr>
<td>distance_neighbor d</td>
<td>exclude all values that are not within distance d of their neighbor</td>
<td></td>
</tr>
<tr>
<td>distance_cluster d</td>
<td>exclude all values that are not within distance d time of their neighbor, starting at position specified</td>
<td></td>
</tr>
<tr>
<td>cluster_support d p</td>
<td>See example in Section 4.2</td>
<td></td>
</tr>
<tr>
<td>inner k</td>
<td>exclude the median plus k — 1 adjacent values, spaced as evenly as possible on each side of the median</td>
<td></td>
</tr>
<tr>
<td>nearest k</td>
<td>exclude the median and k — 1 values nearest it</td>
<td></td>
</tr>
<tr>
<td>random n[%]</td>
<td>exclude random n [%]</td>
<td></td>
</tr>
<tr>
<td>none</td>
<td>exclude none</td>
<td></td>
</tr>
<tr>
<td>coliation</td>
<td>median</td>
<td>choose the median value</td>
</tr>
<tr>
<td>mean</td>
<td>mean</td>
<td>choose the mean value</td>
</tr>
<tr>
<td>mean_neighbor</td>
<td>mode</td>
<td>choose the most common value</td>
</tr>
<tr>
<td>random</td>
<td>random</td>
<td>choose a random ballot’s value</td>
</tr>
</tbody>
</table>

Table 2. VDL Primitives
4 Examples

4.1 SuperMajority

Figure 5 gives an example of a supermajority voting policy implemented in VDL. The policy returns a vote if 60% of the ballots are equal, where “equal” is set to mean being within 1%. An exception is thrown if this cannot be met. Otherwise a confidence value is returned that is equal to the percent of ballots that agreed with the value voted upon (chosen in collation).

The policy initially waits for 60% of the ballots to arrive. It then discards all which are not equal. At this point, if more than 40% have been excluded, then a supermajority of 60% cannot be achieved, so an exception is thrown. If this does not happen, but less than 60% remain (which must be all equal), then the policy branches back to the quorum state to wait for more. If neither the branch nor the exception occur, then the supermajority has been met, and collation selects one value, the median. Finally, the confidence value is returned. This SuperMajority example can be parameterized so that the thresholds of 60% or 1% of the operation median, etc., can be passed in, for example by a voter manager or by a client application.

4.2 Fault-Tolerant Clocks

Clock synchronization is a very important problem in distributed computing area. The various approaches previously proposed have been categorized (in [32]) as master-slave, convergence function, interval, Byzantine agreement, and clock server algorithms.

We demonstrate the use of VDL to describe convergence function algorithms. A typical algorithm in this category contains four basic steps [32]:

- Collect clock values from other nodes,
- Manipulate the collected clock values,
- Calculate a correction term for local clock by applying a convergence function,
- Correct the local clock.

Among those steps, the first three can be mapped onto the VVM’s three voter states: quorum, exclusion and collation. The last step is to employ a correction algorithm to adjust the local clock, which would be the application’s function.

We demonstrate the use of VDL to express two convergence algorithms: Interactive Convergence and Fast Convergence. Figure 6 gives an example of the Interactive Convergence Algorithm (CNVA) [21] implemented in VDL. The synchronizing node first collects clock values from all other nodes, and then replaces the values that are “too far away” from local_value, where “too far away” means not within a given distance d from local_value. If all the values are excluded (replaced), an exception is thrown. After that, the synchronizing node calculates the corrected local clock value by choosing the mean of those manipulated values. Finally, the confidence value is computed according to how many values originally collected are excluded ( pct_remaining_total).

Figure 7 gives an example of the Fast Convergence Algorithm (FCA) [27] implemented in VDL. After collecting values from all other nodes, the synchronizing node excludes those values which receives support from less than p other values, where value_a “supports” value_b means |value_a - value_b| <= d. If all the values are excluded, an exception is thrown. After that, the synchronizing node calculates the corrected local clock value by choosing the mean of those manipulated values. Finally, the confidence value is computed as in CNVA.

5 Discussion

An obvious question is why bother with a separate voting language such as VDL, instead of just allowing Java code
to be uploaded for a given voting policy. In other words, is VDL just a middleware version of “syntactic sugar?”

The answer is “no”, for two primary reasons which indicate that VDL has its place. First, it is much easier to read and be able to understand what a voting policy is going to do, which aids reuse. Second, having a separate VDL allows the possibility of performing offline analysis of VDL to ascertain what tradeoffs a given policy offers in, for example, precision/correctness versus performance. This manageability allows for both adaptive and transparent voting, and thus, beneficially pushes much complexity away from the client and the middleware. Such manageability is one area of research we are beginning.

An initial prototype of a passively replicated version of the VVM architecture was completed at the end of 1999, with very limiting simplifying assumptions. We hope to create a much more realistic implementation. Actively replicated voters are more interesting and challenging. It is crucial to be able to design the VVM and VDL such that the voters are state machines: deterministic and able to make consistent decisions (votes and other outputs) without synchronization. To do this would require a number of new constraints, including ensuring that each voter gets all message requests/replies in the same order (e.g., with a virtually synchronous multicast system), so that synchronization between voters is not necessary; a voter cannot have a timeout; etc. Also, actively replicated voters will have to be seeded with the same exact random seed, at least on a per-connection (client-server relationship) basis.

Finally, in many configurations it will be unnecessary to replicate the VVM. In the case of an unreplicated client, in some configurations the VVM will be on the same host or even in the same address space as the client. In this case, the VVM can be considered to fail if, and only if, the client has failed, for all practical purposes (excluding software design faults).

6 Related Work

Much work has been done on voting in the last few decades, and in this section we constrain our research with that in the literature.

6.1 General Limitations on Parameter Support and Byte-by-Byte Voting

To date, the limitations on voting support in middleware — “majority” or “all” on only the return value — are dictated by the way such active replication schemes tend to handle the parameters of requests and replies, namely, as opaque blocks of data which can only be compared for equality. While this is much simpler to implement and configure, it has severe problems. Voting among active repli-

7
clean layering. Ideally the middleware’s lower levels should be able to treat the reliable multicast as a fancier version of a transport layer such as TCP or UDP. However, this is not possible, because the reliable multicast transport will need to rely on help from the middleware system to unmarshal the message data in order to be able to correctly perform value checking. This is necessary because unmarshalling the message requires information on the application-level data types in the particular message, which of course will depend on the method being called and the direction (request or reply). This implies that the middleware and the reliable multicast transport need to be closely coupled.

6.2 Collation Voting in Middleware

A number of recent projects have extended active replication to CORBA. These systems include Orbix-Isis and Electra [26, 22], Eternal [28], OGS [14], and AQuA [12]. These systems all have the virtue of providing a high degree of (but not total) application transparency, and also of being useful with commercial middleware. However, they all have a very limited form of voting: either voting on a return value in a server reply, not on other parameters in a reply or on any parameters in a request, or performing a naive byte-by-byte comparison of the marshaled parameter buffer. The Immune system provides survivability to CORBA applications via active replication, voting, and a secure multicast protocol [30]. Like Rampart [35], it thus covers a wider range of malicious faults than does the VVM. However, there are no details on the voting design or implementation other than it votes on both requests and replies, and waits for a majority “being identical in value”; we thus presume it employs byte-by-byte voting on the marshaled parameter buffer. As such, it will not work in the face of heterogeneity.

6.3 N-Version Programming

Active replication (which uses synchronization voting) has been developed to mask hardware failures of nodes and communication links. N-Version programming is a technique used to mask software design faults [10, 1]. In this approach, N versions of a module are independently developed from the same specification and executed on separate (typically heterogeneous) nodes. The return value or “answer” from each is collated using collation voting.

We note that N-Version programming suffers from the consistent comparison problem [7], which limits its use in error detection due to the finite precision in computer arithmetic. Two replicas can compute the same variable with slightly different values, then take different branches after a comparison that tested against a cutoff that was between the two values. This can, of course, result in the two replicas diverging. We do not believe that the VVM — even if replicated and used to vote in N-Version programs — suffers from the consistent comparison problem. It has only one version of the code (though even with multiple versions it would not have a problem, because it does not use floating point numbers in its control flow). So long as the presentation coding layer (“unmarshal” below) preserves the ordering between different floating point values across different architectures — a property of every encoding we have studied — then the VVM will work properly, even if its replicas are N-Version. Indeed, the VVM can be used to detect that an application replica has suffered from the consistent comparison problem and diverged, as well as detecting one that has performed a nondeterministic action (that the other replicas did not) and diverged for this reason.

6.4 Voting Algorithms

Theoretical work on voting dates back more than fifteen years. We have implemented, in VDL, various clock synchronization and other algorithms from some of the major work in this field. Section 4 provides three examples.

There are some algorithms which VDL and VVM cannot presently express, though, when we extend VDL to cover voting on multiple parameters, we hope to cover many of these. Examples include distance agreement protocols and distance decision [13], which involve voting in multidimensional space; the generalization of commonly used voting techniques such as Formalized Majority Voter, Generalized Median Voter, etc. in [23]; and the adjudicators and adjudication functions [16], which are generalized concepts of voters and voting algorithms. The Consensus Service is a technique for constructing agreement protocols [17].

Our research is not a theoretical attempt to provide a generalized voting service or notation that strives to express all possible voting algorithms. Rather, it is a pragmatic effort to create an embeddable middleware voting component. Certainly we have tried to make VDL be as general as possible — and it covers much of the space of voting algorithms in the literature, as discussed in [42] — but covering the entire space is not one of its goals.

There are also several software systems that are well known for their implementation works of voting. One of them is the UCLA DEDIX [2] system, which is a distributed testbed for multiple-version software. Another one is the fault-tolerant avionics application of algorithm diversity described in [8]. The EFTOS Voting Farm is a class of C functions that implements application-level voting for parallel message-passing systems[15], instead of more general distributed systems as with the VVM. None of these systems involve middleware.
7 Conclusions

In this paper we have presented the Voting Virtual Machine, a middleware voting component that can be used with different middleware frameworks. Its companion Voting Definition Language allows specification of portable voting algorithms that can be used with different applications and middleware frameworks. Together, the VVM and VDL separate voting policies from the application and voter core, which can help applications deal with the complexities involved with integrating fault tolerance into applications. Finally, we have described the limitations of the byte-by-byte value comparison performed in virtually all middleware and group communication systems, and overviewed the architectural implications for using such systems in the face of heterogeneity.

7.1 Status

We have developed an initial prototype of most components and primitives identified in this paper, and plan a public release of the C++ version, and hopefully also the Java version, in early summer 2001.

A number of parties are interested in using the VVM. NAI Labs (formerly Trusted Information Systems) is very interested in evaluating it for use in its DARPA-funded IT-DOS project [29]. This project involves building support for Byzantine failures in CORBA, using TAO [37] and the multicast algorithms in [9]. It explicitly identified the need for something like the VVM (not just byte-by-byte voting) in their research proposal. BBN is actively considering using the VVM for application-level data fusion in its DARPA-funded APOD project [6]. Finally, TriGeo Network Security plans to use the VVM to adaptively collate and filter intrusion detection alerts [40].

7.2 Ongoing and Future Work

We are currently involved in research in a number of areas that build on the research presented in this paper. An important task is to complete an exhaustive search of the voting literature to ascertain ways in which the VVM and VDL could and should be extended to offer broader coverage. A byproduct will be a large collection of voting algorithms in VDL. Another part of our research involves developing tools to ascertain what tradeoffs between fault tolerance, correctness, and performance a given VDL policy provides. We believe this will require mandating a subset of VDL to enable a policy to be analyzable. The results of this analysis will be used in our research to develop voting managers that choose the best voting algorithm (or at least a reasonable one) for the given operating conditions, perhaps given high-level requirements from the application regarding its preferred tradeoffs between fault tolerance, precision/correctness, and performance, as well as what safety properties could be guaranteed. We also plan to apply the VVM to a much wider range of application architectures, including replicated mobile agents, distributed sensor networks, and intrusion detection systems.

Acknowledgements

We thank Raghava Kashyapa, Marius Sundbakken, and Akran Abou-Emara for their work on the Voting VM, as well as Doug Blough and his students for their feedback. The comments from the anonymous referees greatly improved this paper.

References
