THE VOTING VIRTUAL MACHINE: VOTING SUPPORT FOR DISTRIBUTED SYSTEMS

By
CHRISTOPHER C. JONES

WSU EECS Technical Report number EECS2000_0003

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science
MAY 2000

Acknowledgments

I would like to thank my advisor, Dr. David Bakken, for his plethora of advice, unmeasurable patience, and encouragement. His relentless drive for quality and abundance of energy has been extremely valuable in both my research and in writing this thesis. I would also like to thank my committee members, Dr. Krishna Sivalingam and Jack Hagemeister, for their time and advice. Also, I thank Ms. Ruby Young, the graduate secretary, for all her help, and to the School of Electrical Engineering and Computer Science of Washington State University for its support.

Thank you to Dr. David Karr of BBN Technologies for his collaboration on the VVM. Also, I thank BBN Technologies/GTE Internetworking for funding the VVM patent development costs.

Also, I thank Dr. John Hale for encouraging me to attend graduate school and imparting his wisdom upon me. I thank Zhiyuan "Troy" Zhan for suggesting the VSS query service, and I thank Tony Keen be thanked for answering my Java questions and for his friendship. I thank all the friends I have made during my years at Washington State University, for the good times and many laughs.

I would like to thank my parents for the love and support they have provided me through the six years of my higher education career. I could not have succeeded without knowing they were behind me the whole way. Lastly, I would like to thank my fiance, Christine, for her constant support, help, and love. She was there through the good and bad, and never once stopped believing in my abilities. I could not have finished without her.

The Voting Virtual Machine: Voting Support for Distributed Systems Middleware

by Christopher C. Jones , M.S.
Washington State University
May 2000

Chair: David Bakken

Distributed systems in which clients and servers are replicated often require a mechanism for voting on multiple requests to or replies from a server. Voting is the process of choosing a single reply or request using some specified policy that defines how that single reply or request is determined. Currently, there exist very few systems employing such a voting mechanism. Those that do exist lack flexibility and are not easily embedded into middleware architecture. Work completed for this thesis includes the development of the Voting Virtual Machine(VVM), a voting mechanism that addresses many of the current obstacles within the area of voting in distributed systems.

The VVM provides a powerful, general, portable, flexible, and adaptive voting mechanism. It can be used as a stand alone application for the development and testing of voting algorithms or it can be inserted into middleware systems like CORBA and Microsoft's DCOM. Also, the VVM may be applied to other architectures that require voting, such as quorum consensus, collation engines, merging of partitioned replicas, and agent-based systems. Along with the provision of a voting service, the voter tabulates the values and timing of replicas' replies for performance and security monitoring. This thesis also investigates security issues involving the VVM, including the tolerance of faulty values within different voting policies and the feasibility of an adversary inferring the current voting policy.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The explosive growth of the Internet in the last few years has thrust distributed systems software from the realm of research projects and hand-crafted specialty applications to becoming common, almost ubiquitous. Simultaneously, computer hardware has become much cheaper. These two trends have allowed for the rapid growth of software being distributed across corporate networks, intranets, and extranets. As a result, society is relying much more heavily on services provided by distributed software.

Replication is a commonly used technique to increase the availability of these services and to lower the delay in accessing them, especially in the case of read-only accesses. One major form of replication is Active replication [1, 2]. In this scheme, the service request from a client is multicast to all replicas atomically and totally ordered. Each replica then processes the request and sends back a reply, and one of these 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. This voting process is depicted in Figure 1.1.

Work completed for this thesis pertains to voting issues that are applicable to other areas besides replicated services. The Voting Virtual Machine is a general voting mechanism that can be applied to quorum consensus, a strategy that is used for replica control in which the client obtains a quorum of votes from the replicas before proceeding [3]. Another use for the VVM is in the area of more general collation engines, where replicas are not assumed to return the same values and, indeed, may not even be considered replicas in the given system. However, these systems require a single collated value; examples include network management, Quality of Service(QoS) managers, and fault-tolerant time synchronization in Tempo[4] and NTP[5]. There are many times when a network partition will break up a set of replicas. In some systems the non-primary partition is allowed to continue operations. When the replicas merge back into a single group, it is necessary to merge their states so that all replicas maintain the same state. The VVM could be used in this merging of replica state minimizing the amount of programming required of system programmers. A further potential application area for the VVM is distributed sensor networks

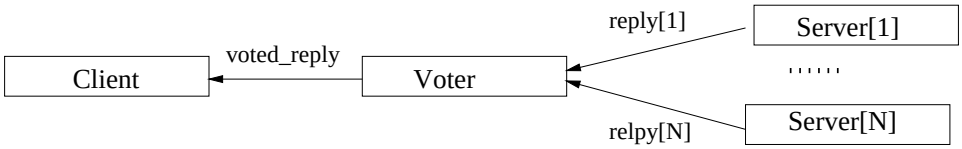

Figure 1.1: Voting Context

(DSNs) [6]. These consist of large numbers of inexpensive, small, dispersed cooperating sensors, whose values of course must be collated to provide a more global view. Application areas for DSNs include surveillance and environmental modeling.

Another application of voting is within agent-based systems where fault-tolerance and security are necessary characteristics. Agents migrate from computer to computer for various reasons (e.g. data collection), and the agents are susceptible to an adversary that wants to corrupt the agent's operation or data. The use of voting in agent-based systems is discussed in [7], where replication and voting are used to mask the effects of corrupted agents. This is implemented by replicating agents at each computer visited, and a voter located at each computer waits for a specified number of equivalent replicas.

An additional promising area in which the VVM can be applied is in the fusion or reduction of data necessary for intrusion detection [8]. As a concrete example, consider the Hummer system, which focuses on system support to share and collate intrusion detection information [9]. Hummer agents run on multiple machines, collecting local information related to potential intrusions from system logs and other local sources. Agents answer queries from other agents about local conditions. A given agent must also query multiple other agents to build a view of the intrusion status of the hosts in the system. In doing so, it must deal with the possibility of value faults caused by Hummer agents running on compromised hosts. This, of course, can be done in a very flexible and adaptable manner by using the VVM. It is also possible to construct hierarchies of Hummer agents, where each higher level (LANs, domains, etc). summarizes the conditions of lower levels with a more global perspective and with less network traffic. This is also a good candidate for insertion of the VVM. Finally, if a client must rely on a non-local Hummer agent (e.g., at a high level in the hierarchy), the client must deal with the potential corruption of a given Hummer agent. This, too, can be implemented using the VVM.

In all the above uses of the VVM, the VVM itself can serve as an intrusion detection status server, notifying others of value faults from Hummer agents. This use of the VVM will be discussed in Section 3.8.

The VVM not only detects faulty values in other applications, but can also impede or thwart attacks on the voter through adaptive voting policies. A voting policy defines how the voter will select a single answer; different policies can help against different attacks. If a malicious user knows which policy the VVM is currently using, then they may be able to bypass detection and corrupt the replies. The VVM has the ability to protect the policy from being learned by the malicious user and to protect the replies from corruption.

While voting has wide application, current support for voting is quite limited in both research projects and in commercial products and standards. This is dictated by the way such active replication schemes tend to handle the parameters of requests and replies, namely, as opaque blocks of data that can only be compared for equality. As a result, voting among active replicas generally consists of delivering a reply from the server replicas when some quorum of the servers (e.g. one, the majority, or all) have sent replies exactly identical byte-for-byte to the one being delivered. For the purpose of this kind of voting procedure, the return value, **out** parameters, and **inout** parameters are all included in the "reply" without distinction. In recent systems that also support replicated clients, a similarly limited functionality may be provided to vote on client requests before delivering one to the server [10, 11].

For example, consider the following CORBA interface:

```
interface foo {
                long method1(in long a);
                void method2(in long d,
                             inout short e,
                             out double f);
}
```

A request to *method1* could potentially be voted on based on the parameter a, while the reply could be voted on based on the return value. In *method2*, however, the request could be voted on by some function of parameters d and e, while the replies could be voted on using e and f. (Note that $method2$ has no return value.) With the current voting strategies supported in middleware, however, the only individual datum which can be voted on is the return value from $method1$.

In the absence of any voting algorithm capable of distinguishing the different parameters in a message, the technique of byte-by-byte comparison depends on properties of data marshalling that in some cases are not guaranteed by the relevant standards. Important semantics of the components of the data being voted on, which depend on the signature of the method being implemented (i.e., the order and data types of parameters and return value) are ignored. They are also very inflexible in dealing with floating point values in particular, as well as with inexact voting in general. Moreover, little progress has been made toward the ability to change voting algorithms at runtime in response to changing conditions or requirements and tradeoffs.

This thesis describes the design and implementation of the Voting Virtual Machine, a voting mechanism designed to help overcome the limitation of current voting mechanisms, and discusses the security aspects of utilizing a voter within middleware. The contributions of this thesis are:

- The design and prototype implementation of the Voting Virtual Machine(VVM) that can be used in distributed middleware systems.
- The design of Voting Description Language(VDL) to enable a user to define a voting policy for the Voting Virtual Machine. This is the first language we are aware of that allows for expressing voting algorithms.
- The design of the Voting Status Service(VSS), an anomaly detection service that collects and processes data about each ballot enabling the detection of abnormal behavior.
- The investigation of the types and number of faulty values the Voting Virtual Machine can tolerate in the space of all policies expressible by Voting Description Language and the likeliness of the Voting Virtual Machine's removing those faulty values.
- A security inference study characterizing the level of difficulty for a malicious user to learn a voter's policy and how useful knowledge of the policy might be to a malicious user.

The remainder of this thesis is organized as follows. Background work is reviewed in Chapter 2. Chapter 3 provides an overview of the VVM context, basics of the VVM operations, advanced operations of the VVM, the VVM development environment, the VDL, and VSS. Chapter 4 gives an overview of the VVM's operations that provide the toleration of faulty values, while Chapter 5 provides a summary of how the VVM can resist malicious attacks. Chapter 6 discusses various implementation issues and the current implementation status of the VVM. Chapter 7 summarizes this thesis and discusses future research on VVM.

Chapter 2

Related Work

This chapter discusses some of the research into voting and the current state of voting. First covered is the current voting technique of byte-by-byte comparison for collation and its drawbacks. Next, the use of voting to synchronize operations running on replicas. The third section of this chapter discusses the past work in choosing one value from many in replication. The idea of independently developing replicas from the same specification is called N-Version programming and is the topic of the fourth section. The last section of this chapter discusses the work done in voting algorithms and how the VDL allows a developer to implement all the classes of algorithms for the voter.

2.1 A Note on Byte-by-Byte Voting

With respect to the byte-by-byte voting used by many collation voting schemes, we note that CORBA strives for interoperability across multiple dimensions:

- CPU architecture
- operating system
- programming language
- ORB implementation

This interoperability (and its elegant architecture) is a big reason for CORBA's success in the marketplace, in the opinion of the authors. For this reason, however, performing a byte-by-byte comparison of the marshalled parameter buffer has limitations, ones that would be considered severe in real CORBA usage, and in some cases can be impossible. For example, CDR uses a canonical intermediate form for basic types, including real numbers. However, if the same application code is being used on different machine architectures (even with the same ORB vendor, OS, and language), there may be different levels of precision or other differences in the (IEEE 754) encoding of the floating point value. Further, the Pentium family of chips has a switch, set in hardware, that tells the processor to round up or down. Thus, even with the same CPU and OS and language and ORB, different hosts may return different answers. Also, GIOP messages may have variable-length header information such as system context that ORB implementations can optionally fill in or leave out. Thus, even with the same CPU, OS, and language, there can be padding in the body field (marshaled parameters) of one copy of of a message from one replica and no padding in the other. It is impossible to detect this and correct the alignment if the IDL is not used to interpret

the header and parameter buffer. Finally, even if the padding locations are identical, the CORBA specification explicitly states that the value of the padding bytes is undefined, which could cause problems in byte-by-byte comparison [12].

As a result of the above limitations, it is quite likely that a user of a replication service based on byte-by-byte voting would have no problems in its in-house testing and development, which would tend to be fairly homogeneous in many dimensions (ORB vendor, OS, CPU architecture). However, when their software was fielded or released to others, the homogeneity could greatly diminish and the above-noted limitations of byte-by-byte voting could cause sporadic and mysterious runtime errors. By contrast, the VVM is the only CORBA (or any other middleware) system we are aware of that examines the application-level data parameters in a presentation-layer message on which to perform the voting.

Finally, we note that these limitations of the byte-by-byte approach are not limited to CORBA; all distributed systems have to deal with presentation-level formatting as well as most or all of the interoperability dimensions that CORBA provides.

2.2 Synchronous Voting

Voting, in the most common technical use of the term, is a pessimistic strategy for replica control that ensures that conflicting operations will not be executed concurrently. We denote this kind of voting *synchronization voting.* In this scheme, a sufficient number of votes must be acquired from different replicas to ensure that a candidate operation does not conflict with another one in progress. The number of votes a given operation requires to not conflict is application-dependent, and must be set by the application programmer.

Synchronization voting was first proposed by Thomas of BBN in [13]. It was generalized to weighted voting in [14], where different replicas are given a different number of votes. There have been a number of generalizations of voting such as dynamic voting [15], multi-dimensional voting [16], and voting with witnesses [17]. A recent and scalable example is Phalanx [18].

BBN's Cronus (CORBA-like) middleware [19, 20, 21] has had replication with synchronization voting support since the mid-1980s; Arjuna is a more recent system using such mechanisms [22, 23, 24]. Much experience was gained with voting applications during this period in which Cronus was deployed widely in various military settings. However, the experience with Cronus showed that synchronization voting as a general mechanism was too difficult for the vast majority of programmers to use.

2.3 Collation Voting

For these reasons, much attention in recent years has turned to methods of replica control such as active replication, which have the potential to be more application-transparent. Active replication is another pessimistic strategy for replica control. It uses voting in the sense of "collating," or choosing one reply from among many. We call this kind of voting *collation voting.* Synchronization voting preceded active replication, however. One early example of collation voting is given in [25]. The earliest example of collation voting we are aware of is the SIFT (Software-Implemented Fault Tolerance) project at SRI[26]. SIFT was designed for real-time aircraft control, and featured eight processors running in loose synchrony on the order of 50 μsec. SIFT's application software is structured as a set of iterative tasks that were run at a frequency that depended on its priority. Each task is executed in parallel on a number of independent processors (on no more than on five of the eight processors), and the output of each task is placed in a buffer. The buffers for each task replica are voted on with a majority vote (i.e., 3 of 5 or 2 of 3) in an "exact match basis," and the voted buffer is used for input to the next task to run. SIFT had no knowledge

of the application-level data types, and had fixed voting algorithms and was embedded in a a special-purpose computer system. This is in contrast with the VVM, which is more generally usable by different applications and middleware substrates.

Much work on active replication (and on other forms, such as passive replication) was done by the Delta-4 project in Europe [27, 28]. However, replication support was not provided in an application-transparent fashion, nor bundled in a package for use with commercial software. The Rampart system extended active replication to tolerate Byzantine failures[29, 30], using Secure Agreement Protocols[31]. As such, it covers a wider range of malicious faults than does the VVM. While details on the comparison mechanisms are sketchy, it appears that they utilize byte-by-byte comparison of network-level messages. It thus has the same limitations that current CORBA research with active replication has, as does Delta-4; these are outlined below.

A number of recent projects have extended active replication to CORBA. These systems include Orbix+Isis and Electra [32, 33], Eternal [34], AQuA [35]. These systems all have the virtue of providing a high degree (but not total) application transparency and also of being useful with commercial middleware. However, they all had a very limited form of voting: either only voting on a return value in server reply, not on other parameters in a reply or on any parameters in a request; or doing a naive byte-by-byte comparison of the marshalled parameter buffer. The Immune system provides survivability to CORBA applications via active replication, voting, and a secure multicast protocol [11]. It thus covers a wider range of malicious faults than does the VVM, like Rampart. 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 marshalled parameter buffer.

2.4 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 [36, 37]. 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. However, N-Version programming has not proved to be practical for any but highly critical systems, because of the expense of developing multiple versions and because an erroneous or ambiguous specification can lead to different design teams coding the same errors in their modules. N-Version programming also cannot be used in cases where the different algorithms can validly return different correct answers (such as a resource manager's reconfiguration decision or in many areas of computational mathematics). N-Version programming also suffers from the consistent comparison problem [38], which limits its use in error detection due to the finite precision in computer arithmetic. (Our research will have to ascertain the extent to which these problems apply to the less general context of the VVM, where the implementations are not different). We are unaware of any N-Version research that either provided for more than just a few fixed voting algorithms, or in which the voting was embedded in middleware and which interpreted the application-level data in the marshalled network message.

2.5 Voting Algorithms

A *voting algorithm* defines how a voter should combine multiple inputs to produce a single output. Such algorithms vary with input types and the computed output types. A survey paper has created a taxonomy of voting algorithms [39], and we describe how well the VDL covers this space of voting algorithms in Section 3.6.1, after we have described the VDL. As we show, the VVM can implement much of the space

of voting algorithms in this taxonomy, and, we believe, some not found in the literature. However, the novelty of our research lies not in the algorithms it supports, but rather in its abitlity to be inserted in different middlware substrates, its independent Voting Description Language, the Voting Status Service, and our analyses of the security properties of the VVM (which apply to many algorithms from the taxonomy).

Voting with predispositions [40] provides many voting algorithms which are provably optimal and take advantage of probability distributions of faulty nodes, non-faulty nodes, and legal values for a given service (i.e., server method). This work also compares reliability of majority, median, and plurality voting algorithms under various assumptions. This could be the basis of future VVM voting managers which choose the most optimal algorithm for the given configuration. Such a manager would also consider other tradeoffs such as security, availability, and correctness versus performance, as discussed in Section 7.2.1.

Chapter 3

Voting Virtual Machine Architecture

3.1 Overview

This thesis research has undertaken the design and implementation of Voting Virtual Machine (VVM), and its companion Voting Description Language (VDL) policy language. They provide runtime adaptive voting policies that change voting algorithms dynamically to trade off correctness and performance in the face of changing resource availability, user preferences, and security needs. They can work with presentation layers from different kinds of middleware architectures, such as those for distributed objects (CORBA[41] and DCOM[42]), message-oriented middleware[43] (IBM's MQSeries[44] or Microsoft's Message Queue (MSMQ), as well as with other kinds of middleware. They support both *static collation voting*, where the voting algorithms cannot be influenced by group membership changes, and *dynamic collation voting*, where they can be so influenced. Finally, VVM and VDL are designed not only to support goals of just higher availability, but also provide tradeoffs against performance and security concerns.

Figure 3.1 gives a simplified picture of the VVM architecture, as applied to CORBA; in this figure, the dashed box represents the module labeled "Voter" in Figure 1.1.

The voter core module has inputs that inform it of the current voting policy (VDL code fragment) as well as notifying it when a failure occurs. It has outputs to provide 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 is passed to the Voting Status Service, which is described further in Section 3.8.

It is worth noting that there may well be different programming languages involved here. For example, CORBA defines mappings from its IDL into a number of different programming languages, including Java, C++, C, COBOL, Ada, and Smalltalk. Thus, an application's client may be written in C++ or COBOL, while its server may be in Ada or Smalltalk. At each end of the client-server path the CORBA Object Request Broker (ORB) translates between the linearized IIOP messages and the client or server's implementation language. However, the VVM's unmarshal and marshal modules translate between linearized network messages and the voter core's implementation language, which is currently Java.

3.2 Failure Model

The VVM assumes a crash failure model for the hosts it runs on. This is a reasonable assumption, 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; the client will not have to deal with the possibility of a failed voter.

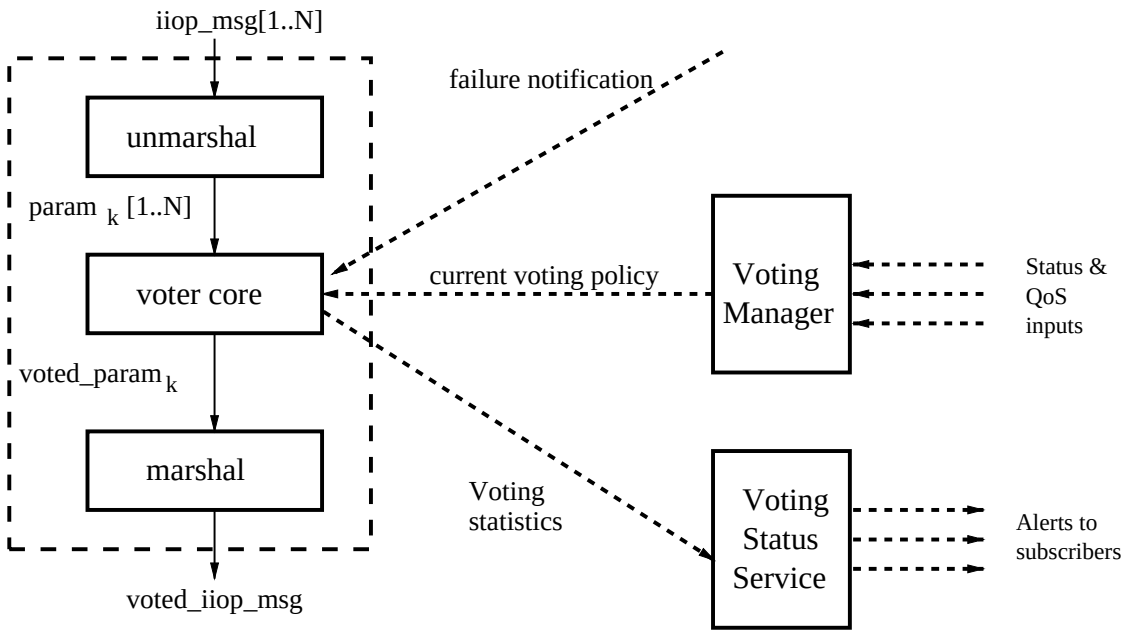

Figure 3.1: VVM Architecture

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. In many cases this will be acceptable, because in almost all systems it is easier to compromise user-level priveleges than adminstrator (root) ones. 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 [2] or with more expensive protocols similar to those in the Rampart system [30].

3.3 Basic VVM Primitives

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.

The voter core shown in Figure 3.1 goes through three states in the processing of a single vote, as depicted in Figure 3.2. At each stage, the VDL code for the current voting policy dictates the action of the voter at each stage. We now describe the basic operations that can be performed in each state, then we describe more advanced features in the following section. Complex examples of their syntax are given in section 3.6.

3.3.1 Quorum

While the voter is in the quorum state, it is waiting for enough ballots to arrive to begin the actual voting process. In the quorum state the following operations are currently available to specify how many ballots to wait for:

M: wait for M ballots

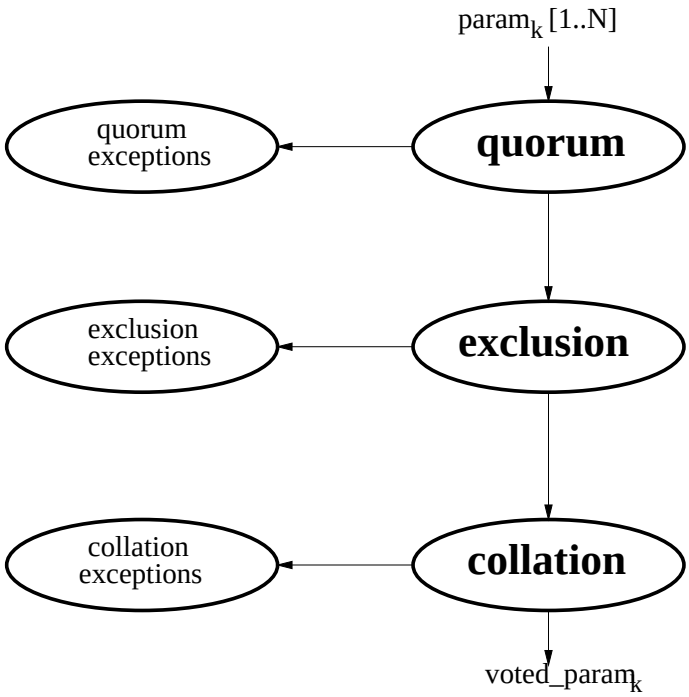

Figure 3.2: Voter States

all but *k*: wait for *M* - *k* ballots

percent: wait for x% of the ballots

The first operation involves static voting, while the last three involve dynamic voting.

3.3.2 Exclusion

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. caused by a faulty (and undetected) floating point processor or a breach in security (for example, an adversary could break into either the application of the middleware in order to insert messages that look like correct IIOP messages, but contain incorrect application data). Operations presently available in the exclusion state are:

lowest n: exclude the lowest n values

highest n: exclude the highest n values

furthest n: exclude the furthest n values (from the median, for integers; is high and low together)

distance e: exclude all values e distance from the mean (for floating point values) or median (for integers)

sigma x: exclude all values x sigma from the mean (for floating point values)

distance-neighbor: exclude all values that are not within a given distance of their neighbor

distance-cluster: exclude all values that are not within a given distance of each other

none: exclude none

3.3.3 Collation

After exclusion is the collation state. Here, one value is chosen from those not excluded. Operations that can be performed to collate include:

median: choose the median value

mean: choose the mean value (for floating point parameters)

mean-neighbor: choose the closest neighboring value from the mean value

mode: choose the mode value (the most common one; probably makes little sense for non-integer values)

At each of the three states exceptions may be thrown as directed by the VDL; in this 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. In the exclusion state an exception could be thrown if too many members are excluded, and in the collation state if the number of ballots which included the voted-upon value was too small.

3.4 Advanced VVM Primitives

3.4.1 Random Voting Operations

The voter can be specified to perform random operations at any or all of the three states, to help thwart an adversary; 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 number could potentially be reset at different time intervals. Finally, the collation state can randomly choose a value. This may well be reasonable in some applications where the particularly bad data have already been discarded in the exclusion state.

The uses of random voting to enhance security are discussed in Section 5.1.5

3.4.2 Weighted Voting

Another way in which the VVM has the potential to improve security is by weighted voting, or giving different replicas varying amounts of trust in the different states of the voting. For example, in the quorum state, instead of waiting for a given number of ballots, it can wait for some number of *points*, where each replica is assigned a different number of points. When a ballot arrives, that ballot's arrival counts toward the number of points dictated by its quorum weighting. In the collation state, the operations can be performed after the remaining ballots are expanded, based on a weighting.

In the following example, replica one has a point value of two, replica four has a point value three, and replica three has a weight of three. Then the ballot info would look like this: {replica number, value, points, weight} So, we would have {replica one, 5, 2, 1}, {replica two, 4, 1, 1}, {replica three, 6, 1, 3}, {replica four, 3, 3, 1}. Replicas five and six will have point and weight values of 1 with values {5} and {3} respectively. Replicas seven, eight, and nine all have different points and weights, but will arrive at the voter after it has matched the needed points for a full quorum. The voter policy is: ballot quorum 9 exclusion none collation median. The set of ballots that the quorum passes to the exclusion state is {3,5,4,3,6,5}. These were the first ballots whose point value matched the quorum threshold of 9. In this policy, we exclude none. Before we sort and find the median, we need to expand the set of values according to the weight of each value. The value from replica three will be expanded. The new set of values is {3,5,4,3,6,6,6,5}. After the set is sorted it looks like {3,3,4,5,5,6,6,6}. The median is found and it is sent to the client.

Using weighted voting for added security benefits is explained in Section 5.1.5.

3.5 VVM Development Environment

We are developing an integrated development environment with a number of tools to help development and deployment of VVM-based systems. They are shown in Figure 3.3. In this figure, single lines denote runtime interactions that occur synchronously to the arrival of a ballot, dashed lines show runtime interactions that happen asynchronously to the arrival of a ballot, and double lines indicate interactions that happen at compile time.

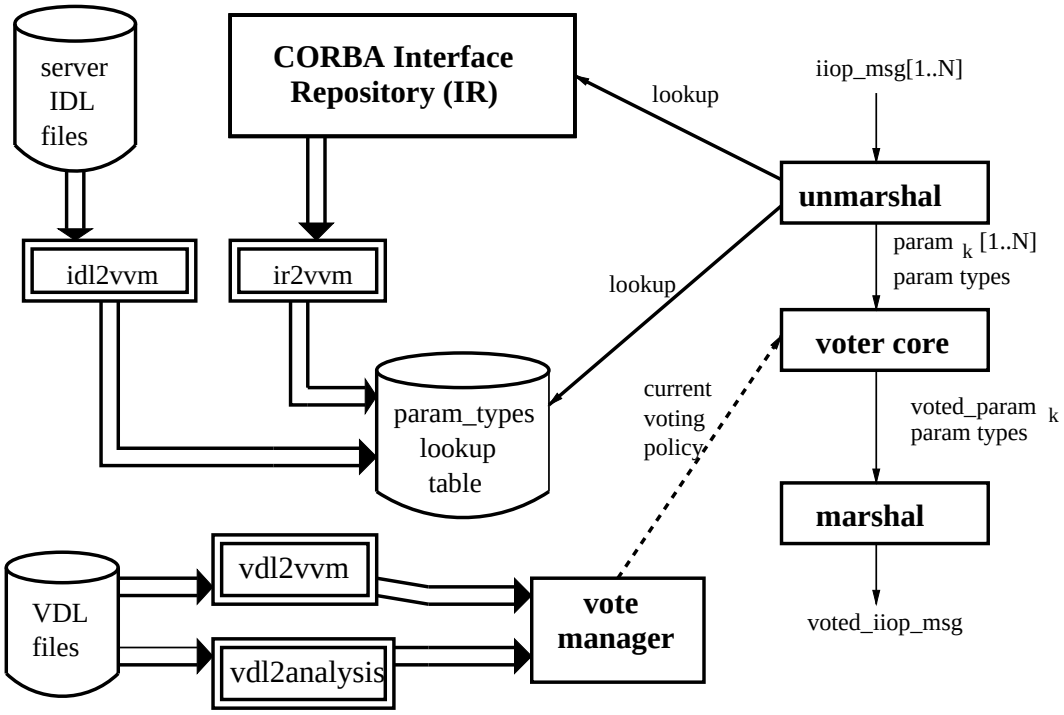

Figure 3.3: VVM Development Environment

The right side of Figure 3.3 is a subset of Figure 3.1. When a message arrives at the unmarshal module, it reads from the message's header the interface name, the method name, and the direction (request or reply). From this, it consults a lookup table to ascertain the types the application-level parameters in the message. If the lookup table does not have information on this interface, unmarshal looks it up in the CORBA interface repository (IR), then maps the returned IDL types of the parameters into their corresponding Java types. If the IR does not have information on that interface, the voter must throw an exception.

After this, unmarshal extracts the parameters from the IIOP message (converting from CDR's representation of that type if necessary), then passes them ($param_k$) to the voter core, along with the type information ($param\_types$). The voter core operates as described in previous sections, then passes along a voted set of parameters ($voted\_params$) along with param_types to marshal. Marshal then uses this information to create an IIOP message in a manner that is largely the reverse of unmarshal, as described above.

Two tools, `idl2vvm` and `ir2vvm`, are used to preload $param\_types$ into the lookup table at compile time. Therefore, some or all of the interfaces with which a voter will be involved with are known *a priori*. This saves an expensive runtime lookup to the CORBA Interface Repository (IR). Tool `idl2vvm` takes an IDL file and loads its parameter information into a file that is used the next time the lookup table is built. It is built using the Flick Compiler toolkit [45]. (The `idl2vvm` tool is used for unmarshal modules of both CORBA's IIOP as well as for our test wire protocol, which for convenience's sake uses CORBA's IDL to describe interfaces.) The `ir2vvm` tool loads the same information from a CORBA IR, which is a part of every CORBA ORB. This can be useful, for example, when the name of an interface is known, but its IDL file is not available.

The VDL compiler, `vdl2vvm`, takes a VDL file and loads it into the vote manager, which controls the voting policy in use by voter core. We are also designing analysis tools, collectively called `vdl2analysis`, to help the vote manager know the quantitative tradeoffs of a given policy: its costs and benefits under various operating conditions and with various group memberships and performance. This is particularly useful, for example, when the vote manager is working in coordination with a framework such as QuO, which can provide higher-level feedback on which tradeoffs are appropriate at the current time [46, 47]. We have begun the design and implementation of a prototype integration of the VVM and QuO.

3.6 Voting Description Language

The Voting Description Language (VDL) is under development, and is currently fairly simple. It allows the specification of the operations as defined in Sections 3.3 above. The language as it stand now allows the user to specify how to determine the number of ballots the voter needs to wait for, what to exclude and the number of ballots to exclude, and how to select or calculate the voted answer. Currently, it does not allow the specification of exceptions; this is future work, as described in Section 7.2.4.

We now give some English descriptions of voting policies, followed by the VDL code fragments:

- "wait until four ballots have arrived, exclude the lowest one, then choose a random one from those left"

  ```
  quorum 4 exclusion lowest 1 collation random;
  ```

- "wait until half of the ballots have arrived, exclude a random one, then choose the median of those left"

```
quorum 50 percent exclusion random 1 collation
median;
```

- "wait until all but two of the messages have arrived, exclude the two highest-valued ones, then choose the most common one left"

```
quorum all but 2 exclusion highest 2 collation
mode;
```

- "wait for six replicas' ballots, exclude the values that are greater than three away in value from the median value, then randomly select one from the remaining ballots"

```
quorum 6 exclusion distance 3 collation random;
```

- "wait until two-thirds of the ballots have arrived, exclude all values that have distances more than two away from the rest of set, then chose the most common one left"

```
quorum 66 percent exclusion distance-cluster 2
collation mode;
```

- "wait for all the replicas to send ballots, exclude the 2 highest and 2 lowest valued ones, then chose the closest value to the mean value"

```
quorum absolute exclusion furthest 2 collation
mean-neighbor;
```

The VDL examples above are for *ballot-based quorums*, which are quorums that depend on the number of ballots received. The following examples are *value-based quorums* where a specified number of equal valued ballots determines whether the quorum is full or not.

- "wait until two values are equal or are close in value to arrive, exclude the values that are greater that two away in value from the median value, then calculate the mean value"

```
quorum 2 equal exclusion distance 2 collation mean;
```

- "wait until three values are equal or are close in value to arrive, exclude the one lowest value, then select the most common value from the remaining ones"

```
quorum 3 equal exclusion lowest 1 collation mode;
```

Further additions to the VDL will be added to specify exceptions for each state and define some sort of fault tolerant properties for the voter. Some of these issues are discussed in Section 7.2.4. The different voting policies have certain fault tolerant properties. These properties are what the next chapter discusses. The expressiveness of the VDL with respect to voting algorithms is reviewed next.

3.6.1 Voting Algorithms

A voting algorithm specifies how the voting result is obtained from the input values and can be the basis for a voting system(i.e. Voting Virtual Machine). Voting algorithms can be classified by their input and output data and by their ballots and voted answer. Figure 3.4 shows the classification of algorithms from [39]. This 4-cube classification leads to 16 classes of voting algorithms. The VVM and VDL are designed to let the user implement most of these classes.

	Input	Output
Data	Exact/ Inexact	Consensus/ Compromise
Vote	Preset/ Adaptive	Threshold/ Plurality

Figure 3.4: Voting Algorithms Taxonomy

Exact/Inexact: Exact input means that the values from the replicas are inflexible, while inexact means that the values are representing flexible region. An exact value, for example, is 3.6, and 3.6 is not equal to 3.5, while inexact values would say that 3.6 is the same as 3.5 and so they are equal in value.

Exact input is the default manner in which the VDL handles the input values. Inexact inputs are handled by two exclusion operations, the **distance-neighbor** and **distance-cluster**. These operations allow the user to stipulate a maximum distance between the values. These operations let a user define the valid values that the collation state will work on. Examples of these operation are in Section 3.6.

Consensus/Compromise: Consensus output means that the voter is looking for a most common value. Compromise voting is defined as voting schemes using mean or median to select output value.

Consensus output is supported by the VDL with the use of the **mode** collation operation with an exception that defines the number of common values needed for the mode operation to select the answer. The compromise output is support by the **mean, mean-neighbor,** and **median** collation operations.

Preset/Adaptive: Preset voting is when weights are set at the start of the system and do not change. Adaptive voting allows the modification of the weights during operation.

The preset weights are supported by allowing the user to set the weights at the start of the system and not allow any change to the weights during the system's run. Adaptive voting is support by allowing the user to manually change the weights or to have the loopback feed change the weights automatically.

Threshold/Plurality: Threshold voting deals with setting a required number of values needed, which is a majority of the values received, in order to meet a consensus or make a compromise. Plurality voting allows consensus and compromise voting without needing a majority of the values supporting the answer.

The VDL supports threshold voting with the use of the VDL exceptions that are used in conjunction with consensus output. There are different threshold schemes, including majority, Byzantine, and unanimous. Majority requires that at least half of the values in the collation state are equal, such that the mode is supported by more that half of the values. Byzantine requires at least two-thirds support for a mode to be correct. The unanimous scheme requires that all the values in the collation state are equal in order for the voter to send the answer to the client. If collation state's values do not meet the threshold requirement, then the voter will throw an exception. The VDL supports plurality output with the same exceptions that the threshold output uses, but it relaxes the majority needed in consensus output.

Examples/Discussion: Table 3.1 displays examples of VDL that covers half of the voting classes. The table shows only those classes with exact voting. In this table and Table 3.2, the threshold voting is covered by the VDL, using a collation state exception. The exception will state a specified number of ballots that must support the voted answer and without which an exception will be thrown back to client. This makes

the most sense for consensus output where the threshold will stipulate that the number of values that are equal must be more that threshold number specified in the exception. Using compromise output with threshold voting does not make sense, but could be stated to specify a minimum number of ballots needed in the collation state for calculating an average.

Table 3.2 displays the inexact input voting classes. Inexact input in voting is covered in VDL with two exclusion operations, **distance-neighbor** and **distance-cluster**. These two operations allow a voter to create a sense of loose equality. Two values that are close, close being defined by a number in the operation, are considered equal.

Input Data	Output Data	Input Vote	Output Vote	Comments/Examples
exact	consensus	preset	threshold	**quorum** 5 **exclusion highest** 2 **collation mode** (with collation exception for threshold value)
exact	consensus	preset	plurality	**quorum** 5 **exclusion furthest** 1 **collation mode**
exact	consensus	adaptive	threshold	**quorum all but** 2 **exclusion furthest** 1 **collation mode** (with collation exception for threshold value)
exact	consensus	adaptive	plurality	**quorum** 50 **percent** **exclusion furthest** 2 **collation mode**
exact	compromise	preset	threshold	**quorum all but** 3 **exclusion lowest** 1 **collation mean** (with collation exception for threshold value)
exact	compromise	preset	plurality	**quorum** 5 **exclusion highest** 1 **collation mean-neighbor**
exact	compromise	adaptive	threshold	**quorum** 50 **percent** **exclusion furthest** 2 **collation median** (with collation exception for threshold value)
exact	compromise	adaptive	plurality	**quorum** 6 **exclusion furthest** 1 **collation random**

Table 3.1: VDL Coverage of Voting Algorithms Taxonomy, Part1

Input Data	Output Data	Input Vote	Output Vote	Comments/Examples
inexact	consensus	preset	threshold	**quorum** 70 **percent** **exclusion distance-cluster** 2 **collation mode** (with collation exception for threshold value)
inexact	consensus	preset	plurality	**quorum all but** 2 **exclusion distance-neighbor** 1 **collation mode**
inexact	consensus	adaptive	threshold	**quorum all but** 2 **exclusion distance-neighbor** 2 **collation mode** (with collation exception for threshold value)
inexact	consensus	adaptive	plurality	**quorum** 6 **exclusion distance-cluster** 2 **collation mode**
inexact	compromise	preset	threshold	**quorum** 6 **exclusion distance-cluster** 1 **collation mean** (with collation exception for threshold value)
inexact	compromise	preset	plurality	**quorum** 6 **exclusion distance-neighbor** 1 **collation mean-neighbor**
inexact	compromise	adaptive	threshold	**quorum** 60 **percent** **exclusion distance-neighbor** 2 **collation mean** (with collation exception for threshold value)
inexact	compromise	adaptive	plurality	**quorum** 60 **percent** **exclusion distance-neighbor** 2 **collation mean**

Table 3.2: VDL Coverage of Voting Algorithms Taxonomy, Part2

3.7 VVM API for non-VDL Languages

The VVM can be used with applications or voter managers where it is not feasable or desirable to use a separate voting language and compiler such as the VDL. While they do not directly specify the VDL policy to be used, voting algorithms in their essence provide a statistical tradeoff between correctness and performance. We have, therefore, created a means for such applications to still be able to control the VVM from higher-level specifications which are then mapped down to the VDL. Figure 3.5 gives the CORBA IDL describing this **module**. It first defines enumerated types for each **quorum** state,

```
//////////////////////////////////////////////////
// IDL for API for VVM primitives, for a non-VDL
// language to control VVM.
//////////////////////////////////////////////////

module VVM {
   enum quorum_op {Q_ALL_BUT, Q_PERCENT, Q_VALUE_BASED};
   enum exclusion_op {E_HIGH, E_LOW, E_FURTHEST, E_SIGMA,
                      E_DISTANCE_NEIGHBOR,
                      E_DISTANCE_CLUSTER, E_RANDOM};
   enum collation_op {C_MEDIAN, C_MEAN, C_MEAN_NEIGHBOR,
                      C_MODE, C_RANDOM};
   interface vvmCallback
   {
    void update_from_vvm(out long status);
   };
   interface voter_api
   {
    void set_voter_ops(in quorum_op q,
                       in exclusion_op e,
                       in collation_op c,
                       in vvmCallback cb);
    void get_voter_ops(out quorum_op q,
                       out exclusion_op e);
                       out collation_op c);
   };
}; //module VVM
```

Figure 3.5: VVM API

then an **interface** by which the user passes in a callback object to get notified of any crucial changes. The **interface** `voter_api` defines a method to set the VVM operations and to pass in a notification callback (a **null** callback reference passed here registers no callback for the caller). The current voter operations can be queried with `get_voter_ops`.

Of course, IDL maps into C++, Java, Ada, and other languages, so the IDL in Figure 3.5 *ipso facto* defines APIs for these languages, too.

The API above is, of course, a direct mapping of VVM states. Other APIs offering higher-level functionality are possible. In these cases, the voting manager must add additional heuristics to translate to the particular VVM operations in order to best implement the voting choice specified in the API call. One such API could allow the specification of the assumed failure mode, the maximum number of faults to be tolerated, and the initial number of replicas for the service (to initialize the voter manager with). Another API could offer simple choices such as "MAJORITY," "ALL," "CONSENSUS," etc. Yet another API might allow for the specification of the 4-dimensionsal voting criteria from the taxonomy in [39], which is discussed further in Section 3.6.1.

3.8 Voting Status Service

The voter core is tracking various kinds of information to implement its functionality. Some of this can be useful to other entities, so we have created the Voting Status Service (VSS) as part of the VVM architecture. The VSS is shown in Figure 3.6. It receives various voting statistics from the voter core. This includes the progress of each ballot through the core; e.g., did it arrive after the quorum has arrived, was it excluded, or was it chosen in collation? It also is informed if a replica sends a duplicate ballot or one which is early, and so forth. The VSS collects this information and maintains various kinds of moving averages that we call *status conditions.* Different entities can register to be informed when a given status condition crosses a threshold. For example, a performance management system may want to know if a given replica (or host or domain) is frequently late (its ballots are not making it past the quorum state). For example, a security management system may want to know if a given replica (or host or domain) is frequently giving bad data (its ballots are being excluded by non-random exclusion operators). The security management system subscriber might be an intrusion detection system. It may be able to use the VSS's information to concentrate its activities on certain machines. The VSS also provides an interactive query interface to allow users to query current conditions.

One application of the VSS is for group communication systems. Group communicat ion systems which provide virtual synchrony can suffer severe performance degrad ations when even a single group member is overloaded [48]. It is typically very difficult or impossible for such systems to exclude a slow member without knowing how the group is being used by the application or other information not generally available to the system. However, this is straightforward to do with the VSS: the group communication system gets an alert when replies are coming in sufficiently late compared to the others, over some recent moving average of votes. This alert can be based on the following conditions:

- a given replica in a given group is late "too often"
- any replica in a given group is late "too often"
- any replica on a given host is late "too often"
- too many replicas on any one host are late "too often"
- too many replicas in any one domain are late "too often"

where the "too often" is a moving average that the subscriber can parameterize. Given such an alert, then, the system can exclude the slow member to improve performance.

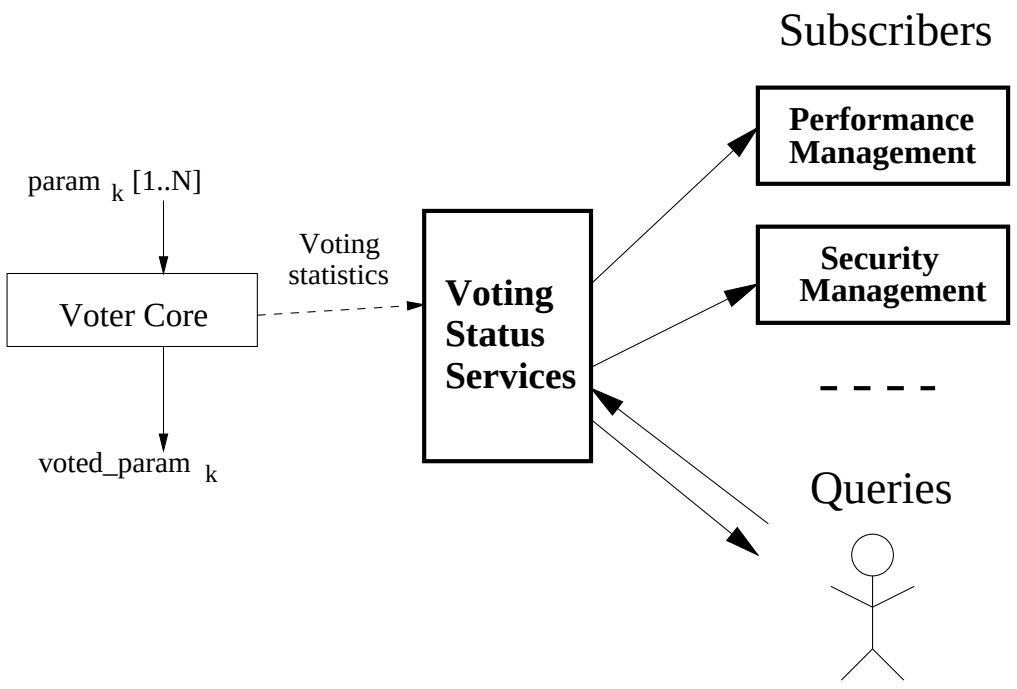

Figure 3.6: Voting Status Service

Chapter 4

Tolerating Faulty Values

The VDL policies described earlier provide means for detecting and filtering of faulty valued ballots. A value is faulty when it does not fall within a range of acceptable values. The range would be application or user defined. The faulty values could be sent by one or more replicas that have been subverted by an adversary. When faulty values are filtered before collation of the ballots, the collated answer is more accurate.

Table 4.1 provides symbols that are used throughout the rest of this thesis. The following sections describe each of the VVM states in terms of the security that they offer through various voting policies.

4.1 Quorum

Each vote starts in the quorum state. A VDL policy requires that the voter's quorum state wait for a certain number of ballots before continuing to the next state. There are two types of quorum defined for use within VVM, *ballot-based quorum* (BQ) and *value-based quorum* (VQ). BQ is required to wait for M ballots, whereas VQ is required to wait for L equal valued ballots. Table 4.2 summarizes the quorum state operations and the following sections discuss the security offered by BQ and VQ.

4.1.1 Ballot-based Quorum

In BQ there exist three operations for waiting on M ballots. The first is waiting for a given percentage of the membership, which is translated into M at run time, using the current size of the server group. The second operation is to explicitly specify M, which must be less than or equal to N, within the VDL policy. The third operation is '**all but *k***', where k is a given number of ballots and, therefore, M is equal to N - k ballots. These operations allow the voter to continue after only M of N replicas has responded. This is useful for replicated answers that vary in value, such as in the following cases:

- Multiple correct ballots are allowed to be returned by replicas.
- Floating point values may vary due to hardware and operating system differences.
- Value failures (either accidental or malicious)

BQ does not offer any security benefits as a stand alone operation, because it does not filter or detect faulty values entering the VVM system. As the value of M increases, the VVM system performance decreases due to increased latency in waiting for ballots.

Symbol	Description
N	number of replicas in system
B	number of faulty ballots
G	number of non-faulty ballots
L	number of equal values needed for value-based quorum
M	number of ballots needed to have a full quorum (B + G)
E	number of ballots excluded in the exclusion state
T	number of ballots sent to collation (M - E)
R	number of ballots to remove in random exclusion
D	distance from average value
V	voted value
H	set of values collected during the quorum state on the voter
J	set of values sent to collation state, a subset of H
J'	a different set of values sent to collation state, a subset of H

Table 4.1: Symbols for Chapters 4 and 5

Operations in Quorum
All but *k*
***M* ballots**
Percent of *N*
***L* equal values**

Table 4.2: Quorum Operation

4.1.2 Value-based Quorum

VQ is the second type of quorum considered within the VVM system. *Value-based quorum* must wait for *L* equal values before continuing. Once the quorum receives *L* equal valued ballots, then *M* ballots are sent to the exclusion state, where *M* is the number of ballots held in the quorum state at that time. This operation is effective only if the first *L* equal values are not generated from malicious and colluding replicas. The lower the value of *L*, the easier it is for an intruder because it requires fewer replicas to be corrupted. The greater *L*, the harder it is for a malicious user to corrupt the system, but the probability of receiving greater numbers of equal valued ballots may be quite small.

VQ does not give any real security benefits over BQ other then ensuring that *L* replicas are able to return the same value. If it can be assured that the most common value is a valid one and that *L*, the number of equal valued ballots, makes up 50 percent or more of *M*, then there may be greater benefit in using VQ. However, VQ may result in larger values of *M*, which allow malicious users the chance to pass more faulty values into quorum. The more faulty ballots entering quorum state, the more faulty ballots that are further passed into the exclusion state. Also, VQ results in dynamic quorum ballot set sizes. Therefore, an exclusion state operation that incorporates the size of the ballot set should be used.

Once the quorum state finishes, the exclusion state that is described in the next section is entered.

Operations in Exclusion
Random
High
Low
Furthest
Distance

Table 4.3: Exclusion Operation

4.2 Exclusion

The purpose of the exclusion state is to exclude E of the M ballots that were collected in the quorum state. Five operations for removing values within the exclusion state are described in this section and listed in Table 4.3. The security aspects of each operation are now considered.

4.2.1 Random

The first operation is that of selecting E random ballots and removing them from the collected set. The **random** operation offers no guarantee of filtering malicious values. Random removal of faulty ballots is an example of sampling without replacement, therefore the hyper-geometric probability function best models this operation. This function accepts three variables: number of samples (E), number of faulty ballots (B), and number of total ballots (M). The hyper-geometric probability function gives us the probability that x bad values are removed for a given E and B and M.

The **random** exclusion operation offers no real consistent security benefits to the collation state, because there is no guarantee of removing faulty values leading to a higher probability of faulty values passing into the collation state. However, using the **random** exclusion operation offers a *lack-of-pattern* property to the exclusion state and hence the output of the collation. An entity maintains lack-of-pattern characteristics when its actions or outputs are unpredictable or cannot be consistently or accurately determined. This is useful in thwarting any kind of attacks on the voter that exploit predictable patterns within the voting policy (see Section 5.1.2).

4.2.2 High and Low

The second and third operations are the removal of high and low values. The **high** and **low** operations exclude extreme values, which may thwart malicious strategies designed to corrupt the mean or median of the values. Extreme values are those values at the end points of the list of sorted values and are often a great distance away from the non-faulty values. **High** and **low** are extreme value exclusion operations. The greater the number of extreme values removed, the more accurate the average collation result. The fewer the number of extreme values removed, the greater the chance faulty values bypass the exclusion state and corrupt an average collation operation. In the worse case, the **high** and **low** operations' probability of removing x bad values is no worse than random. The **high** and **low** operations remove those values that may be outside a valid range, the range of the values that are non-faulty. The adversary will inject values that are outside the majority range to corrupt mean or with enough bad values to corrupt the mean-neighbor (2 + E) or median.

Operations in Collation
Mean
Mean-neighbor
Median
Mode
Random

Table 4.4: Collation Operation

4.2.3 Furthest

A combination of the **high** and **low** operations makes up the fourth exclusion operation, which is the removal of the furthest values from the average. The furthest values are those having the greatest distance from the median. The removal of the furthest values is another operation for removing extreme values, and, as the **high** and **low** operations, maintains the same advantages and disadvantages. Within the VDL policy, the specific number of furthest values to be excluded is defined. The further out the voter goes, the fewer values it is removing and visa versa; the closer the voter goes the more values it is removing. In the worse case, the probability that the **furthest** operation removes x faulty values is the same as the random operation.

The extreme value exclusion operations - **high, low, furthest** - gives a more centered set of ballots to the collation operations. These exclusion operations may thwart an attacker's injected ballots from reaching the collation state, assuming that the injected ballots are extreme. Again, if an adversary wants to corrupt the voted answer, then they may try injecting values that are extreme. This may be effective, depending on the collation operations. This will be discussed in Section 5.1.2.

4.2.4 Distance

Finally, the fifth exclusion operation is the removal of values that are D distance from the average value. For this operation, the value **D** must be specified within the VDL policy. In the worst case, the *distance exclusion operation* is no worse than random. The smaller the value of D, the greater the chance that extreme values are excluded from the collation state.

We call the distance operation a *distance-based exclusion* operation. *Distance-based exclusion* differs with extreme exclusion, in that it is value-based; i.e., the distance is given by the user and not influenced by the number of ballots. In an example with $H = \{3, 3, 5, 6, 6, 100, 200, 200\}$ and a policy like **quorum** 7 **exclusion distance** 2 **collation mean**, the median in H is 6, and so the set J is $\{5, 6, 6\}$ and the mean answer is 6. This is opposed to a policy with **exclusion furthest** 2 that gives J as $\{5, 6, 6, 100\}$ and a mean answer of 29.

After the exclusion state, the remaining T ballots are sent to the collation state.

4.3 Collation

The collation state of the voter determines a single answer by one of five operations that was described in Section 3.3.3. The operations are listed in Table 4.4 . Each collation operation will be briefly summarizes and the level of security provided by the operations is explained.

4.3.1 Mean

The **mean** operation provides an average answer, but, for the most accurate answer, requires that all *T* values reaching the collation state to be non-faulty. This operation can be greatly affected by any faulty or extreme value that may pass through the exclusion state. For example, if $J = \{4, 3000, 3, 3\}$, then the mean value is 752. However, if the extreme value of 3000 had been removed during exclusion state, then a mean value of 3 would have been calculated, which is a much better representation of the ballots' values.

The **mean** collation operation is greatly improved by the exclusion of the extreme values through either the **high, low, or furthest** exclusion operations. The more extreme the value, the greater the effect it has on the mean. Example: $H = \{9, 4, 4, 5, 4, 100\}$, with policy **quorum** 6 **exclusion furthest** 1 **collation mean** will produce $J = \{4, 4, 5, 9\}$ and a mean answer of 5. The mean without the exclusion is 21. The mean answer of 5 is a better representation of the values than the mean value of 21.

A **mean** collation with *distance-based exclusion* gives no worse an answer than the *extreme exclusion* with **mean** collation. The *distance-based exclusion* at the least excludes the same values as the *extreme exclusion*, but does it from a value standpoint. *Distance-based exclusion* may be better if it can give a tighter exclusion (filtering more extreme values). The following example shows that Example: $H = \{100, 4, 4, 5, 4, 2, 100\}$ and a policy of **quorum** 8 **exclusion distance** 1 **collation mean** will generate a collation set $J = \{4, 4, 4, 5\}$. A policy with **exclusion furthest** 1 will generate $J' = \{4, 4, 4, 5, 100\}$. The mean of *J* is a better answer than that of *J'*.

4.3.2 Mean-Neighbor

Mean-neighbor is very similar to **mean**, but requiring T - 1, rather than *T*, non-faulty values for a valid answer. A mean value is generated and the value closest to the calculated mean is chosen as the answer. This allows the voter to return a value that is an actual value received by the voter, which is required in some applications. Using the example above, the mean-neighbor operation returns the closest value to 752, which is 4. This is obviously a much better result as compared to the mean operation.

The **mean-neighbor** operation works well if *M* is greater than two with at most one faulty value ($B = 1$). In order for a malicious user to corrupt the **mean-neighbor**, two or more extreme values in the collation state ($B = 2$) are necessary. For example, in the collation ballot, $J = \{4, 3000, 3, 3\}$, the value 3000 does not affect the answer. In a ballot collation of $J = \{4, 3000, 700, 3\}$, the mean is 926.75 and the neighboring value is 700. In order for the two bad values to corrupt, the second value must be closer to the average than the highest valued non-faulty value. Looking at the set $\{k_1, k_2, ..., k_{n-1}, k_n, x, N\}$, where *N* is a maximum possible value and *x* is the second faulty value and k_n is the highest valid value, then *x* needs to be closer to average than k_n. Therefore, the bounding of *x* is such: ($k_n < x <$ mean + (mean - k_n)). To corrupt the **mean-neighbor** operation, a malicious user must corrupt at least two servers. This doubles the adversary's work and chances of being detected by an intrusion detection systems(IDS).

The **mean-neighbor** operation combined with extreme exclusion (i.e. **high, low, or furthest**) offers a more accurate value due to the filtering of the most extreme values, which greatly affect the mean. The following example will show the effect of the exclusion of extreme values on the **mean-neighbor** Example: $H = \{45, 4, 4, 5, 4, 100\}$, with a policy of **quorum** 6 **exclusion furthest** 1 **collation mean-neighbor** produces $J = \{4, 4, 5, 45\}$. The mean-neighbor of the set *H* is 45 since the mean of *H* is 27 and 45 is a closer value than 5, while the mean-neighbor of the set $J = 5$ because the mean is 14 and the closest value is 5. Example: $H = \{100, 4, 4, 5, 4, 100\}$ and $J = \{4, 4, 5, 100\}$, the mean-neighbor answer is 5. Mean-neighbor handles a single fault, but, with the furthest exclusion, the voter as a whole can now handle two faulty values. Depending on the number of exclusions and number of faulty values, the tolerance level can be higher.

A *distance-based exclusion* in some cases can provide a better subset of ballots for the **mean-neighbor** collation operation. If both faulty values in the mean-neighbor attack are not in the distance of D from the median, then the *distance-based exclusion* will eliminate it and the threat of its subverting the mean-neighbor value. Example: $H = \{100, 4, 4, 30, 5, 100, 3\}$ and a policy of **quorum** 7 **exclusion distance** 1 **collation mean-neighbor** produces $J = \{4, 4, 5, 30\}$ and an answer of 5. *J'* using **exclusion furthest** 1 is $\{4, 4, 5, 30, 100\}$, which will corrupt the mean-neighbor with a bad value of 30.

4.3.3 Median

The **median** operation results in a valid answer only if G, the number of non-faulty ballots, is greater or equal to the $\lceil T/2 \rceil$. The **median** operation sorts the values and selects the middle value. For a malicious user to corrupt the median, enough faulty values must reach the collation state that the middle value is a faulty value. Take the example of $J = \{4, 3000, 5, 3, 3\}$; the median is 4. If a second value were to become corrupt so that the set is $J = \{4, 3000, 3000, 3, 3\}$, then a median of 4 is still the result. With a set such as $J = \{4, 3000, 3000, 3, 3000\}$, however, $G = 2$, which is not greater or equal to the $\lceil T/2) \rceil = 3$, and the median value is corrupted.

Extreme exclusion provides a better set of values to the **median** collation operation. The median is not as susceptible to extreme values as mean, but faulty extreme values that are a result of collusion may cause the median's value to be shifted up or down. Example: $H = \{100, 4, 4, 100, 100, 100\}$, with **high** exclusion excluding 2, will produce $J = \{4, 4, 100, 100\}$. The median of the set H is 100, while the median of the set $J = 4$. A set of values that had a number of faulty values over the median level of the ceiling of $T/2$ can be corrected with the use of the correct exclusion operation to remove some of the faulty values and reduce the number of faulty values to under the $\lceil T/2 \rceil$.

The median answer can be greatly affected by the values excluded and in the example below, the *distance-based exclusion* gives a better answer than that of *value-based exclusion.* Example: $H = \{100, 4, 4, 5, 3, 100, 100\}$ with a policy of **quorum** 7 **exclusion distance** 1 **collation median** will produce $J = \{3, 4, 4, 5\}$ and a median answer of 4. *J'* excluding the lowest 2 values gives $J' = \{4, 5, 100, 100, 100\}$ and a median answer of 100.

4.3.4 Mode

The **mode** operation selects the most common value and is valid only if the number of equal, faulty values is less than the number of equal, valid values. From the example above, $J = \{4, 3000, 3, 3\}$, the **mode** operation results in an answer of 3. The mode can be corrupted with a collation set such as $J = \{4, 3000, 3, 3000\}$ in which the **median** operation has a chance of selecting 4 if rounding down. The number of faulty values handled by mode depends on the number of corrupt servers and correct servers that return equal or closely-equal values. As long as the number of correct servers returning equal values is greater than the number of faulty servers that are sending the same bad value, then the voter will be able to choose the correct answer. Also, the value-based quorum will give the exclusion state a set of equal values that, barring the exclusion of none, will give the collation state a good mode value. This assumes the most common value in the quorum is not a faulty value.

The *extreme exclusion* operations offer no better filtering to **mode** than random exclusion offers. However, if the colluding replicas return the same value and are deemed extreme by the exclusion operation and removed, then this strategy of *extreme exclusion* and **mode** collation works well. Example: $H = \{100, 4, 100, 4, 5, 100\}$, excluding two highest values, produces $J = \{4, 100, 4, 5\}$ with a mode answer of 4.

Again, the type of exclusion may affect the collation operation. The example below shows such an example, where the *value-based exclusion* gives a mode answer that is not as good as the *distance-based*

exclusion. Example: $H = \{100, 2, 4, 5, 100, 100, 3, 6\}$ with a policy **quorum** 8 **exclusion distance** 1 **collation mode** produces $J = \{3, 3, 4, 5, 6\}$ and a mode value of 3. $J' = \{100, 3, 4, 5, 6, 100\}$ when a **furthest** 1 exclusion is use and provides the mode answer 100. The answer that better represents the non-faulty values is the distance-based exclusion policy.

4.3.5 Random

The last operation is the random operation, which provides no guarantee of selecting a valid value. Random collation combined with random exclusion offers the greatest lack-of-pattern defense against an adversary inferring the policy. Extreme exclusion and random collation provide a good outcome. The values that are considered dangerous, the extreme values, are removed, and then the random collation operation selects one. This provides good filtering with randomness used to include the lack-of-pattern property to a policy. Example: $H = \{100, 4, 4, 5, 4, 2\}$, with a policy **quorum** 6 **exclusion furthest** 1 **collation random** produces $J = \{4, 4, 5, 4\}$ and then the random collation picks one from J.

Random collation with *distance-based exclusion* can also provides good answers. As we have seen from the examples in this section, the *distance-based exclusion* can sometimes provide a better set of values for collation. *Distance-based exclusion* can in some cases provide the random collation with a better set of values from which to choose the answer.

4.4 Weighted Voting

Weights in the collation state can act as a filter. A values affect on the collated answer can be regulated by the weight assigned to it. A value with a lower weight than the other values has less of an impact on the answer than the other values. The following example shows that, with weighting, a faulty values effect on the **mean** collation can be decreased. Suppose $J = \{1000, 5, 4, 6, 7, 6, 2\}$ with all values having a weight of one and the value 1000 is considered the faulty value. The **mean** collation answer is 170. If we give the following weights of $\{1,2,1,3,1,1\}$ to the value, then $H = \{1000, 5, 5, 4, 6, 6, 6, 7, 2\}$, and the answer will be 115. If we give the following weights of $\{1,4,4,4,4,4\}$, then the answer is 55. As the weight of the faulty value is lesser and lesser than the other non-faulty values, its decreased impact is evident as the voted answer gets closer to the non-faulty values. Another possibility is that if a known replica is faulty, but it is kept active for reasons of monitoring the malicious user, then the weight of the replica could be set to zero. This gives the replica's values no chance of corrupting voted answer.

The next chapter discusses the possibility of a malicious user attack against the VVM system and the level of security that VVM maintains against such attacks.

Chapter 5

Resilience Against Malicious Attacks

5.1 Inferring Voting Policy

The last chapter described the difficulty for an adversary to corrupt a vote. Determining the voter's policy gives the adversary a better chance of corrupting a vote, because the adversary will know what values pass exclusion and affect the voted answer. In this section, we explore the possibility of an adversary determining a voter's policy by observing incoming ballots and the outgoing voted answer. The VVM and the VDL policies have many mechanisms that can be used to impede or thwart adversaries from determining the policy. These mechanisms include randomness, weighted voting, and policy modulation. These mechanisms of the VVM and VDL policies are described below.

5.1.1 Assumptions

First, we assume that the adversary has the access and means to observe the voter's input and output traffic undetected, without disrupting the voter. The adversary must be able to read the incoming ballots and outgoing answers to learn the values in each. We assume that the control command messages from the VVM manager are encrypted and, therefore, cannot be read. These vital control messages contain the changes to the voter policy, group membership, and replica weights. In the worse case, the adversary will be on the same local network as the voter; therefore, we assume the adversary will know the order of ballots received by the voter. The adversary only knows which ballots the voter received; therefore, we assume that the adversary does not know which ballots passed the quorum. For simplicity, we assume the vote is on only one parameter. We also assume that the adversary has some knowledge of the application using the voter and on which parameter the voter is voting. The final assumption is that the adversary has not subverted any servers. Some of these assumption will be relaxed later in this chapter.

An adversary might start by looking for any patterns in the incoming ballots and the corresponding voted answers that may be able to divulge the collation operation. Some of these patterns are described below. If a pattern is not easily detected, then the user could start a brute force approach by selecting one collation operation at a time, testing with different exclusion operations using different values of E. This approach requires that the adversary test for all possible combinations of N ballots received; therefore, there would be 2^N different combinations for each of the 30 combination of exclusion and collation operations. For *distance-based exclusion*, the adversary may sort the ballots and look at the median value in comparison with the other values, trying different distances. The *value-based exclusion* requires the adversary to iterate through the different operations trying different values for E. The starting point for an adversary is determining the collation operation.

5.1.2 Inference of Collation Operations

An easily determined policy is one employing the **mean** collation operation. The **mean** collation returns a new value that may not match any of the ballots, so the user will see that the selected value, in the outgoing message, is not one of the incoming ballot's values. No matter which quorum type or exclusion type, the distinct output value will tell the adversary that the collation operation is mean. The **mean** operation may not always produce a new value. Therefore, the adversary needs to view more than one vote in order to determine if there are distinct values for multiple voted replies before prematurely disregarding the mean and starting their to focus on **mean-neighbor**, **mode**, or **median** collation operations. In order for the adversary to determine which ballots were used in a vote, they need to try all, 2^N, combinations of ballots with a mean calculation in the worst case. However, there may be multiple subsets of the received ballots that produce a mean equal to the voted answer. The adversary will need to examine multiple votes to determine which ballots are not used in the collation. This process allows the adversary to determine the ballots that were either excluded by the exclusion operation or that did not make it through quorum.

Adding random 'salt' to the mean value masks the true subset of ballots used. This will confuse the adversary if no combinations give the salted value, or may lead the adversary to determine that a certain subset of ballots is correct when, in fact, another subset is the true subset.

The rest of the collation operations are more difficult to discover, because their answer is one of the values from the set of ballots. Therefore, the adversary, to determine the collation operation in use, must try the different collation operations with different subsets of the ballots. The adversary might start by comparing the voted answers against the ballot sets to see if the most common ballot value is the voted answer.

Determining if a policy has a **mode** collation operation requires that the adversary compare the most common ballot value and the voted answer for many votes. Sometimes, the exclusion of the most common value will cause the **mode**'s answer to be different. An example would have a set $H = \{3,3,8,6,3,6,5,9,10\}$ in which the exclusion operation removes $\{3,3,9,10\}$, the two most extreme values, and leave the set $\{3,5,6,6,8\}$. A mode answer of 6 would be the voted answer, which is not the most-common value from the received set. With a wide variety of voting sequences with different values, an adversary should be able to determine if the most common or next most common value is chosen as the voted reply's value. If the second or third most common value is chosen, then one or more of the first or second most common values must have been excluded during the exclusion state, or did not pass the quorum state. If the above process does not produce any results, then the adversary may try either of the next two operations.

Determining if the policy employs a **mean-neighbor** collation requires a little more work than **mode**. With the brute force process of trying different exclusions, calculating the **mean**, and comparing closest neighboring value and the output message's value, an adversary could determine that the policy is a **mean-neighbor** collation, and the exclusion operation in use. As with **mean**, the **mean-neighbor**'s voted answer can be calculated with different subsets of the ballot set, and thus the adversary needs more than one vote to determine the quorum and exclusion operations. If the **mean-neighbor** values and the voted answers do not match for all votes tested, the adversary could try the **median** collation operation.

An adversary can determine if the collation operation is the median operation by performing a sort on the ballots and performing the brute force approach of trying all the possible combination of ballots with each exclusion state. If none of the combinations work for the median collation, then the adversary has failed to determine the policy. Either the adversary has overlooked a pattern, or has not completed enough analysis of the data, or the voter's policy is random collation.

If the adversary has done all work above and not found an answer, then the lack of a pattern may prompt the adversary to determine that the voter is using some kind of random selection in the collation. The random collation operation removes any patterns that correlate the ballots and voted answer.

5.1.3 Inference with Multiple Votes

Along with random collation, random exclusion offers the voter a lack-of-pattern property with all the collation operations. Random exclusion may confuse the adversary when trying to determine the collation operation. With enough data, and depending on the number of values randomly excluded and the number to exclude from, a user should still be able to find the patterns or brute force a conclusion about the collation operation.

Analyzing a single vote may not give the user a correct policy. Therefore, more data — incoming and outgoing messages — an adversary has, the easier the adversary can find patterns in the incoming ballots and voted replies and determine the policy. Analyzing a single vote may not give the user a correct policy. The following example shows how a single vote cannot give the correct policy, because different policies can get the same voted answer. Take, for example, the set $H = \{3,5,3,1,2,4\}$. With exclusion of the furthest values $\{1,5\}$, the mode answer is 3, the mean answer is 3, and the median answer is 3. If the exclusion is *distance-based*, then the median before is 3 and with a distance, D, of 1, the collation operations give the same answers as given above for the *value-based exclusion* operations. If the user collects another set of ballots, $H = \{9,15,9,9,16\}$, and the voted answer is 9, then the user may guess that the collation operation is **mode** or **median** if the exclusion is again removing the furthest values. The mean answer is 11, which excludes the mean collation as a possible operation. A third sample with $H = \{3,9,4,3,10,6,2\}$ with a voted answer of 4. With the exclusion of the furthest value, $\{2,10\}$, the median is 4. The mode answer is 3 and the mean answer is 5, and these values exclude the two operations from being the correct one. From these three samples, an adversary could guess that the collation operation is median and that the extreme values are excluded. With more samples the adversary can test their guesses and possibly determine the exclusion operation.

5.1.4 Corrupting Servers to Aid Inference

We now relax the constraints a little and assume an adversary can subvert one or more of the servers. If the adversary can subvert one of the servers, then the adversary may be able to inject test values to help determine the voter's policy, much like the classic plaintext attack. The adversary could have the subverted server inject values and see if it affects the expected output of the voter. With more than one server subverted, the user could inject many values, helping to bypass the exclusion operations. A smart adversary will not use the subverted servers at all, or very little, in determining the policy, to reduce the chance of being detected by the voter status services and IDS.

The injection of ballots by trapping the original ballots and replacing the message with different values is not easy, if even possible. On a wide area network like the Internet, the adversary would need to subvert a router and change the routing table so that the packets destined for the machine with the voter are sent to a subverted computer, where the values can be modified and then forwarded to the voter. On a local area network like Ethernet, the broadcast of a message cannot be trapped on a computer unless the computer is a gateway. Then the adversary would need to subvert the gateway or install a gateway into the network. Doing either of the previous methods requires a great deal of time and greater risk of detection. If possible through adding a gateway of sorts or corrupting a router, then the signing of the ballots with checksums will thwart this man-in-the-middle attack. While this requires a key infrastructure, it does not have the overhead of full message encryption.

5.1.5 Weighted Voting to Impede Inference

The assignment of the weights and changing of the weights are affected by the VSS and the voter's weight manager. The weights could be affected by messages from external systems or sensors. These systems

may have the ability to detect intrusion or malicious activities and advise the voter. The voter can decide whether to change the weights of replicas or dismiss the warning. A human manager of the voter can be notified if thresholds are crossed or intrusions from IDS are received.

Weighted voting causes the adversary more work in determining the voter's policy. The weights of the voter are part of the voter and command messages, and so the adversary would not know the weights. Some of the patterns will still exist. The mean could still generate a new value, but you may not know which ballots were excluded, since any of the ballots could be duplicated many times. The number of possible combinations will become larger than 2^N, and with unlimited weight value (or the max of an integer value) the possible combinations could be infinite. This will greatly hamper any kind of brute force attack.

5.1.6 Value Quorum Usefulness

The VQ in the VVM can offer the voter assurance that some of the replicas are returning the same value. The number of equal values that the VQ waits for can be modified by a voter manager that may have received a warning from an external system, i.e. ID system. If a voter policy was waiting for three equal values in the quorum state, and then the voter's manager receives a warning about possible intrusions, the manager may increase the number of ballots from three to impede the effectiveness of the adversary's actions. This is changing the voter's policy at run time, which is what the next section discusses.

5.1.7 Policy Modulation

Another way for the voter to thwart an adversary from determining its policy is to modulate the policy. For each method, every vote, or every so many minutes the voter could swap policies. This causes the analysis of the incoming ballots and voted answers to be more complex, since the adversary will not know when the policy changes.

Most users of VVM may not have large security requirements. The combination of weighted voting and policy modulation thwarts an adversary from corrupting a vote, by knowing the voter's policy and being able the corrupt the voted answer. This provides a relatively safe voter without the complexity of encryption, and a key infrastructure needed to give a more secure vote.

5.1.8 More Problems for Adversary

In this section we will discuss the added complexity for the adversary if we further relax some of the assumptions we made at the top of this chapter. If we loosen the assumption that the adversary knows which parameter is voted on, then the brute force attack will need to be for each parameter. There may be instances when patterns of voting are detected in a parameter that is not being voted on. This may have the adversary thinking that a different parameter is the one being voted on, and either chose to corrupt it because it is critical parameter or avoid it and avoid possible detection by the voter.

We can also relax the notion that the adversary is close to the voter and thus knows the order of ballots received. The adversary may guess at how many ballots and which ballots do not pass quorum, and know that the ballots received last have a greater chance of not passing. In the voter, most of the quorum operation will take the first M ballots that were received. If the adversary does not know the order of the ballots, then it will be more difficult to determine which ballots passed quorum and which passed exclusion.

The attack on determining a voter's policy can be done with a brute force approach, but if the previous ideas of weights and policy modulation are used then the adversary's probability of determining the policy

is very small. The next section describes how, if possible, the adversary can use the determined policy to corrupt the voting.

5.2 Exploiting a Known Voting Policy

If an adversary is able to determine a voter's policy, then what can they do with that knowledge? This depends on how easy the policy can be subverted, which was discussed in Chapter 4. Also, it depends on whether the voter is using weighted voting and whether the adversary knows the weights. We will first examine the weight-less voter.

If the server's ballots are not weighted and the adversary knows the policy, then what can the adversary do to corrupt the voted answers? We will first assume that the adversary has subverted one server. Depending on the number of faults the voter's policy can handle, this may or may not be a problem. These were analyzed in Chapter 4 . If the policy's collation operation can handle no fault values, then the adversary needs to try to send a value that will be selected in the quorum state and not be removed by the exclusion state. This will allow the faulty ballot to corrupt the voted answer.

The higher the number of faults the policy can handle, the more servers the adversary needs to subvert. With the increase in the number of servers needed comes the increase in risk an adversary takes in being detected, even with intrusion detection systems that may only catch a fraction of intrusions. The more computers one attempts to gain control of, the greater the chance that the IDS will detect the malicious activities.

For an adversary to subvert a **mean**, the faulty value needs to be a number with a distance big enough to affect the mean value, but may not need to be too far away, depending on the application. For the **mean-neighbor**, an adversary needs two values in the collation state. The **median** requires more than half of the replicas to be subverted. The **mode** needs enough subverted replicas to make sure that the value they send is the most common. This could require up to half of the servers. These numbers are for the collation operations. The ballots from the subverted replicas need to get through the exclusion state. The adversary knows the exclusion and is able to know what values should pass to the collation state. It may require some subverted replicas' ballots to be sacrificed so that other subverted replicas' ballots may pass to the collation state.

If the voter is weighting the ballots and the adversary does not know the weights but has determined the policy, then the adversary could guess the weights. The adversary may need to try the different combinations of weights with values and will probably find that many schemes would get the voted answer. Doing this across many different votes may get the adversary the weights.

If the adversary knows the weights, then the adversary will know which replicas have more importance in the collation. This may help the adversary chose which replicas to subvert and, also, which replicas may have more security. The higher the weight for a replica, the higher the voter thinks it can trust that replica, because it is supposed to be secured. The adversary needs to consider the trade-off between risk of detection and ease of subversion against the power that a higher weighted replica has on the voted answer.

Chapter 6

Discussion

6.1 Implementation Status

We have implemented many of the pieces of the VVM. The voter core has been completed with most of the state operations incorporated. The incorporated operations include the **absolute, all but *k*, *M* ballots, and percent of *N*** quorum operations. The completed exclusion operations are **random, high, low, furthest, and distance**, with the exception of the sigma and epsilon distance operations. **Mean, median, mode, and random** are the working collation state operations. Random and weighted voting is implemented along with a mock membership manager that handles the replicas' quorum and collation weights. An initial VSS is implemented that can collect and output the data for each ballot received by voter. The unmarshal and marshal components that handle our test protocol messages are done. The voter's policy manager is a GUI interface that allows the user to set a policy for the voter. The voter core has been tested with different policies for verification that each state in the core is acting according to what the policies dictate.

6.2 Implementation Issues

6.2.1 CORBA Integration Issues

We have recently begun a task to provide an experimental integration of the VVM with CORBA. Part of this involves providing the unmarshal and marshal modules for CORBA. Toward that end, tools (`idl2vvm` and `ir2vvm`) are being developed to allow the developer to compile interfaces that are likely to be used, as discussed in Section 3.5, to achieve better performance. We are also working on a runtime lookup to the CORBA Interface Repository to allow VVM's use with interfaces that were not compiled at runtime.

Presentation-layer issues are only part of the problem for integrating into CORBA. It is still necessary to insert the VVM into the client-server request-reply path. Our CORBA integration effort is using the gateway approach [49], which was developed to separately integrate quality of service (QoS) properties such as replication and bandwidth management on the QuO projects [46].

The gateway approach works as follows. A normal CORBA client obtains an object reference from its ORB by calling `bind()`, which contacts a naming service for it. The ORB creates a proxy (stub) object and then returns to the client a pointer or reference (depending on the language) for that proxy; this is what the client considers the object reference. After this, the client calls the proxy object just like it would a local object. The gateway approach modifies this by having the client not call the ORB's `bind()`, but rather a similar routine we provide, `connect()`. This "forges" a stringified object reference that is set

to "point" to the gateway (correct IP address and port, for example). Method `connect()` the calls the client-side ORB's string_to_object() method, which creates a proxy for the client, a pointer or reference to which is returned to the client, as the return value from `connect()`. The client can now make invocations on this object reference just as it would one returned by the ORB's bind(), and the IIOP messages emitted by the ORB are delivered to the gateway.

The VVM is not restricted to using only the gateway approach, however; it is in practice very flexible as to where it can be inserted. There only needs to be passed a marshaled message or a vector of application-level parameters; i.e., $iiop\_msg[1..N]$ or $param_k$. For example, it would be possible to use CORBA interceptors, which provide the system builder the ability to insert code at predefined locations in the ORB, either after or before the linearized request buffer has been created. It should also be straightforward to integrate the VVM into a service-based CORBA replication scheme. Indeed, it would even be possible to create a tool, taking as inputs the IDL and VDL, to generate a voting algorithm in a smart proxy (or smart stubs), an optional layer containing user-level code above the ORB's proxy, but below the client. However, getting multiple replies into the same smart proxy (or even interceptor) would be problematic with some ORBs, and would depend on how they implemented suppression of duplicate messages.

6.2.2 Group Changes During a Vote

One issue that must be dealt with when integrating the VVM in a real setting is the possibility of a group membership change happening while a vote is in progress. There are two situations to handle regarding this. The first is in regard to notification that a given member of the server group has failed. In this case, with most group communication systems, it is impossible for a ballot to be sent to the VVM from that group member; the group communication system will discard any messages after having declared the member to have failed. In this case, upon receipt of a failure notification, it is safe for the VVM to reduce the number of ballots required to achieve the quorum (e.g., to meet 50%), if the voter is in the quorum state. However, if the vote is past the quorum state and a ballot was already received (before the failure notification), that ballot can be used as it normally would.

The second case regards notification that there is a new member of the server group. In this case, the voter needs to be informed on which invocation sequence number the new member is expected to first vote. This has to be coordinated with state transfer to the new member, and, in fact, this sequence number can be considered part of the member's state, through state that is not installed at the application layer, but is given to the VVM. In a worst case, this may involve minor extensions to the group communication system. However, if the voter is not involved with asynchronous invocations, we believe it will be possible to infer to which sequence number the new member will begin sending ballots. It may be possible, in some systems, to not know or infer the first sequence number to a new member will send, but, rather, just to wait for the first ballot from that member to arrive. (In this case, the assumed membership for the dynamic quorum calculations may be wrong; however, it seems very unlikely that an application's correctness would depend on this.)

6.2.3 Integrating with a CORBA Fault Tolerance Implementation

The Object Management Group (the OMG) is standardizing replication for CORBA. The submissions so far have included both interception-based and service- based approaches to providing replication [10]. We believe that, whichever approach is adopted, it will be significantly easier to integrate the VVM into an implementation of the standard, rather than "from scratch" as the discussions in the previous two subsections assume (adding in a group communication system to an ORB which is unaware of replication or the VVM). This is because group communication and state transfer will already be in place, something

like a sequence number will already be in place, and there will likely be (judging by the submissions to date) a number of well-defined places in the architecture in which the VVM could readily be inserted, etc.

Indeed, the CORBA fault tolerance proposal has very recently been updated [50]. It includes a placeholder replication style `ACTIVE_WITH_VOTING`, which it notes is not supported in the current specification "but is an anticipated extension." We hope to be involved in the development of this placeholder and possibly to incorporate a subset of the VVM to this placeholder.

Chapter 7

Conclusions

7.1 Summary of Thesis

The Voting Virtual Machine (VVM) captures the notion of voting at a high level of abstraction, generality, and portability. When added to appropriate middleware technology, for example CORBA and BBN's QuO, the VVM can provide a powerful, well-specified, flexible, and adaptable tool for transmitting correct requests and replies between clients and servers, with wide-ranging applications including active replication, security, and N-version programming. Furthermore, the ability of VVM to monitor and report trends in failure modes has potential applications in performance, dependability, and security.

Chapter 2 described the architecture of the Voting Virtual Machine, discussing the primitive and advanced operations and how it fits into current middleware systems. The design of the Voting Description Language in Chapter 3 gives the voting machine great adaptability and is the first language of its kind that allows for implementation of a wide number of voting algorithms. Chapter 4 discusses the design of the Voting Status Service and its benefits as a performance and security monitor. The design has been extended by others to increase its abilities and uses for performance analysis and fault detection. Chapter 5 discusses the VVM's tolerance to faulty values and the effectiveness of the voter in removing those values. The resilience of the voter to malicious user attempts to learn the policy and, if the policy is learned, the ways in which VDL policies make it difficult for a malicious user to corrupt a vote was described in Chapter 6.

The research for this thesis has dealt with the development of the Voting Virtual Machine and the security aspects of parameter-based attacks to distributed systems. This thesis has answered some questions about the ways in which a voter can be used to thwart or impede a malicious user from performing a parameter-based attack. From this thesis, further work has been defined.

7.2 Future Work

7.2.1 Voter Management

The VVM allows the development and deployment of a wide variety of voting algorithms. This can, in many ways, help applications and systems to be more adaptable, scalable, and robust. However, only a small fraction of applications can realistically be modified to choose their voting algorithm. Further, even if they wanted to, most (arguably almost all) application programmers would not be capable of choosing the best voting algorithm, let alone knowing when to switch between them as conditions changed.

To overcome these shortcomings, we plan on developing *voting managers* to chose an appropriate

voting strategy for applications, and to switch strategies when conditions change sufficiently. These managers will provide *voting transparency*, allowing applications already employing replication transparency to benefit from voting without being aware of its existence.

A voting manager will take inputs from a number of sources, including

- Status information from a network management service giving network utilization, capacity, etc.
- Security management status information regarding which failure model may be most appropriate.
- Intrusion detection alerts warning about hosts or domains that may have been compromised.
- QoS requirements or preferences from a local QoS contract, using the Quality Objects (QuO) framework [49, 46]. These requirements or preferences can provide the application's desired levels of service of and/or tradeoffs between different QoS attributes such as performance, availability, security, and correctness. The QoS contract can be provided by the application. Alternately, if *QoS transparency* is desired, the contract can be provided and updated by an external application management system.

The voting manager uses these inputs to choose an appropriate voting algorithm from its repertoire based on offline analysis as described in Section 3.6.1. Clearly, the selection of an optimal algorithm in an arbitrary situation is an intractable problem. However, we will develop heuristics to help the manager choose as close to optimal an algorithm as possible in a wide variety of situations.

7.2.2 Multiple Parameter Collation

The voting algorithms from the literature mentioned above have all been developed to vote on a single data value. We will explore the voting on multiple data values and extend VDL to handle this situation. For example, this will enable voting on multiple **in** or **inout** parameters in a replicated client request, or on multiple **out**, **inout**, or return value in replicated server replies. However, because voting on multiple parameters has not been explored before, this opens up many research issues, which we will investigate.

We will investigate the possible ways for combining different vectors of parameters. For example, we presume that most applications will require that the voted message contain a vector of values from one of the reply messages generated by a server. To do this, one could rank each parameter in a reply by comparing it to the others, then take the reply whose parameter ranks totaled the most.

This project will investigate the ways to specify the above rankings as well as the kinds of applications which may use the different way of collating based on multiple parameters. We will also develop *voting patterns*, which allows VDL policies to be used for a wide variety of method signatures.

7.2.3 Confidence Value

We could easily pass along a *confidence return value* with a vote, a number indicating how good the outcome was. For example, the value may depend on: the number of values which were equal to (for integers) or "close to" (for floats) the other values, or the arrival distribution (jitter or standard deviation) of the messages. This would allow the client to also decide how to use a reply based on how good it is perceived to be. The confidence value could be specified in VDL. Alternately, it may be best to have a separate *confidence description language*, so that pieces of code (confidence policies) could be written by a different person from the VDL programmer and also could be reused with different VDL policies, to the extent to which that is possible. The exact utility of such a value, and how to specify and measure it, is open research.

7.2.4 VDL Extensions

This project will also extend VDL in a number of other ways to make voting more flexible and useful. For example, exceptions are widely regarded as a useful language feature in fault-tolerant systems[51, 52], and are supported in most modern programming languages and middleware systems. We will thus implement user-definable exceptions for VDL, to allow the voter to throw an exception for any of a number of reasons, rather than being forced to return a value. We will also support the voting on replicated exceptions (exceptions can have data in them, after all!). We will also support compound specifications in the quorum and exclusion states, for example, to allow the exclusion of the two lowest values, and then those outside 1.5 sigma of the mean value. Finally, we will investigate the supporting of branching between VVM states; e.g., to go back to quorum if exclusion discards too many values.

Appendix A

Publications

- Bakken, David E. and Karr, David A. and Jones, Christopher C. and Hale, John C., *"The Voting Virtual Machine: A Flexible Mechanism for Collating Replicated Client Requests and Server Replies"*, Digest of Fast Abstracts presented at the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS-29), FTCS-29, IEEE, June 15-18, 1999, Madison, WI.
- SYSTEMS AND METHODS FOR VOTING ON MULTIPLE MESSAGES, US patent application (joint WSU and BBN), David E. Bakken, Christoper C. Jones, and David A. Karr, 2000
- Bakken, David E. and Jones, Christopher C. and Karr, David A., *"The Voting Virtual Machine: A Flexible Mechanism for Collating Replicated Client Requests and Server Replies"*, to be submitted for publication in Summer, 2000.

Bibliography

[1] F. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, December 1990.

[2] M. Chereque, D. Powell, P. Reynier, J. Richier, and J. Voiron, "Active replication in Delta-4," in *Proceedings of the Twenty-Second International Symposium on Fault-Tolerant Computing*, pp. 28–37, 1992.

[3] D. Malki, *The Encyclopedia of Distributed Computing*, ch. Quorum Systems. Kluwer Academic Publishers, 1999.

[4] R. Gusella and S. Zatti, "The accuracy of clock synchronization achieved by TEMPO in Berkeley UNIX 4.3BSD," *IEEE Transactions on Software Engineering*, vol. 15, pp. 847–853, 1989.

[5] D. Mills, "Internet time synchronization: The network time protocol," *IEEE Transactions on Communications*, vol. 39, no. 10, pp. 1482–1493, 1991.

[6] J. Agre and L. Clare, "An integrated architecture for cooperative sensing networks," *IEEE Computer*, vol. 33, pp. 106–108, May 2000.

[7] F. Schneider, "Towards fault-tolerant and secure agentry," in *11th International Workshop on Distributed Algorithms*, September 1997.

[8] T. Bass, "Intrusion detection systems and multisensor data fusion," *Communications of the ACM*, vol. 43, April 2000.

[9] D. Frincke, D. Tobin, J. McConnell, J. Marconi, and D. Polla, "A framework for cooperative intrusion detection," in *Proceedings of the 21st National Information System Security Conference*, 1998.

[10] OMG, "Object management group home page," 1999. `http://www.omg.org/techprocess/meetings/schedule/Fault_Tolerance_RFP.html`.

[11] P. Narasimhan, K. Kihlstrom, L. Moser, and P. Mellior-Smith, "Providing support for survivable CORBA applications with the Immune system," in *Proceedings of the Nineteenth International Conference on Distributed Systems*, IEEE, May 1999.

[12] OMG, "The Common Object Request Broker: Architecture and specification (revision 2.3.1)." OMG document formal/99-10-07, October 1999.

[13] R. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Transactions on Database Systems*, vol. 4, pp. 180–209, June 1979.

[14] D. Gifford, "Weighted voting for replicated data," in *Proceedings of the Seventh ACM Symposium on Operating Sysetms Principles*, pp. 150–159, 1979.

[15] D. Davcev, "A dynamic voting scheme in distributed systems," *IEEE Transactions on Software Engineering*, vol. 15, pp. 93–97, Jan. 1989.

[16] S. Cheung, M. Ahamad, and M. Ammar, "Mulit-dimensional voting: A general method for implementing synchronization in distributed systems," in *Proceedings of the Tenth International Conference on Distributed Systems*, pp. 362–369, 1990.

[17] J. Paris, "Voting with witness: A consistency scheme for replicated files," in *Proceedings of the Sixth International Conference on Distributed Computing Systems, IEEE*, pp. 606–612, 1986.

[18] D. Malkhi and M. Reiter, "An architecture for survivable coordination in large distributed systems," *IEEE Transactions on Knowledge and Data Engineering*, 1999.

[19] BBN, "Cronus system/subsystem specification," Tech. Rep. 5884, BBN Systems and Technologies, 1981.

[20] R. Gurwitz, M. Dean, and R. Schantz, "Programming support in the Cronus distributed operating system," in *Proceedings of the Sixth International Conference on Distributed Computing Systems*, May 1986.

[21] R. Schantz, R. Thomas, and G. Bono, "The architecture of the Cronus distributed operating system," in *Proceedings of the Sixth International Conference on Distributed Computing System*, May 1986.

[22] M. Little and S. Shrivastava, "Using appilcation specific knowledge for configuring object replicas," in *Proceedings of the Third International Conference on Configurable Distributed Systems*, pp. 136–143, 1996.

[23] A. Team, "The arjuna system programmer's guide: Public release 3.0," tech. rep., Department of Computing Science, University of Newcastle UponTyne, 1994. http://arjuna.ncl.ac.uk.

[24] M. Little and D. McCue, "The replica management sytem: A scheme for flexible and dynamic repilcation," in *Proceedings of the Second International Workshop on Configurable Distributed Systems*, March 1994.

[25] G. York, D. Siewiorck, and Z. Segall, "Software voting in asynchronous NMR computer structures," Tech. Rep. CMU CS 83 128, Carnegie-Mellon University, Department of Computer Science, 1983.

[26] J. Wensley, L. Lamport, J. Goldberg, M. Green, K. Levitt, P. Melliar-Smith, R. Shostak, and C. Weinstock, "SIFT: Design and analysis of a fault-tolerant computer for aircraft control," *Proceedings of the IEEE*, vol. 66, pp. 1240–1255, October 1978.

[27] D. Powell, "Delta-4: A generic architecture for dependable distributed computing," in *ESPRIT Research Report*, vol. 1, Springer-Verlag, 1991.

[28] D. Powell, "Lessons learned from Delta-4," *IEEE Micro*, vol. 14, no. 4, pp. 36–47, 1994.

[29] M. Reiter and K. Birman, "How to securely replicate services," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 986–1009, May 1994.

[30] M. Reiter, "The rampart toolkit for building high-integrity services," *Theory and Practice in Distributed Systems*, pp. 99–110, 1995. (Lecture Notes in Computer Science 938.).

[31] M. Reiter, "A secure group membership protocol," *IEEE Transactions on Software Engineering*, vol. 22, pp. 31–42, January 1996.

[32] S. Maffeis, *Run-Time Support for Object-Oriented Distributed Systems with CORBA*. PhD thesis, University of Zurich, 1995.

[33] S. Landis and S. Maffeis, "Building reliable distributed systems with CORBA," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 31–43, 1997.

[34] L. Moser, P. Mellior-Smith, and P. Narasimhan, "Consistent object replication in the eternal system," *Theory and Practice of Object Systems*, vol. 4, no. 2, 1998.

[35] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. Sanders, D. Bakken, M. Berman, D. Karr, and R. Schantz, "AQuA: An adaptive architecture that provides dependable distributed objects," in *Proceedings of the Seventeenth Symposium on Reliable Distributed Systems (SRDS-17, IEEE*, (West Lafayette, IN), October 1998.

[36] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operations," in *Proceedings of the 8th International Symposium on Fault-Tolerance Computing Systems (FTCS-8), IEEE*, pp. 3–9, 1978.

[37] A. Avizienis, "The N-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, vol. 11, pp. 1491–1501, Dec. 1985.

[38] J. Brilliant, J. Knight, and N. Leveson, "The consistent comparison problem in N-version software," *IEEE Transactions on Software Engineering*, vol. 15, pp. 1481–1485, Nov. 1989.

[39] B. Parhami, "Voting algorithms," *IEEE Transactions on Reliability*, vol. 43, pp. 617–629, December 1994.

[40] D. Blough and G. Sullivan, "Voting using predispositions," *IEEE Transactions on Reliability*, vol. 43, pp. 604–616, December 1994.

[41] OMG, "Object management group home page." `http://www.omg.org`, 1999.

[42] M. Corporation, "Understanding the distributed object component model (DCOM) architecture." `http://www.microsoft.com/NTServer/appservice/ techdetails/prodarch/DCOM/`, 1999.

[43] International Middleware Association, "International middlware association(IMWA)." `http://www.moma-inc.org`, 1999.

[44] IBM, "Mqseries family." `http://www-4.ibm.com/software/ts/mqseries`, 1999.

[45] U. of Utah, "Flick: The flexible idl compiler kit," 1999. `http://www.cs.utah.edu/projects/flux/flick/`.

[46] J. Zinky, D. Bakken, and R. Schantz, "Architectural support for quality of service for CORBA objects," *Theory and Practice of Object Systems (Special Issue on CORBA and OMG)*, vol. 3, April 1997.

[47] J. Loyall, D. Bakken, R. Schantz, J. Zinky, D. Karr, and R. Vanegas, "QoS aspect languages and their run-time interactions in languages, compilers, and run-time systems for scalable computers," in *Letcure Notes in Computer Science 1511* (O. David, ed.), Springer-Verlag, 1998.

[48] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budia, and Y. Minsky, "Bimodal multicast," *ACM Transactions on Computer Systems*, vol. 17, May. 1999.

[49] R. Schantz, J. Zinky, D. Karr, D. Bakken, J. Megquier, and J. Loyall, "An object-level gateway supporting integrated-property quality of service," in *Proceedings of the Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 1999.

[50] OMG, "Fault-tolerant CORBA: Joint revised submission." OMG document orbos/99-12-08, December 1999.

[51] F. Cristian, "Exception handling and software-fault tolerance," in *Proceedings of the Tenth International Symposium on Fault-Tolerant Computing*, pp. 97–103, IEEE, 1980.

[52] F. Cristian, "Correct and robust programs," *IEEE Transactions on Software Engineering*, vol. 10, pp. 163–174, March 1984.